# El manual para el clustering con openMosix

**miKeL a.k.a.mc$^2$** ≡ Miquel Catalán i Coït

Versión 1.0
6 de septiembre de 2004

Este manual está dedicado a todos aquellos
que han hecho, hacen y harán que haya algo que
documentar. A todos los desarrolladores del proyecto
openMosix, muchas gracias.

Menciones especiales para:

**Louis Zechtzer**
**Martin Høy**
**Brian Pontz**
**Bruce Knox**
**Matthew Brichacek**
**Matthias Rechenburg**
**Maurizio Davini**
**Michael Farnbach**
**Mark Veltzer**
**Muli Ben Yehuda (a.k.a. mulix)**
**David Santo Orcero (a.k.a. irbis)**
**Moshe Bar** coordinador, autor de MFS y DFSA

Este manual ha estado posible gracias a las importantes contribuciones de:

**Carlos Manzanedo y Jordi Polo**. Ambos han aportado texto en los capítulos sobre generalidades de la supercomputación. También del apéndice sobre GNU/Linux. Ingenieros Informáticos por la *Universidad de Alcalá de Henares* (Madrid).
Extractos de la documentación de su proyecto de final de carrera *Clusters para Linux* (septiembre 2001).

**David Santo Orcero** (irbis). Aportaciones desde su inmenso archivo personal que cubren las secciones de instalación, ajustes y herramientas de usuario de openMosix. La relevancia de dichas aportaciones se deben a que es el desarrolaldor de las mismas.
Ingeniero Informático por la *Escuela Técnica Superior de Ingeniería Informática* de Málaga.

Asimismo agradezco las colaboraciones de:

**Ana Pérez Arteaga** correcciones ortográficas
**C. W. Strommer** traducción de las PMF (preguntas más frecuentes)
**Jaime Perea** capítulo sobre PCMCIA
**Marcelo Stutz** acceso a openMosixview con `ssh` y el *Stress-Test*
**Ross Moore** gracias por contestar todas mis dudas sobre `latex2html`



Todos nosotros nos hemos esforzado y lo haremos para que a usuarios como tú les sea útil esta guía, por ello te animamos a hacernos llegar cualquier ambigüedad, error o consejo para poder mejorarla.
También a tí que has sabido ver el poder de la comunidad libre y vas a convertir tus PCs en un supercomputador, gracias.

Lo primero que sueles preguntarte cuando un libro se pone a tu alcance es si valdrá la pena pasar tiempo leyéndolo. Y las dudas parecen ser directamente proporcionales al tamaño del mismo.
Sea cual haya sido la referencia que te haya llevado hasta aquí, si has llegado ha sido porque ya te has preguntado: *¿puedo disponer de la potencia de todas mis computadoras a la vez?*
Igualmente, sea cual haya sido la razón que te ha motivado a pensar en montar tu propio sistema cluster, sabrás que no es fácil andar solo el camino.
Montar un supercomputador no es trivial y los autores -y la comunidad que ha querido contribuir en esta documentación- pretenden acompañarte en tu andadura.

...porque no eres el único cansado de la ley de Moore, porque alguien más piensa que en la unión está la fuerza, porque alguien tiene que hacer el trabajo sucio: **sigue leyendo.**

# § ABSTRACT

Los sistemas cluster hace años que fueron diseñados, la computación paralela y distribuida no es ninguna novedad en el año 2004. No obstante no había sido hasta ahora que el usuario había empezado a necesitarlas. La tecnología del silicio está llegando a sus postrimerías y los computadores cuánticos aún están en fase de desarrollo.

Mientras grandes empresas, instituciones y universidades selectas disponen -desde hace años- de grandes computadoras superescalares, el usuario había estado relegado -al menos hasta ahora- a máquinas SMP en el mejor de los casos.

Pero todo esto está canviando: la demanda de rendimiento no puede ser suplida por la arquitectura monoprocesador y menos por los x86 compatibles. La solución que han adoptado los fabricantes ha estado saltar a arquitecturas de 64 bits o aumentar más aún la frecuencia de bus. Son desplazamientos del mismo problema en el tiempo.

En este marco toman mayor importancia los clusters, concebidos para proporcionar cálculo paralelo con componentes habituales en el mercado. Estaciones de trabajo conectadas por red trabajando de forma cooperativa que permiten aumentar notablemente las prestaciones de todas ellas por separado.

En este documento se han desarrollado diversos métodos para llegar a construir -y mantener- un cluster openMosix a partir de cero, es decir, desdel hardware.

La documentación aportada deja claras las grandes capacidades tecnológicas de openMosix, que se podrán aprovechar en proyectos de pequeña, mediana o gran dimensión gracias a su escalabilidad y flexibilidad. También se enfatiza en la importancia de la aplicación de las tecnologías de programario libre como mejor solución para poner en manos del usuario las mejores herramientas que posibilitarán un futuro enriquecedor tanto tecnológica como socialmente. Este entorno es el que mejor defiende la propia congruencia de intenciones ya sea en la lógica docente, donde priman -o deberían hacerlo- el conocimiento y su libre difusión, o dentro de la lógica empresarial -donde se prioriza el beneficio al menor coste posible-.

PALABRAS CLAVE: **supercomputación, cluster, gnu/linux, openmosix, ssi, diskless.**

# Índice de figuras

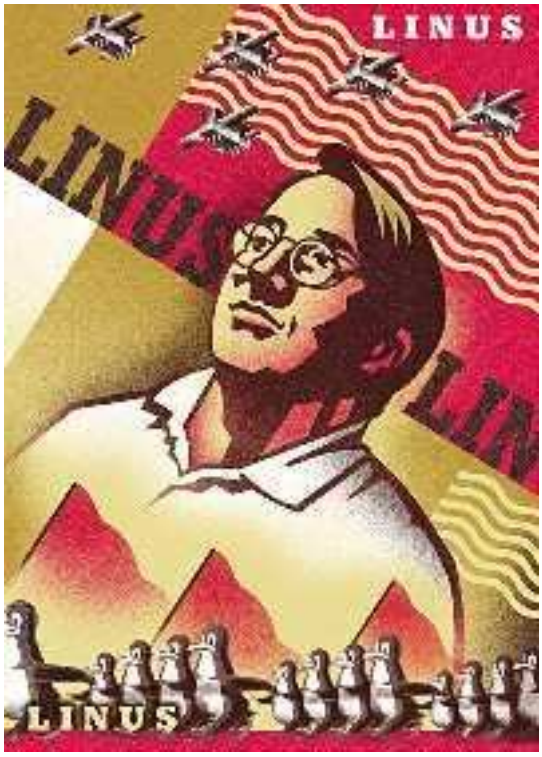
# Índice de cuadros

# Índice general

# Capítulo 1

# Presentación

## 1.1. PRELIMINARES



*Now this is not the end. It's not even the beginning of the end.*
*But it's, perhaps, the end of the beginning.*

*Winston Churchill*

Primero fue MOSIX, ahora es openMosix, un proyecto mucho más interesante no sólo desde un punto de vista técnico sino porque se han mejorado los términos de la licencia que mantenía a MOSIX bajo código propietario.

Este manual está dirigido a conocer el proyecto openMosix y no MOSIX por la simple razón que el primero tiene un sector de usuarios mucho más amplio y con mayores garantías de crecer en los próximos tiempos (Moshe Bar estima que el 97 % de los usuarios de la antigua comunidad MOSIX migraron a openMosix durante el primer año, 2002).

Parte de los capítulos que aquí se presentan pertenecen íntegramente a la literatura que Kris Buytaert ha escrito en el *The openMosix Howto*. Se ha añadido no obstante otra documentación (escrita o traducida) llegada de las personas que se han querido sumar a este proyecto de documentación, a los cuales ya se ha hecho referencia.

Intentado abarcar el mayor abanico de usuarios se ha organizado el texto en términos de complejidad creciente. Esperamos que esto suponga una ventaja a la gran mayoría de lectores para que podáis ahondar y ampliar conocimientos sobre openMosix y, como no, en GNU/Linux.

Nada une ya a MOSIX y openMosix, e intentar buscarles parecidos resultará, como los grandes avances en el proyecto demuestran, cada vez más difícil. Este manual no es la documentación del proyecto MOSIX.

### 1.1.1. Sobre este documento

Este documento te dará una amplia descripción de **openMosix, un paquete software que posibilita que una red de computadoras basadas en GNU/Linux funcionen como un cluster**.

A lo largo de este camino que empezaremos juntos se introducirán conceptos como la computación paralela, breves tutoriales para programas que tengan utilidades especiales para las posibilidades que openMosix pueda ofrecerte, e incluso un repaso histórico sobre los inicios del clustering como nueva alternativa en la supercomputación. Será importante saber con qué nos estamos manejando y conocer también por qué la computación masiva está tirando hacia esta dirección.

Kris Buytaert escribió el HOWTO original en febrero de 2002, cuando Scot Stevenson buscaba a alguien para llevar a cabo este trabajo de documentación. Esta versión en castellano fue iniciada por miKeL a.k.a.mc2 en el mismo año como parte del trabajo de final de carrera en la EPS (Escola Politcnica Superior, Lleida, Espaa) para la Ingenier´ ia Técnica Informática. El contenido del *howto* oficial se traducirá aquí, en ocasiones ampliado.

### 1.1.2. Limitación de responsabilidad

Utilice la información contenida en este documento siendo el único responsable del riesgo que puedan correr sus equipos. Yo mismo y el equipo de colaboradores repudiamos cualquier responsabilidad sobre las consecuencias que el seguimiento de estos contenidos puedan provocar.

El seguimiento de los ejemplos aquí descritos corren a cargo del lector.

Es recomendable hacer copias de seguridad (*backups*) de su sistema antes de iniciar cualquier instalación, ya que el trabajo desde `root` (administrador de su equipo UNIX) puede provocar pérdidas y/o modificaciones irreversibles de su información.

### 1.1.3. Política de distribución

### 1.1.4. Nuevas versiones de este documento

Las versiones oficiales de este documento serán hospedadas en LuCAS[1] y en mi propia web[2].

Los borradores de la documentación oficial (en inglés) se encontrarán en la web de Kris Buytaert[3] en el subdirectorio apropiado. Los cambios en este documento normalmente serán anunciados en las listas de distribución de openMosix. Los posibles cambios en ésta, la versión en castellano, serán igualmente anunciados en la citada lista y podrás obtenerla en mi sitio web en los formatos PS, PDF o HTML4.0.

### 1.1.5. Mantenimiento

Actualmente este manual está mantenido por miKeL a.k.a.mc2 (Miquel Catalán i Coït), por favor manda tus dudas o preguntas a la dirección de correo electrónico que encontrarás en su sitio web.

Para dudas concretas sobre la tecnología openMosix, por favor dirígete a las listas de correo del proyecto (ver sección *Para más información*).

[1] `http://lucas.hispalinux.es/`
[2] `http://como.akamc2.net`
[3] `http://howto.ipng.be/Mosix-HOWTO/`

## 1.2. ORGANIZACIÓN

*El manual para el clustering con openMosix* ha sido dividido en nueve capítulos, a la vez divididos en diversas secciones. Seguidamente se expone un breve resumen de cada una de ellas.

**CAPITULO I: Presentación**

- **Sección 1.1: Preliminares.** Se define la limitación de responsabilidad de los autores respecto el seguimiento de los ejemplos aquí contenidos y se indica dónde encontrar nuevas versiones de este documento.
- **Sección 1.2: Organización.** Esta sección.

**CAPITULO II: Teoria de la supercomputación**

- **Sección 2.1: Introducción.** Se introduce al lector en el contexto histórico-económico en que se desarrollan las diferentes arquitecturas paralelas. Se dan las bases para comprender qué objetivos se persiguen aquí .
- **Sección 2.2: Paralelismo.** Conceptos teóricos sobre el modelo matemático existente tras la paralelización de procesos. Se explican algunos límites conocidos.
- **Sección 2.3: Arquitecturas.** Construcciones físicas que pueden implementarse con los componentes de que debe disponer todo sistema basado en el esquema de Von Newman.
- **Sección 2.4: Sistemas distribuidos.** Se especifica en un tipo de arquitectura, consistente en distribuir los recursos entre varias estaciones de trabajo. Ésta será la arquitectura sobre la que se centrará el clustering en los siguientes capítulos.

**CAPITULO III: Implementación de los Sistemas Distribuidos**

- **Sección 3.1: Sistemas operativos.** Nociones sobre los principales módulos de que se componen. Útil para entender el rol de la migración, la problemática que aporta y, consecuentemente, qué componentes deberán ser modificados para dotarlos de este servicio.
- **Sección 3.2: La importancia de la red.** Uno de los recursos que marcará el rendimiento de nuestra arquitectura será la interconnectividad entre nodos.

**CAPITULO IV: Clusters**

- **Sección 4.1: Nociones generales.**
- **Sección 4.2: Clusters HA.** Clusters de alta disponibilidad.
- **Sección 4.3: Clusters HP.** Clusters de alto rendimiento.
- **Sección 4.4: Requerimientos y planteamientos.** Construir un cluster no es algo trivial, así que habrá que tener en cuenta ciertos aspectos tanto a nivel físico como de programario.

**CAPITULO V: Clustering con openMosix**

- **Sección 5.1: ¿Qué es realmente openMosix?** Se expone la situación de los clusters y la necesidad de openMosix dentro del marco de la supercomputación. Se da un ejemplo de como puede sernos útil un cluster en la vida cuotidiana.
- **Sección 5.2: Características de openMosix.** Se divide el concepto de openMosix en sus cuatro subsistemas. Se analizan brevemente sus pros y contras así como la política que se ha implementado para conseguir la migración de procesos.
- **Sección 5.3: Instalación de un cluster openMosix.** Se dan los pasos para llegar a convertir un PC en el nodo de un cluster. Este capítulo ha sido redactado íntegramente por el Dr. David Santo Orcero, uno de los desarrolladores de openMosix.
- **Sección 5.4: Administración del cluster.** Una vez hecha la instalación convendrá saber administrarla tanto para adaptarla a necesidades concretas como para evitar agujeros de seguridad.

- **Sección 5.5: Ajustes en el cluster.** Si algo debe preocupar en la computación paralela es el rendimientos que proporciona el sistema. Aquí se exponen algunas pautas a seguir para comprobar que todo funciona a pleno rendimiento.
- **Sección 5.6: openMosixview.** Nunca está de más disponer de un interfaz gráfico para podernos manejar más intuitivamente.
- **Sección 5.7: Problemas más comunes.** Seguramente habrá algún percance durante el proceso. Aquí se intentan cubrir los puntos conflictivos.
- **Sección 5.8: Para más información.** El proyecto openMosix es un proyecto muy vivo y como tal tiene una magnífica comunidad de seguidores que estaremos encantados de responder tus dudas en la listas de distribución o foros.

**CAPITULO VI: openMosix a fondo**
Comentarios sobre el código fuente.

**CAPITULO VII: Tutoriales para casos especiales**

- **Sección 7.1: Nodos sin discos.** Generalmente el cluster lo conformaremos con computadoras de sobremesa con todo el hardware para funcionar independientemente. No obstante, ésta no es la única alternativa.
- **Sección 7.2: ROMs para arranque sin discos.** Parte relativa a la sección anterior. Aquí se explica como construir las roms necesarias.
- **Sección 7.3: *Live Linux CD!* Linux en un cdrom.** La potencia y flexibilidad de las *metadistros* puede también obtenerse a partir de una instalación hecha. Aquí se expone un método para pasar una instalación de disco duro a cdrom arrancable.

**CAPITULO VIII: APÉNDICES.** En los apéndices se incluye información adicional sobre los temas que se han tratado en el documento. o aclarar el formato de algún fichero.

- **Sección 8.1: Aplicaciones funcionando, o no.** Clasificación de las aplicaciones más utilizadas según permitan migración de procesos o no.
- **Sección 8.2: Salidas de comandos y ficheros.** Formato de diferentes ficheros y formas de configuración.
- **Sección 8.3: Acrónimos.** Útiles para conocer cualquier acrónimo aparecido.

**CAPITULO IX: GNU Free Documentation License**. Cláusulas que rigen el uso de este documento.

# Capítulo 2

# Teoria de la supercomputación

## 2.1. INTRODUCCIÓN

*Your theory is crazy,*
*but it's not crazy enough to be true.*

*Niels Bohr*

### 2.1.1. Visión histórica

En lo que se refiere a la capacidad de procesamiento, existen varias alternativas para el futuro. Actualmente la capacidad de integración y el abaratamiento de las tecnologías permite que casi cualquier empresa pueda contar con una capacidad de cómputo antes inimaginable para las tareas que necesita. Se prevé que la capacidad de integración llegue a un techo tecnológico, en el cual se necesite un nuevo paradigma para poder seguir incrementando la capacidad de procesamiento de las máquinas. Uno de esos paradigmas es el procesamiento paralelo.

Por **procesamiento paralelo** se entiende **la capacidad de utilizar varios elementos de proceso para ejecutar diferentes partes del mismo programa simultáneamente.**

La resolución de problemas mediante procesamiento paralelo no es nueva, está basada en el viejo y conocido método de *divide y vencerás* utilizado para resolver problemas de carácter computacional.

Un analogía para explicar las ventajas y límites de este método es la siguiente: se ha decidido ordenar una biblioteca mediante el criterio tipo y autor. Una solución sería separar todos los libros por su tipo en pilas y luego que una sola persona ordenara cada uno de esas pilas por el nombre de su autor. En la resolución paralela del problema se añadiría una segunda persona, de manera que cada persona catalogase según el tipo la mitad de la biblioteca, tardándose la mitad de tiempo en esta fase, y luego que cada uno fuese colocando las pilas de los tipos por autor. La solución paralela obtiene como ventaja, en este caso, la reducción del tiempo a la mitad para solucionar el problema. ¿Qué sucedería si se añadiesen más personas dedicadas a catalogar la biblioteca? En un principio, cuantas más personas trabajen en el proceso, antes acabará éste, es decir, existe una relacción lineal entre el tiempo de resolución del problema y el número de personas que trabajan en la ordenación de la biblioteca. Pero por otro lado, parece estúpido contratar a 200 personas para colocar una biblioteca de 200 libros.

Esta analogía muestra las ventajas que puede tener la resolución de un problema mediante el procesamiento paralelo, pero también muestra los límites en la resolución.

Relativo al mundo de la tecnología y al campo de los procesadores en general, se descubrió que las arquitecturas paralelas podían solventar de manera más rápida cierto tipo de problemas. Desde 1955 personas como Gene Amdahl[1] han investigado en el campo de arquitecturas paralelas obteniendo aquellos parámetros que optimizaban las arquitecturas así como aquellos que hacían que la relación coste-rendimiento aumentase. Empresas como IBM, DEC y desde luego muchas otras organizaciones como el MIT, se llevan interesando en la computación paralela desde las décadas de los 50-60, y de hecho siguen investigando y obteniendo resultados en la actualidad, hasta el punto en que prácticamente todos los ordenadores que existen actualmente en el mercado explotan de una u otra manera soluciones paralelas.

En la década de los 80, la compartición de recursos mediante redes de computadores hizo posible un nuevo planteamiento para aprovechar no solo recursos como capacidad de almacenamiento o capacidad de impresión, sino para utilizar ciclos de CPU de otras máquinas conectadas a la red (los llamados multicomputadores). En los 70 y a primeros de los 80, personas como Bruce J.Nelson[2] de Xerox expusieron trabajos teóricos de cómo se podía utilizar mediante software esta capacidad de procesamiento paralelo que hasta ahora estaba relegada principalmente al hardware, limitándose el software a aprovecharlo mediante técnicas de programación explícita. En 1985, Intel produjo el primer iPSC/1. Este multicomputador era una combinación de muchos 80286 conectados en una topología hipercubo a través de controladoras ethernet, mostrando que era real y posible utilizar este tipo de redes para explotar los sistemas paralelos.

[1] En aquellos momentos trabajaba en IBM como principal diseñador de la arquitectura del 704, este ordenador fue el primer ordenador comercial en tener unidad de coma flotante, y tenía un rendimiento de unos 5Kflops.

[2] Propuso en 1981 una primera descripción de lo que se llamaría RPC (Remote Procedure Call), que permitió crear luego infinidad de aplicaciones distribuida s asícomo sistemas distribuidos.

En la década de los 90, el uso de las redes de computadores se extendió de manera exagerada en comparación a otros campos como el de sistemas operativos o el de arquitectura de computadores.

Otro concepto importante que no se debe confundir con el de *paralelo* es el término *concurrente*. Edger Dijstra en 1965 describió el problema de las regiones críticas en las arquitecturas paralelas y como solucionarlo mediante semáforos (solucionado en 1968), pero fue en 1975 cuando introdujo el concepto de concurrencia, basándose en sus trabajos anteriores. Actualmente la implementación de concurrencia la realizan muchos lenguajes de programación.

Si por paralelismo se entienden procesos que se ejecutan en varios elementos de proceso para llegar la resolución conjunta de un problema, por concurrencia se entiende procesos que se ejecutan de manera independiente en un mismo procesador, para la resolución de uno o varios problemas (*multitarea*). Un ejemplo de concurrencia lo tenemos en prácticamente todos los sistemas operativos que se utilizan en la actualidad, puesto que comparten el mismo procesador para prestar un único servicio al usuario.

Otro ejemplo sería la utilización de un programa concurrente en el cual dos procesos solucionen un problema único[3]. El uso de concurrencia y de paralelismos conllevan principalmente un problema de comunicación entre los elementos de proceso o los procesos entre sí para que la resolución de un problema mediante estos paradigmas sea viable.

### 2.1.2. Problemas que se pueden resolver con sistemas paralelos

Se pueden distinguir dos épocas en las cuales los problemas que han provocado la aparición de sistemas paralelos y distribuidos han sido diferentes:

- Por un lado las décadas de los 60-70-80[4], en las cuales el máximo problema era optimizar la capacidad de procesamiento, y de esta manera aumentar el rendimiento de las máquinas y la producción de éstas.
- Por otro lado, desde la década de los 90 hasta la actualidad, donde los problemas han aumentado: a los que existían en las décadas anteriores se han sumado los provocados por la red Internet y el fenómeno de la nueva economía.

Este último punto es sencillo de entender: la nueva economía está formada por comercios a imagen y semejanza de los de la tradicional, pero con las ventajas aportadas por el mundo de las máquinas. Son nuevas tiendas y negocios que funcionan 24 horas al día 7 días a la semana, que no necesitan de personal, excepto técnico, para su puesta en marcha y al que se accede a través de Internet. Con este nuevo tipo de negocio, muchas empresas hacen inversiones en equipo y personal técnico, para ofrecer a nivel mundial soluciones que de otra manera podrían ser inviables por precio, tiempo u organización. Las empresas exigen a esta nuevas tecnologías, lo mismo que han exigido siempre a las antiguas:

- Máximo rendimiento, mínimo coste. Intentando hacer lo imposible por que las inversiones realizadas sean amortizadas sin desperdiciar ningún recurso.
- Máximo aprovechamiento de los recursos existentes.
- Disponibilidad máxima. En en un negocio tradicional si uno de los trabajadores se pone enfermo, se intenta cubrir esta vacante con otro trabajador que satisfaga el trabajo. Con las nuevas tecnologías sucede lo mismo, se han creado infinidad de soluciones para evitar cierres de negocios temporales mediante UPS (para las caídas de luz), fuentes redundantes, equipos redundantes y otras muchas técnicas dedicadas a cubrir por completo el término *alta disponibilidad*.
- Confiabilidad máxima. Sabiendo que el sistema se va a comportar de la manera que se espera de él.
- Adaptación a los cambios. Tanto en forma de carga para el sistema como en forma de nuevo planteamiento del negocio. El sistema debe ser flexible y escalable.

Este último punto es importante por motivos claramente económicos (no solo a nivel de empresa) y supone un gran reto en el diseño de sistemas para que estos puedan adaptarse de manera eficiente a nuevas exigencias. Hablamos de un término muy importante que se utilizará a lo largo de todo el documento, la *escalabilidad*. Véase

[3] Por ejemplo la adquisición de datos mediante varios sensores en procesos independientes y su procesamiento.
[4] En los 50 la optimización de los sistemas dependian de manera casi única de los avances tecnológicos más que de los téoricos.

en un ejemplo. Una empresa quiere poner un negocio en Internet, contratan un asesor técnico que les explica que para lo que ellos quieren hacer necesitarán una capacidad de proceso equivalente al número máximo de clientes potenciales que creen que sus productos pueden acaparar en el mercado. La empresa compra los ordenadores que poseen dicha capacidad de proceso, sabiendo que éstos cumplirán con las expectativas iniciales planteadas, de manera que todos los recursos invertidos estarán siendo utilizados de manera continua.

Pasado un tiempo de continua prosperidad empresarial, la empresa se da cuenta de que el sistema se quedó pequeño para el volumen de ventas, vuelven a contactar con el asesor técnico y este les explica que la única opción es comprar un segundo sistema, esta vez el doble de potente (y varias veces más costoso).

La empresa decide negarse porque la inversión realizada en el primer equipo aún está por amortizarse, además del gasto inútil que habrían realizado en el primer equipo, que en un principio quedaría inoperativo después del cambio.

He aquí que la competencia decide invertir en otro sistema más potente y mejor diseñado (no necesariamente más caro), con lo que da mejor servicio a los clientes y en poco tiempo provoca la quiebra de la primera. Ésta pues decide intentar dedicarse a otro sector, en el cual necesitarán nuevas tecnologías. Llaman a otro asesor técnico (esta vez mucho más listo que el anterior), que les explica como podrían reutilizar componentes del anterior sistema ahorrándose la inversión inicial para el nuevo proyecto.

Este ejemplo, algo drástico, refleja la realidad de muchas empresas que han quebrado por su popularidad y por su incapacidad de crecimiento. En cualquier caso, no es más que un ejemplo para introducir un concepto de escalabilidad. Ilustra la ventajas de sistemas fácilmente escalables como pueden ser cluster con respecto a otros, no tan fácilmente escalables, como pueden ser *mainframes* y otros supercomputadores vectoriales.

La definición de escalabilidad más apropiada a los términos que se referirá en este documento es: un sistema se dice escalable si es capaz de escalar, es decir, de incrementar sus recursos y rendimiento a las necesidades solicitadas de manera efectiva o, en el caso de *scale down*, reducir costes.

Aunque la mayoría de las veces se habla de *escalar hacia arriba*, es decir de hacer el sistema más grande, no es siempre necesario. Muchas veces interesa hacer el sistema más pequeño pudiendo reutilizar los componentes excluidos. Que un sistema sea escalable implica:

1. Funcionalidad y rendimiento. Si un sistema escala, mejora su rendimiento, de manera que de forma ideal, al aumentar en `N` el número de elementos de proceso del sistema éste debe aumentar en `N` el rendimiento.

2. Escalabilidad en coste. De lo anterior se deduce que idealmente el coste de la escalabilidad de 1 a `N` en un sistema lleve un coste de `N` por el coste de un procesador. La escalabilidad perfecta es lineal, si una potencia 10 veces superior nos cuesta 15 veces más, el sistema no está escalando bien.

3. Compatibilidad de componentes. De manera que la inclusión o exclusión de componentes en el sistema no suponga la inutilización, infrautilización o coste adicional en los componentes.

Con todo esto queda patente que tener un alto factor de **escalabilidad es un requisito interesante para cualquier sistema.**

También es importante hacer notar que los sistemas distribuidos (y otros sistemas paralelos) son, por ahora, los sistemas que más se acercan a la escabilidad lineal. En el ejemplo anterior (supercomputador, mainframe) realmente un equipo el doble de potente no vale el doble sino varias veces más; en cambio en sistemas distribuidos al doble de precio se consigue mejor relación. Esta relación de precios puede verse en los precios de microprocesadores: costar el doble no significa el doble de potencia, sino que los precios siguen una curva exponencial según aumentan sus prestaciones.

### 2.1.3. Soluciones actuales que se dan a dichos problemas

Respecto a la evolución de los sistemas y con objeto de obtener mayor capacidad de procesamiento, el paralelismo a todos los niveles ha sido una de las soluciones más utilizadas, de hecho en la actualidad, la práctica totalidad de los ordenadores y microprocesadores explotan de una manera u otra tecnologías paralelas, ya sea en multiprocesadores, en multicomputadores o en procesadores independientes[5]MX en los procesadores Intel, 3DNow! en los AMD, Altivec en la arquitectura PPC, entre otras.

[5] M

Respecto a los requisitos de *alta disponibilidad* que requieren las soluciones actuales se proponen soluciones como UPS, generadores redundantes, hardware redundante[6], soluciones software de alta disponibilidad como la tecnología HA-Linux u otros clusters de este tipo de otras compañías como Piranha de RedHat, Va-Linux, Compaq, Sun... prácticamente todas las grandes marcas de ordenadores han sacado su sistema cluster específico al mercado.

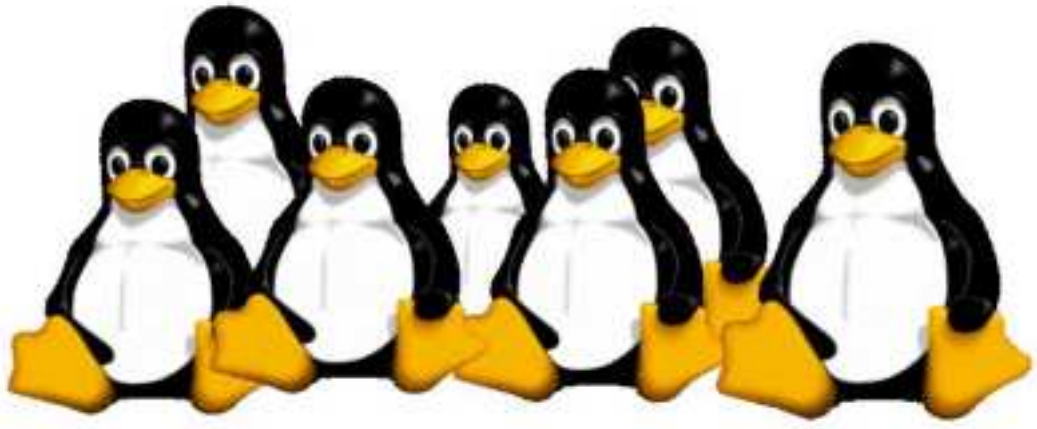
Otra solución existente es el *balanceo de carga*. Este tipo de cluster es muy utilizado en conjunción con los de alta disponibilidad, sobre todo en las conexiones de servidores a Internet, de manera que varios servidores dan un único servicio de manera transparente a sus usuarios.

Otro balanceo distinto, relativo a los clusters de alto rendimiento y a la migración de procesos, es el ofrecido por los llamados multicomputadores, o los sistemas distribuidos. En general estos son los sistemas que se acercan más a el concepto de sistema operativo distribuido del que se hablará más adelante. Clusters o sistemas de este tipo son openMosix, Beowulf y otros en el mundo del software libre y otros tantos en el del software propietario. A otra escala existen los paquetes software que permiten construir *grids* con todo el tratamiento de certificados y permisos de usuario necesarios. Estos sistemas no se tratarán en este manual.

Como se puede ver existen una infinidad de soluciones distintas para cada sección de problema concreto, no obstante no existe un sistema de carácter general que resuelva todos los problemas planteados debido a las dispares finalidades con que se diseñan.

[6] Mainboards y microprocesadores que replican el procesamiento que hace su homólogo a los datos de manera que cuando uno de los dos cae, el otro toma el trabajo y lo continúa de manera transparente e intenta solucionar, si es que puede, los problemas de su espejo. Supone un alto precio pero una gran disponibilidad.

## 2.2. PARALELISMO



*Software is like sex: it's better when it's free.*

*Linus Torvalds*

En esta sección se tratará el palalelismo a nivel teórico. Como se explica en la sección de introducción, paralelizar los problemas en problemas más sencillos y distribuirlos entre varios procesadores no siempre es una solución conveniente y dependerá en gran medida de la naturaleza de los problemas que se quieran tratar.

Se hablará de programación paralela y los problemas que pueden surgir cuando se trata de utilizar este paradigma. Para ello debe considerarse el sistema sobre el que se programa y se lanzan los programas paralelos: ya sean multiprocesadores o, como será el caso, multicomputadores. Al explicar estos límites que impone la propia arquitectura tendrá que hablarse forzosamente de la **ley de Ahmdal** así como de otros problemas relativos a la ordenación de instrucciones debido a dependencias.

En general en esta sección se hablará acerca de todos los conceptos que interesan a la hora de programar aplicaciones y hacerlo utilizando de una manera u otra el paralelismo. Se darán diversos ejemplos de programación con openMosix y explicaciones con PVM o MPI y se profundizará en conceptos como la **granularidad del paralelismo** o el **problema de la transparencia** en lo que se refiere a la programación.

### 2.2.1. Definiciones previas

**Parámetros de rendimiento en computación paralela**

**Velocidad de ejecución *rate (R)*.** Mide *salidas (outputs)* por unidad de tiempo. Según la naturaleza de las salidas, tendremos:

- $R = \frac{instrucciones}{segundo}$ que se miden con MIPS (instrucciones x $10^6$ por segundo).
- $R = \frac{op's}{segundo}$ que se miden con MOPS (operaciones x $10^6$ por segundo).
- $R = \frac{op's\,flotantes}{segundo}$ que se miden con MFLOPS (op's coma flotante x $10^6$ por segundo).

**Acceleración (*speedup* $S_p$).** Ratio entre el tiempo de una ejecución serie y una paralela.

$$S_p = \frac{t_1}{t_p} \tag{2.1}$$

$t_1$: tiempo requerido para realizar la computación en 1 procesador,

$t_p$: tiempo requerido para realizar la computación en $p$ procesadores.

Normalmente se da la relación $1 \leq S_p \leq p$, debido al tiempo que se pierde en la inicialización, sincronización, comunicación y otros factores de *overhead* que se requieren en la computación paralela.

**Eficiencia (E).** Ratio entre la acceleración de una ejecución paralela y el número de procesadores.

$$E = \frac{S_p}{p} = \frac{t_1}{p\,t_p} \tag{2.2}$$

**Redundancia (R).** Ratio entre el número de operaciones realizadas utilizando $p$ procesadores en una ejeciópn paralela y su correspondiente ejecución serie en 1 procesador.

$$R = \frac{O_p}{O_1} \tag{2.3}$$

**Utilización (U).** Ratio entre $O_p$ y el número de operaciones que podrian realizarse utilizando $p$ procesadores en $t_p$ unidades de tiempo.

$$U = \frac{O_p}{p\, t_p} \tag{2.4}$$

**Modelos matemáticos**

I. Principio Armónico Ponderado

Sea $t_n$ el tiempo requerido en computar $n$ operaciones de $k$ tipos diferentes, donde cada tipo de operación consiste en $n_i$ operaciones simples las cuales requieren $t_i$ segundos cada una para su ejecución. Luego:

$t_n = \sum_1^k n_i t_i$ y $n = \sum_1^k n_i$. Por definición sabemos que $R_n = \frac{n}{t_n} = \frac{n}{\sum_1^k n_i t_i}$ MFlops.

Si definimos:

$f_i = \frac{n_i}{n}$ como fracción de operaciones ejecutadas a una velocidad $R_i$,

$t_i = \frac{1}{R_i}$ velocidad de ejecución en computar 1 operación,

luego

$$R_n = \frac{1}{\sum_1^k f_i t_i} = \frac{1}{\sum_1^k \frac{f_i}{R_i}} \tag{2.5}$$

donde $\sum_i f_i = 1, R_i = \frac{1}{t_i}$.

Lo importante de esta ecuación es que muestra que una sola operación donde su ejecución no sea equiparable con el resto, en cuanto al tiempo de ejecución, tiene una gran influencia en el rendimiento global del sistema.

II. Ley de Amdahl

$$R(f) = \frac{1}{\frac{f}{R_H} + \frac{1-f}{R_L}} \tag{2.6}$$

Para comprender esta ley sería conveniente recuperar la analogía del problema que se planteaba en el primer tema, acerca de la persona que debía colocar los libros en dos partes. En el caso de la ordenación de la biblioteca se llegaba a la conclusión de que era estúpido contratar a 200 personas para que organizasen una biblioteca de 200 ejemplares, por la falta de eficiencia en las operaciones.

El *speedup* –o mejora de rendimiento– idealmente aumenta linealmente con el número de unidades de procesamiento que posea el sistema. Esto es en un principio cierto, pero estamos contando en todo momento con un programa modelo ideal, sin tener en cuenta la naturaleza de dicho programa. En el caso de programas concretos hemos de contar con la naturaleza del programa, qué pretende solucionar y la manera en la que lo soluciona. Eso permitirá poder dar con la medida real del *speedup*.

En cualquier programa paralelizado existen dos tipos de código: el código paralelizado y el código secuencial. Como es sabido existen ciertas secciones de código que ya sea por dependencias, por acceso a recursos únicos o por requerimientos del problema no pueden ser paralelizadas. Estas secciones conforman el código secuencial, que debe ser ejecutado por un solo elemento procesador. Es pues lógico afirmar que la mejora de rendimiento de un programa dependerá completamente de:

1. El tiempo en el que se ejecuta el código serie $R_L$.
2. El tiempo en el que se ejecuta el código paralelizable $R_H$.
3. El número $f$ de operaciones ejecutadas de forma paralela (en tiempo $R_H$).

Esta es la llamada **ley de Amdahl** y fue descrita por Gene Amdahl en 1967. Las implicaciones que trae esta ecuación son, a pesar de que no tenga en cuenta las características de cada sistema en concreto:


Figura 2.1: Paralelismo. Ejemplo de incremento de *speedup* obtenido con la ley de Amdahl

- el rendimiento no depende completamente del número de procesadores que posea el sistema: en la mayoría de los casos dependerá del número de procesadores máximo que se aprovecharán simultáneamente para ejecutar un programa.
- cuanto mejor paralelizado esté un programa más susceptible será de aumentar su *speedup* y por tanto explotar el rendimiento del sistema paralelo que lo ejecute.

Supongamos ahora que tenemos un programa que inicialmente no hemos paralelizado, cuyos tiempos de ejecución son 12 % y 88 %, en serie y en paralelo respectivamente.

Como se puede ver en la figura, la parte no paralelizable del código impide que se pueda escalar de forma lineal, llegará un momento que añadir nuevos procesadores no añadirá una ventaja real al sistema, porque todo lo que estará en ejecución será código secuencial. Por lo tanto para maximizar el aprovechamiento de los sistemas paralelos debe tenerse mucho cuidado con la forma de paralelizar las aplicaciones: cuanto más código secuencial tengan, más problemas de escalabilidad.

La ley de Amdahl es un caso particular del Principio Armónico Ponderado, anteriormente descrito. Si recuperamos:

$$R_n = \frac{1}{\sum_1^k \frac{f_i}{R_i}} \qquad \text{y hacemos } K = 2 \text{ queda} \qquad R_2 = \frac{1}{\sum_1^2 \frac{f_i}{R_i}}$$

si se igualan

$f_1 = f$: fracción paralela

$f_2 = (1 - f)$: fracción serie

da la forma de la Ec.(2.6) $\quad \rightarrow \quad R(f) = \dfrac{1}{\frac{f}{R_H} + \frac{1-f}{R_L}}$ .

### III. Límites de la computación paralela

Un programa paralelo requiere sincronizar las tareas de que se compone. Estas tareas se distribuyen entre los diferentes procesadores del sistema paralelo. La suma de los tiempo de ejecución de estas tareas en un sistema paralelo es más grande que la suma de los tiempos de ejecución en un solo proesador. Esto es debido a diferentes tiempos adicionales de procesamiento (*overhead*).

Sean:

$t_s$ : tiempo de sincronización

| **Parámetro** | $p \to \infty$ **,** $N$ **fijo** | $N \to \infty$ **,** $p$ **fijo** |
|---|---|---|
| $S_{N,p}$ | $\frac{N}{1+\frac{t_s+t_0}{t}}$ | $\frac{p}{1+\frac{t_0}{t}}$ |
| $E_{N,p}$ | 0 | $\frac{1}{1+\frac{t_0}{t}}$ |

Cuadro 2.1: Paralelismo. Límites de la computación paralela

$t$ : granularidad de la tarea. valor medio del tiempo de ejecución de las diferentes tareas.

$t_0$ : *overhead* de la tarea debido a su ejecución en paralelo.

$N$ : número de tareas entre puntos de sincronización.

$p$ : número de procesadores.

El tiempo requerido para ejecutar $N$ tareas, cadauna con un tiempo de ejecución $t$ en un solo procesador viene dado por

$$t_1 = N\,t$$

En un entorno paralelo cada tarea requiere $t + t_0$ unidades de tiempo para su ejecución. El tiempo requerido para ejecutar $N$ tareas en $p$ procesadores será pues:

$$t_{N,p} = t_s + \left\lceil \frac{N}{p} \right\rceil (t + t_0)$$

y recuperando el *speedup*, esta vez siendo el cómputo las $N$ tareas

$$S_{N,p} = \frac{T_1}{T_{N,p}} = \frac{N\,t}{t_s + \lceil \frac{N}{P} \rceil (t + t_0)} \tag{2.7}$$

y la *eficiencia* ahora quedaria determinada como

$$E_{N,p} = \frac{S_{N,p}}{p} \tag{2.8}$$

Los límites de la computación paralela estan reflejados en el cuadro 2.1.

### 2.2.2. Límites en el hardware

En lo que se refiere a explotación de programas paralelos en máquinas SMP hay que tener en cuenta el modo de ejecución de las instrucciones. Actualmente la mayoría de los ordenadores comerciales son de los llamados *ordenadores superescalares*: lanzan varias instrucciones a la vez aprovechando una división por fases. Esta división se construye sobre el principio que cada fase requiere de una parte específica del hardware del procesador para completarse. De esta manera se aprovechan mejor todo los componentes del procesador pues se utilizan a cada ciclo.

Para lograrlo deben solucionar de la manera más apropiada posible retardos por dependencias[7]. Generalmente esto se soluciona con renombramiento de registros (fase que puede o no hacer el compilador), lanzamiento de instrucciones fuera de orden, unidades superescalares y supervectoriales y algunas otras técnicas que se describirán sin ahondar en el capítulo dedicado a arquitecturas.

[7]Cualquiera de los tres tipos de dependencia que existen: dependencia real, dependencia inversa y antidependencia. Otra terminología utilizada es dominio-rango, rango-dominio y rango-rango.

La granularidad del hardware es la instrucción máquina (o en lenguaje ensamblador). Esta unidad es la más fina que puede tratar. Existen gran variedad de mejoras de rendimiento a los límites que se imponen por el programa, estas mejoras se van incorporando en los procesadores de última generación. El problema es que esta solución puede resultar en muchos casos demasiado cara, sobretodo por la dificultad o tiempos de implementación. Además siempre se necesita cierta ayuda de parte del software, principalmente el compilador. Así aunque el Pentium4 tenga nuevas instrucciones vectoriales SSEII, no son prácticamente usadas por los compiladores actuales y el rendimiento de un P4 es bastante inferior al esperado. En cambio usando el compilador especialmente desarrollado por Intel, los mismos programas aprovechan las instrucciones vectoriales del procesador mucho mejor con lo que se consiguen programas mucho más rápidos.

Hoy en día es difícil ver programar en ensamblador para procesadores que no estén dedicados a tareas como dispositivos empotrados o de tiempo real, e incluso en estos este lenguaje se hace más raro cada dia. Por lo tanto las optimizaciones de uso de las nuevas capacidades de los procesadores tienen que venir de parte del compilador que es quien se encarga de trabajar a nivel más bajo que el programador.

### 2.2.3. Límites del software

Como se ha visto, la ley de Amdahl pone límites en lo que se refiere al incremento del rendimiento de cualquier sistema en el que se utilicen fragmentos de código no paralelizable, es decir, de todos los sistemas que actualmente se conocen. Una vez asumida dicha limitación, no queda más opción que optimizar los programas para que el rendimiento de los mismos en el sistema sea el mejor posible. Esto amenudo implica añadir una fase más en la vida de nuestros programas.

Así pues a las fases de análisis, diseño e implementación hay que añadir una fase de paralelización, en la que se deberá tener en cuenta las necesidades del programa y del problema a resolver, así como de los medios de los que se disponen para poder hacer que el programa aumente el rendimiento a medida que aumenta la capacidad de procesamiento de sistema.

Existen otras muchas limitaciones además de la ley de Amdahl que afectan al desarrollo de aplicaciones paralelas, comenzando por el sistema elegido para paralelizar aplicaciones, hasta la granularidad del paralelismo (que a este nivel siempre será denominado paralelismo de grano grueso). Muchos de estos métodos son brevemente referenciados en el tema de sistemas distribuidos, otros son de una granularidad algo más fina, relativos a compiladores que preparán el código paralelizado para sistemas especiales.

Para ilustrar esto con un caso pueden suponerse dos compiladores diferentes para un mismo lenguaje. Un compilador *inteligente* frente a uno que no lo es tanto ante una misma entrada: un programa con una secuencia de divisiones con coma flotante, después unas sumas del mismo tipo y por último unas operaciones con enteros, todas sin dependencias entre síÒcurre que:

- El compilador *relajado* no las reordena con lo que si el procesador no tiene ordenación de instrucciones hará que las instrucciones de enteros, que seguramente tardan solamente 1 o 2, ciclos tengan que esperar a las instrucciones de división en coma flotante, operaciones que pueden tardar más de 30 ciclos. Mientras se hacen las divisiones en coma flotante no se están usando las unidades de suma de flotantes ni las de operaciones con enteros. Estos componentes del procesador permanecen ociosos mientras otros se usan intensivamente.

- En cambio el compilador *inteligente* primero podría ejecutar las instrucciones que más tiempo tardarán en ejecutarse, y después ejecutaría las instrucciones que usan otras unidades para que el procesador pueda estar procesando de forma paralela todos los datos, aprovechando al máximo su hardware. Por supuesto que esta misma operación (primero las operaciones que más tardan) si se realiza sobre las mismas unidades hace que el retardo medio de finalizar una instrucción aumente por lo que el compilador debe hacer las elecciones correctas si está intentando realizar una ordenación óptima.

### 2.2.4. Granularidad del paralelismo

Al hablar en los apartados anteriores acerca de la granularidad del paralelismo se hacía referencia a la implicación que tiene el paralelismo en el ámbito de la programación. De este modo, se puede decir que la granularidad en el paralelismo de un sistema se encuentra en paralelizar código de procesos, rutinas, módulos o bucles a nivel de instrucción. El término **granularidad se usa como el mínimo componente del sistema que puede ser preparado para ejecutarse de manera paralela**. Por norma general cuanto más fuertemente acoplado (en

el sentido descrito en la sección Arquitecturas) es un sistema, menor es la granularidad del paralelismo en la programación. Dependiendo del grado de granularidad del sistema se diferencia en:

1. Sistemas de granularidad fina.

   *a*) bucles

   *b*) sentencias

   En general se hace a nivel de instrucciones en ensamblador. Generalmente son explotados por sistemas hardware muy caros con los nodos o procesadores fuertemente acoplados. Es la granularidad más pequeña y basa prácticamente todo su funcionamiento en propiedades del hardware. El hardware puede ser suficientemente inteligente para que el programador no tenga que hacer mucho por soportar esta granularidad, por ejemplo el hardware puede aportar reordenamiento de instrucciones.

2. Sistemas de granularidad media.

   *a*) módulos

   *b*) rutinas

   *c*) tareas o procesos

   Dentro de estos sistemas se incluyen varios de los que se describirán más adelante, como pueden ser RPC, openMosix y otros, si bien estos mismos están entre el paralelismo de grano grueso y el paralelismo de grano medio. El paralelismo de grano medio en general es explotado por el programador o el compilador. Dentro de él también se encuentran diversas librerías como pueden ser PVM o MPI. El hardware normalmente también se prepara para poder aprovechar este tipo de paralelismo, por ejemplo, los procesadores pueden disponer de instrucciones especiales para ayudar en el cambio de una tarea a otra que realiza el sistema operativo.

3. Sistemas de granularidad gruesa.

   *a*) trabajos o programas

   *b*) módulos o procesos

   El paralelismo de grano grueso es el que explota el programador mediante programas que no tienen por qué requerir la utilización de ninguna librería externa, sino solamente el uso de conocimientos de programación para paralelizar un algoritmo. Se basa principalmente en cualquier tipo de medio que utilice el programador para crear un programa, que solucione un problema de manera paralela, sin tener por que hacer uso más que de su habilidad de programador y de un buen algoritmo. Son los más limitados al carecer de métodos específicos para comunicación entre nodos o procesadores, se dan en sistemas muy débilmente acoplados.

Una vez vistos los tipos de granularidades existentes podemos catalogar algunos de los sistemas que vamos a tratar a lo largo de este documento. Para empezar cabe señalar que mayormente se tratará del estudio de sistemas distribuidos y clusters, esto implica que el sistema está compuesto por diversas máquinas con un sistema operativo que puede hacer el sistema más acoplado o no. El software de cluster en el que se particulariza es openMosix, como se explicará con mucho más detalle. La granularidad de openMosix está en los procesos, es decir, podemos paralelizar procesos y hacer que estos se ejecuten en nodos distintos. De hecho openMosix se encarga de balancear la carga de los nodos de modo que la carga global del sistema permanezca homogéneamente distribuida. Este tipo de granularidad es del tipo medio, si bien puede ser confundida con la de grano grueso (el límite siempre esta bastante difuso). Otros sistemas como pueden ser RPC, MPI, PVM o clusters Beowulf se acercan más al paralelismo de grano grueso.

### 2.2.5. El problema de la transparencia

Uno de los mayores problemas que existen en la creación de programas que hagan uso de paralelización (quitando los de granularidad fina que ya se ha visto que son explotados a bajo nivel por el compilador y por el hardware) es la transparencia en la programación de dichos programas. A la hora de crear un programa que resuelva un problema mediante el uso de un algoritmo que explote de alguna manera la paralelización hay que

conocer el sistema de ejecución. Dependiendo del sistema elegido y teniendo en cuenta que por norma general se paralelizan tareas, procesos, procedimientos, rutinas o componentes distribuidos que realizan este trabajo, hay dos modelos de implementación:

1. Modelo de programación explícita.

   En el que se requiere una biblioteca de funciones especiales que se encargan tanto de realizar la comunicación como los métodos de migración y demás factores que en un principio[8] no debe afectar al programador, el cual debe abstraerse de esta capa. Este tipo de modelo es el que utiliza RPC, MPI o PVM. Requiere un conocimiento especial de dichas bibliotecas, lo que limita la velocidad de desarrollo del software y además lo hace más costoso debido al tiempo que se debe gastar en el conocimiento de las funciones.

2. Modelo de programación implícita.

   Es un modelo quizá más atractivo. Basa todo su funcionamiento en que el programador sepa lo mínimo del sistema para paralelizar sus procesos. Generalmente este modelo lo explotan los compiladores especiales de un sistema particular.

   Por otro lado se suele depender de macros o funciones especiales que delimitan la granularidad de los procesos a migrar. Un caso especial de este tipo es openMosix: la programación y migración de procesos en openMosix no requiere de conocimiento del usuario respecto al cluster de máquinas. Lo ideal para obtener transparencia en la programación será programar de manera completamente implícita y que al mismo tiempo el sistema implantado fuese lo menos intrusivo posible en lo que se refiere a comportamiento con el usuario final.

Como se puede ver en ambos sistemas y en el estado de arte en el que se encuentran los sistemas distribuidos y clusters, no existe de momento ninguna manera, ya sea mediante programación explícita o implícita, de hacer uso de la transparencia en la migración de procesos, o la ejecución de procesos remotos, el uso de tareas programadas con PVM o MPI, *etc.*

Para que un sistema fuese realmente transparente y al mismo tiempo suficientemente eficiente, en lo que se refiere a programación paralela, el programador debería conocer a fondo la programación del sistema, i.e. las habituales llamadas de creación de memoria, librerías estándar, o basadas en las estándar, llamadas a sistema y demás herramientas habituales de la programación. Por otro lado, el compilador o el sistema se deberían encargar de efectuar todas aquellas tareas que de manera habitual se hacen en los sistemas que actualmente existen, es decir, la comunicación entre procesos, la localización de dichos procesos, rutinas o tareas, la paralelización del código del programa y demás tareas. Así el programador no tendría que preocuparse de los detalles y se podrían hacer desarrollos más rápidos y baratos.

Lamentablemente dicho sistema no existe. La explotación de los sistemas distribuidos de momento requiere un gran conocimiento del sistema en el que se trabaja por parte del programador. De hecho, las optimizaciones de los algoritmos se basan en gran medida no en cómo de bueno es el modelo del algoritmo, sino en las capacidades especiales de cada sistema.

Es decir, cada sistema especial como puede ser Beowulf, PVM u openMosix tienen puntos fuertes y puntos débiles en lo que se refiere a la programación, y es mediante el conocimiento de estas características de la manera que se pueden optimizar programas paralelos. Un ejemplo de esto puede encontrarse en aplicaciones que requieren comunicación entre procesos. El caso de openMosix, como se verá más detalladamente, es el caso de uno de los sistemas que poco a poco ha pretendido ser un sistema lo mas transparente posible y que basa toda su granularidad del paralelismo en el *proceso*. Las aplicaciones diseñadas para ejecutarse en openMosix contienen procesos que se ejecutan de manera independiente y que de una manera u otra se deben comunicar de manera transparente con otros procesos. Estos procesos pueden estar en el nodo local donde se ejecutaron o en otros nodos, lo cual complica la cosa, ya que la relación que une a un proceso con otro puede ser un *pipe* con nombre y sin embargo estar los procesos en dos nodos distintos.

Imagínese el típico caso de la multiplicación de una matriz. Se supone que el caso ideal para openMosix sería utilizar memoria compartida para las matrices y dejar que el sistema haga todo de la manera más efectiva posible. Sin embargo openMosix no implementa en las versiones actuales[9] un manejo eficiente de la memoria compartida y por tanto se deben buscar otros mecanismos de programación para aprovechar al máximo la utilización del sistema.

[8] Según el concepto de diseño de esa biblioteca.

[9] Y no hay espectativas que lo consiga, por la propia funcionalidad que persigue.

Otro posible ejemplo sería el de PVM. Supóngase el caso de determinadas tareas con uso de CPU intensiva que se sitúan en unos nodos concretos antes de empezar su ejecución. Supóngase también que el tiempo de ejecución de dichas tareas es en muchos casos aleatorio, y por tanto, el sistema tendrá muchos nodos altamente cargados y otros poco cargados. Como PVM no posee de la capacidad de balanceo automático de carga que tiene openMosix se estará desaprovechando la capacidad de cómputo del cluster en general y la tarea que se le encomendó tendrá que esperar hasta que el último de los nodos acabe con ella, mientras los demás nodos están ociosos y no pueden ayudar en su finalización. En cualquier caso la transparencia en los sistemas distribuidos y clusters que más se utilizan actualmente requiere de un conocimiento exhaustivo por parte del programador, lo que en muchos casos requiere un coste adicional[10].

Por otro lado está el problema de la intrusión en el código. En este caso se supondrá el caso ideal de haber conseguido un sistema realmente transparente a los ojos del programador, lo que implicaría que la mayoría de las aplicaciones necesitarían de una compilación especial o un compilador especial para el sistema en concreto, lo que hará las aplicaciones normales incompatibles con nuestro sistema, teniendo que compilar todas las aplicaciones de nuevo. En un principio se podría pensar que en el caso de el sistema operativo linux esto no sería problema más que para las distribuciones (aparte de algún programa de más bajo nivel que habría que rehacer).

El problema real está en lo intrusivo que puede llegar a ser un sistema de estas características, cuando los servicios distribuidos que se proveerían solamente serían apreciados por una pequeña parte de los usuarios del sistema operativo, seguramente por esto se crearía más antipatía que simpatía a estos cambios en el kernel.

Uno de estos casos ha sido el de Amnon Barak. Amnon Barak es el líder del proyecto Mosix. En una de las primeras implementaciones trató de hacer un sistema como el que se ha descrito, con el resultado de tener que implementar aproximadamente el 60 % del kernel, más la reimplementación de las librerías necesarias para ejecutar los programas más habituales; esto sin tener en cuenta que todos los programas que eran afectados por dichas librerías debían de ser recompilados para poder ser utilizados. De esta manera se puede hacer notar que por el momento el tener un sistema transparente en lo que se refiere a la programación supone tener un sistema altamente intrusivo.

### 2.2.6. Paralelización de programas

Se ha visto que la ley de Amdahl limita el incremento de rendimiento de los programas cuando éstos se ejecutan en sistemas multiprocesadores. También que la optimización de los programas suele depender en la mayoría de los casos del conocimiento del sistema más que del algoritmo que se utilice. Comprendiendo esto puede empezarse a definir cómo debe ser el software a nivel de aplicación para explotar al máximo el paralelismo de estos sistemas. Por un lado se intentará, basándonos en la ley de Amdahl, que el aumento de rendimiento (o *speedup*) de los programas sea el máximo posible, esto implica que los fragmentos de código no paralelizable deben ser los mínimos posibles. Por otro lado deberán conocerse las limitaciones y características del sistema para el que programaremos.

El primer paso en el desarrollo de un programa paralelizado es, como siempre, plantear el problema mediante técnicas de divide y vencerás. Deberá localizarse lo paralelizable en el problema de manera abstracta antes de pensar en la paralelización del código, es decir que existen problemas inherentemente paralelos, como pueden ser la suma de las componentes de dos vectores según sus posiciones o la renderización de imágenes. Estos problemas se pueden resolver de manera óptima. Por otro lado, están los problemas en los que en un principio no vemos claramente la inherencia del paralelismo como pueden ser la multiplicación de matrices, compresión de la información o compilación.

Teóricamente se puede paralelizar cualquier cosa que se haya diseminado mediante técnicas de divide y vencerás, procesos, módulos rutinas, o algoritmos paralelos completamente.

Existen dos formas bien conocidas y fáciles de comprender de paralelismo:

1. El **paralelismo funcional** divide las aplicaciones en funciones. Se podría ver como paralelismo de código. Por ejemplo puede dividirse en: entrada, preparación del problema, solución del problema, preparación de la salida, salida y mostrar la solución. Esto permite a todos los nodos producir una cadena. Esta aproximación es como la segmentación en funciones de un procesador.

   Aquí se sigue la misma idea pero a nivel de software, se dividen los procesos formando una cadena y dependiendo uno del siguiente. Aunque al principio no se logre el paralelismo, una vez que se ponen todos

[10] Si bien es cierto que los dos ejemplos que hemos mencionado anteriormente representan el escenario ideal para utilizar ambos clusters, openMosix y PVM de manera conjunta, de modo que uno se encarge de las transferencias de las matrices y la memoria compartida y el otro se encargue de el balanceo de la carga de las tareas que PVM lanza.

los nodos a trabajar (i.e. cuando hayamos ejecutado `N` veces lo que estemos ejecutando, siendo `N` el número de pasos en los que hemos dividido la aplicación) se consiguen que todos los nodos estén trabajando a la vez si todas las funciones tardan el mismo tiempo en completarse. Sino, las demás funciones tienen que esperar a que se complete la función más lenta.

La idea es exactamente la misma que trataremos en el tema *Arquitecturas* aunque allí se referencie con el término *pipeline*. Esta forma de paralelizar no es ampliamente usada puesto que es muy difícil dividir en funciones que tarden el mismo tiempo en ejecutarse. Sólo se usa en casos concretos donde se ve claramente que es una buena opción por ser fácilmente implementable.

2. El **paralelismo de datos** se basa en dividir los datos que se tienen que procesar. Típicamente los procesos que están usando esos datos son idénticos entre sí y lo único que hacen es dividir la cantidad de información entre los nodos y procesarla en paralelo. Esta técnica es más usada debido a que es más sencillo realizar el paralelismo.

   Ejemplos pueden ser los films *Final Fantasy* o *El señor de los anillos* que fueron renderizadas en un cluster de renderización con el software Maya y Renderman. Esta forma de renderizar se basa en dividir los frames (cuadros) que se deban renderizar en una escena en concreto o todos los frames de la película entre todos los ordenadores. Para ser más exactos, primero se divide el número de frames por el número de nodos y se envía el número resultante de frames a cada nodo (suponiendo que es un cluster donde todos los nodos tienen la misma capacidad de proceso). Esta es la fase de preparación y suele ser secuencial, puesto que es una pequeña parte de la producción. Una vez hecho esto, cada nodo renderiza los frames que le fueron asignados. Esta fase puede durar semanas. Una vez se tienen todos los frames se unen en un fichero único que tiene toda la película. En este caso en particular se usa paralelismo de datos, y aunque algunas imágenes tardan más en renderizarse que otras y por lo tanto algún nodo tardará menos en acabar su trabajo que otro, lo que se hace es no elegir imágenes de forma secuencial (de la 1 a la `n` al nodo 1) sino paralela (1 al nodo 1, 2 al nodo 2... n+1 al nodo n+1) esto suele compensar la carga entre los nodos por lo que más o menos todos los nodos se mantienen en el máximo rendimiento (si no se comprende esto, piense en lo poco que tardaría el nodo que renderize unos títulos de crédito simples).

Por supuesto muchos problemas se pueden solucionar con cualquiera de los dos paradigmas. En el anterior ejemplo podría haberse hecho una división funcional y hacer que un nodo renderizase luces, otro pusiera texturas, otro reflejos, *etc.* En cambio otras soluciones son específicas o al menos mucho más efectivas con una de las dos aproximaciones. Todos los casos en los que no se tiene una idea clara de lo que pueda tardarse en una función es más paralelizable con paralelismo de datos.

Una vez analizada la fuente de paralelismo de los programas sólo falta comenzar a realizar el estudio en pseudocódigo o cualquier otro sistema parecido. Hay que conocer los límites y características del sistema para el que va a programarse. La migración o ubicación de un proceso, módulo o rutina está ligada a un coste adicional; hemos de conocer los costes de nuestros sistemas y evaluar la conveniencia de paralelizar acciones.

Un programa paralelizado no debe obtener peor tasa de eficiencia que un programa secuencial, lo cual no siempre se cumple ni es posible debido a transferencias de comunicación entre los procesos, módulos, nodos o elementos del cluster. En este apartado juega un papel importante la red de comunicación, el cual ocupa un capítulo entero. La paralelización del código está comprometida a las ventajas de los sistemas implantados, por tanto no podremos hablar en forma general de ésta. En cualquier caso se dará una visión de un sistema concreto openMosix y se hará alguna alusión a sistemas del tipo PVM.

La paralelización del software se basa en principios generales asociados a la manera de programar que utilizamos actualmente tanto por arquitectura de los computadores, funcionamiento de compiladores como lenguajes de programac ión. Uno de estos principios generales es el denominado principio de localidad, del cual existen dos tipos de localidades en nuestros programas:

- Localización temporal.

  Este tipo de localización se basa en la manera en la que se puede encontrar el código de los programas y como se ejecutan estos en el tiempo. Generalmente, los programas están compuestos por un alto porcentaje de estructuras en forma de bucle que ejecutan instrucciones. Lo que implica que con tener en memoria estos bucles, tendremos acceso a la mayor parte del programa. Por otro lado normalmente procedimientos o funciones que están relacionadas suelen estar en zonas de memoria adyacentes, así como las instrucciones que se ejecutan secuencialmente.

- Localización espacial.

  Este tipo de localización se basa en la manera en la que los programadores organizamos nuestros datos en estructuras, objetos, matrices, vectores. Esta manera de organizar elementos de los programas que están muy relacionados entre sí hace que a la hora de solicitar memoria para los mismos éstos sean alojados en segmentos contiguos de memoria. Cuando en un programa se pide memoria, se intenta pedir de una vez toda la memoria que vayan a usar la mayor parte de datos posible, para precisamente tener que estar pidiendo memoria continuamente. Esto produce esta localidad.

Nuestra intención principal debería ser explotar el paralelismo al máximo nivel, para esto pueden utilizarse las típicas formas que se usan para organizar la información y paralelizar al máximo posible, accesos a la misma, operaciones a realizar y demás acciones. Por ejemplo, un programa cuyo núcleo de proceso principal se encuentre en un bucle sería perfectamente paralelizable, pongamos que el bucle se realizara 15000 veces. En el caso de tener 3 nodos podemos distribuir su ejecución en 3 procedimientos que se ejecuten en estos tres nodos y que realicen bucles de 5000 iteraciones de bucle cada uno. Para que esto sea posible es necesario cumplir un objetivo que también debemos tener en cuenta siempre que programemos aplicaciones paralelas en sistemas distribuidos como clusters. Los segmentos de código, rutinas, bucles o cualquier operación sobre estructura s de datos, deben ser ortogonales, en el sentido de que no deben interferir entre sí.

Al paralelizar programas se ve que se obtiene como ventaja un incremento del rendimiento. Este incremento puede ser decremento si estamos comunicando continuamente nuestros procesos a través de una red, que generalmente suele ser el cuello de botella del sistema. Es por esto que los elementos de procesos a paralelizar deben ser lo más independientes y ortogonales posibles.

También hay que pensar que si un proceso tiene que dedicar instrucciones a la comunicación con otros procesos, este código no es paralelizable y además es código que añadimos a cada uno de los procesos, por lo que la escalabilidad disminuye drásticamente con la cantidad de información pasada entre procesos. Aquí es cuando es especialmente importante el desarrollo software pues una buena división en procesos puede evitar esta sobrecarga de comunicación.

En el caso que aquí se trata, programación de clusters o sistemas distribuidos, se llamarán *elementos de proceso* a los módulos, procesos, procedimientos o rutinas que se consolidan como unidades de proceso a ejecutar en los nodos, aunque haya que diferenciarlos claramente del concepto que tiene el término elemento de proceso en sistemas multiprocesad or.

Los elementos de proceso deben ser ortogonales entre sí. El problema que anula generalmente la ortogonalidad de estos elementos, esto es la dependencia de datos. Cuando surgen dependencias, ya sea de datos o de control[11], el programa o sistema deja de ser paralelizable y pasa a ser secuencial, se necesita interconectar generalmente los elementos para obtener sus resultados o para obtener sus estados y poder continuar el programa, esto requiere como ya hemos dicho comunicación a través de la red, lo que puede llegar a ser bastante costoso.

No existe un método general para evitar las dependencias o para ortogonalizar elementos de proceso, es por esta razón por la que todavía puede ser denominado arte en lugar de ciencia a este tipo de investigaciones. En la mayoría de ocasiones se debe utilizar algoritmos que en programación normal se darían por descartados a simple vista. En otras ocasiones el sistema permite utilizar algoritmos óptimos paralelizables. De este modo, aunque el sistema sea el mejor sistema paralelo existente, si la resolución de los problemas que tienen que hacer no son paralelizables estaremos tirando tiempo y dinero. La mayoría del esfuerzo en el desarrollo de programas que resuelvan problemas de manera paralela, se gasta en:

- Estudio completo del problema.

  En el cual se evalúe las maneras de atacar el problema, se dividan las distintas secciones de éste y se tenga en cuenta las implicaciones. Se corresponde con la parte de análisis del programa

- Estudio de la paralelización del problema.

  En la cual se busca la ortogonalización del problema para que este pueda ser fácilmente paralelizable. Se evalúan todas las posibles vías de paralelización y se eligen las que más se adecuen al sistema concreto sobre el que se vaya a ejecutar.

- Estudio del sistema.

[11] Entendiendo por dependencia de control el momento en el que un elemento debe tomar el control de un recurso único no compatible, y por tanto solicitar permiso al resto o por lo menos avisarlos para evitar situaciones de interbloqueo

Tanto a nivel práctico como teórico. Por ejemplo en el caso de PVM basta con conocer el modelo de programación, su sintaxis y el cluster sobre el que se va a ejecutar, la carga que puede aceptar cada nodo, dispositivos y recursos de cada nodo, *etc*.
En el caso de openMosix se debe conocer el estado del cluster, las maneras actuales de poder comunicar procesos en el sistema, creación de entorno adecuado. Cada sistema debe ser explotado de una manera diferente.

- Estudio de la paralelización del código.

  Este apartado depende completamente de los dos anteriores, es decir, sólo se puede hacer correctamente en el caso que se hayan cumplido los dos apartados anteriores, corresponde con la fase de diseño del programa. Es en esta parte del desarrollo en la que muchas veces se deben cambiar algoritmos adecuados por otros que lo son menos para que sean más fácilmente paralelizables en un sistema concreto y se obtenga mejor rendimiento del mismo.

- Pruebas del sistema.

  Muchos programas desarrollados mediante el paradigma de programación paralela hacen uso de un modelo de programación poco ortodoxa, que los hace difíciles de probar y de depurar al mismo tiempo. Por ejemplo, cuando se ejecuta el `gdb` sobre un programa que lanza otros programas y se comunica con estos mismos, la depuración se hace un tanto más difícil. Otra consecuencia de la programación poco ortodoxa es que en muchas ocasiones el programa no se comporta de la manera o con la eficiencia con la que habíamos supuesto contaría en un principio[12].

### 2.2.7. Ejemplos de problemas paralelizables

Generalmente los programas convencionales no cuentan con ninguna manera de paralelizar su código. Es decir, sería genial poder contar con algún ejemplo de programa que comprima de formato DVD a DivX en paralelo y poder comprimir una película en un cluster como openMosix, o poder ejecutar programas de compresión en varios nodos a la vez. La realidad es otra. La mayoría de los programas son pensados y implementados de manera secuencial. Si bien existen muchos campos donde el desarrollo de programas que hagan uso de una manera u otra de paralelismo, está creciendo cada vez más. Actualmente, entre estos campos contamos con:

- granjas de compilación
- granjas de renderización
- procesos matemáticos
  - multiplicación de matrices
  - suma de vectores por componentes
  - multiplicación de matrices o vectores por escalares o funciones escalares.
  - integrales definidas (creando intervalos más pequeños como elementos de proceso)
- compresión
- aplicación a nuevos paradigmas inherentemente paralelos
  - redes neuronales
  - algoritmos genéticos

En el campo científico es en el que más se están haciendo desarrollos de este tipo de programas ya que es el campo en el que generalmente es más fácil encontrarse problemas a paralelizar por la naturaleza de estos. Actualmente otras nuevas vías de desarrollo están siendo bastante utilizadas.

Existen programas de renderizado como MAYA o 3D Studio que utilizan paralelismo en forma de *threads* para poder hacer uso de sistemas multiprocesador, de modo que el tiempo de renderizado se reduzca considerablemente. Es una lástima que ninguno de estos sistemas haga uso de soluciones libres que permitan abaratar

[12] Muchas veces esto suele deberse a falta de conocimiento del sistema sobre el que se ejecuta el programa en cuestión.

el coste del sistema, por ejemplo un MAYA para Linux (que existe) que utilice un cluster Beowulf, PVM, MPI u openMosix permitiendo abaratar gastos a la empresa que lo utiliza y al mismo tiempo, reducir tiempos de renderizado. Para aprovechar varios nodos se usa un programa especial llamado Renderman el cual decide a que nodo enviar las imágenes a ser renderizadas. Este programa es una solución particular que se desarrolló para *Toy Story* (cluster de máquinas SUN) y que sólo funciona para el problema de las granjas de renderización.

Existen también múltiples entornos de trabajo distribuidos en empresas que utilizan de una manera muy superficial paralelización de los problemas, ya que la mayoría están dominados por paradigma cliente-servidor que impera en el mercado.

En el campo científico es ampliamente utilizado. Un ejemplo real es el de estudio oceanográfico realizado por el instituto nacional de oceanografía española[13], en este estudio, se utilizaban matrices descomunales para poder controlar el movimiento de las partículas comprendidas en una región de líquido, que tenía en cuenta viscosidad, temperatura, presión, viento en la capa superficial del liquido, y otras muchas variables. Este ejemplo es el típico en el cual PVM y openMosix podrían ser utilizados conjuntamente.

El ejemplo de código paralelizable más típico y simple es la multiplicación de una matriz.

[13] presentado en un Hispalinux por hacer uso de software libre en su totalidad.

## 2.3. ARQUITECTURAS

*It is true greatness to have in one*
*the frailty of a man and the security of a god.*

*Seneca*

En este tema se tratarán tanto las arquitecturas hardware como software que se han desarrollado para conseguir hacer trabajo paralelo o distribuido. Estas arquitecturas fueron inventadas para superar los problemas de rendimiento y responden a distintos enfoques para conseguir sacar más rendimiento gracias al paralelismo. Algunas arquitcturas pretenden tener una mejor relación velocidad/precio y otras ser las máquinas más rápidas del mercado.

Las soluciones hardware fueron las primeras en aparecer, las soluciones software tuvieron que esperar hasta que el hardware diese la potencia necesaria y en el caso de los sistemas distribuidos se tuvo que esperar a que en los años 70 se desarrollaran las redes de área local. En el capítulo *Sistemas operativos* se explicará con detalle el software distribuido a nivel de sistema operativo.

### 2.3.1. Soluciones hardware

Las soluciones hardware han estado en el mercado de la computación desde sus inicios. Para muchas empresas la única manera de crear mejores máquinas era crear arquitecturas paralelas a partir de las que ya poseían.

Como el número de estos tipos de máquinas es elevado, existen numerosas y diversas soluciones. Quizás la división más conocida y básica sea la taxonomía de Flint. Flint dividió las soluciones hardware en cuatro categorías:

- SISD: un flujo de instrucciones único trabaja sobre un flujo de datos único, a esta categoría pertenecen las CPUs simples y las superescalares.
- SIMD: un flujo de instrucciones único trabaja sobre un flujo de datos múltiple, en esta categoría tenemos los computadores matriciales.
- MISD: un flujo de instrucciones múltiple trabaja sobre un flujo de datos único, resultado teórico de la clasificación, el modelo que más se acerca son los computadores sistólicos.
- MIMD: un flujo de instrucciones múltiple trabaja sobre un flujo de datos múltiple, estos son los multiprocesadores y multicomputadores.

Se irá viendo cómo se explota el paralelismo en el hardware a varios niveles: desde lo más pequeño (procesadores) hasta lo más grande (multicomputadores).

#### SISD

Un procesador se dice superescalar cuando acepta varias instrucciones a la vez, por tanto se ejecutan a la vez. Todo microprocesador destinado a computadores actuales es superescalar, para que un microprocesador sea superescalar necesita tener redundancia de unidades funcionales, esto no se debe confundir con los computadores SIMD. Estas CPUs pueden realmente ejecutar `N` instrucciones a la vez (son superescalar es grado `N`) solamente si no hubiera dependencia entre los datos. Hay tres formas en las que podemos ejecutar las varias instrucciones simultáneas:

- Emisión de instrucciones en orden con terminación en orden: según llegan las instrucciones se ejecutan, hay que parar el proceso en cada dependencia de una instrucción con la anterior. También las instrucciones más rápidas tienen que esperar a que terminen las instrucciones más lentas para mantenerse el orden de partida.
- Emisión en orden con terminación en desorden: las instrucciones pueden acabar en cualquier orden, evitando tener que esperar a las instrucciones lentas.
- Emisión en desorden con terminación en desorden: las instrucciones se emiten y acaban en cualquier orden, este es el método que consigue mayor rendimiento. Se necesita tomar decisiones sobre qué instrucción ejecutar primero de entre un buffer de instrucciones. Esta técnica hace tiempo que la vienen usando los compiladores para conseguir mayores optimizaciones de los programas.

El problema de los ordenadores superescalares está en los llamados *parones de instrucciones*. Los parones de instrucciones se producen por la dependencia que puede existir en instrucciones secuenciales. Estos parones son retardos, vienen dados cuando existe una dependencia entre dos instrucciones que están intentando usar un mismo registro para escribir datos o una instrucción que debe esperar a que otra actualiza un dato para leer de él. Para eliminar los parones[14] podremos recurrir a:

- **Renombramiento de registros**, esta técnica también viene siendo usada por compiladores hace tiempo y varias arquitecturas la soportan, por ejemplo los R10000 tienen 64 registros para renombramiento.
- **Adelantamiento de datos** a nivel de unidades funcionales, hacemos que no tenga que esperarse que el dato se almacene en un lugar determinado para que esté disponible.
- **Predicción de saltos**, desde las las sencillas hasta las de nivel 2 o las de predicción por trazas del Pentium4.

Usando estas técnicas eficientemente, en ausencia de dependencia de datos y otro tipo de parones, es cuando se consigue el auténtico paralelismo en computadores superescalares. Estas técnicas se suelen dar a dos niveles:

- Nivel de procesador.

  Son los procesadores con planificación dinámica. El procesador lee las instrucciones secuencialmente, decide qué instrucciones pueden ser ejecutadas en un mismo ciclo de reloj y las envía a las unidades funcionales correspondientes para ejecutarse. El problema de esta solución es que carga al procesador de trabajo y el circuito electrónico para implementarlo requiere muchos transistores. Por ejemplo el procesador AMD K7 tiene un planificador con 36 entradas para las operaciones de punto flotante, con esas 36 entradas toma las decisiones oportunas para rellenar un registro de 88 entradas que son la cola que espera para que las 3 unidades de ejecución totalmente independientes del K7 las ejecuten. En este ejemplo se ve la complejidad del hardware necesario.

- Nivel de compilador. La máquina VLIW.

  Un procesador VLIW deja al compilador hacer todo el trabajo de decisión. El compilador VLIW agrupa las instrucciones en paquetes de una cierta longitud y los envía al procesador. Las decisiones se toman una sola vez en tiempo de compilación y no cada vez en tiempo de ejecución. El problema es que al tener que enviar paquetes de instrucciones fijos (y por otros trucos) se necesita enviar instrucciones NOPS[15] para rellenar los huecos vacíos. Como ejemplo la arquitectura MAJC de Sun dispone de cuatro unidades de ejecución totalmente independientes que no saben nada de los tipos de datos, esto quiere decir que pueden realizar cualquier tipo de operaciones por lo que cuando se ejecuta un programa de un solo tipo de operación (por ejemplo coma flotante) no se están desaprovechando las unidades que no se encargan de ese tipo (enteros). Eso permitirá a este procesador un máximo uso de su cauce.

Un **procesador** se dice **segmentado** cuando tiene sus funciones divididas en varias subfunciones que se realizan de forma independiente y que pueden ejecutarse simultáneamente[16]. Si todo el procesador actuara como un bloque, solo se podría ejecutar una sola instrucción a la vez que tardase `M` ciclos, `N` instrucciones tardarían `N*M` ciclos. En un ciclo sin parones pueden ejecutarse `N` instrucciones en `M +(N-1)M/k` ciclos siendo `k` el número de etapas. Por tanto cuantas más etapas más trabajo paralelo se realiza y más velocidad se obtiene.

Así por ejemplo un procesador dividido en 4 etapas sin dependencias de datos y empezando por la primera instrucción se ve como en el cuadro 2.2. Si se ejecutaran a la vez tantas instrucciones como subfunciones tuviera podría realizarse una instrucción por ciclo. Como ejemplo, las 3 unidades de punto flotantes que se han visto en un ejemplo anterior del K7 están divididas en 15 etapas.

Seguidamente se muestran unos datos sobre como esta división en tantas etapas permiten a los K7 hacer las operaciones en menos ciclos por instrucción que a los Pentium4, que están divididos en menos etapas (ver cuadro 2.3). Para los amantes de las mejores arquitecturas aquí se da una buena razón para inclinar la balanza hacia los de Advanced Micro Devices. La diferencia entre ciclos por instrucción sería mucho mayor al incluir en el cuadro una arquitectura PPC de Apple: éstas llegan a ser competitivas aún funcionando a 800MHz en unos tiempos donde los P4 llegan ya los 3GHz.

[14] Interesante objectivo puesto que evitan la posibilidad de paralelismo, es decir, se tiene que esperar a otras instrucciones. Si se dieran contínuamente el ordenador no tendría ninguna mejora por el hecho de ser superescalar.

[15] Instrucción del lenguaje ensamblador para dejar un ciclo de instrucción sin realizar nada.

[16] Dentro del procesador, el hardware utilizado para cada etapa es diferente.

| | Tiempo 1 | Tiempo 2 | Tiempo 3 | Tiempo 4 | Tiempo 5 | Tiempo 6 |
|---|---|---|---|---|---|---|
| Subfunción 1 | Instr 1 | Instr 2 | Instr 3 | Instr 4 | Instr 5 | Instr 6 |
| Subfunción 2 | | Instr 1 | Instr 2 | Instr 3 | Instr 4 | Instr 5 |
| Subfunción 3 | | | Instr 1 | Instr 2 | Instr 3 | Instr 4 |
| Subfunción 4 | | | | Instr 1 | Instr 2 | Instr 3 |
| Resultados | | | | | Result 1 | Result 2 |

Cuadro 2.2: Arquitecturas. Procesador dividido en 4 etapas y sin dependencias

| Instrucción | ciclos en P4 | ciclos en K7 |
|---|---|---|
| FADD | 1 | 1 |
| FMUL | 2 | 1 |
| FDIV float | 23 | 13 |
| FDIV double | 38 | 17 |
| FSQRT float | 23 | 16 |
| FSQRT double | 38 | 24 |

Cuadro 2.3: Arquitecturas. Inferioridad del procesador P4 frente a K7

La solución mixta de los dos anteriores es un **procesador segmentado superescalar**. En este procesador entran varias instrucciones a la vez que se ejecutan en cada una de las subfunciones, así se están ejecutando a la vez `M*N` instrucciones siendo `M` y `N` los grados de segmentación y escalabilidad del procesador. Si en el cuadro 2.2 se añadiese la capacidad de ser superescalar en cada casilla cabrían `N` instrucciones, así donde antes había `instrucción 1` ahora habría `instrucción 1 ... instrucción N`. Asimismo en `instrucción 2` habría `instrucción N+1 ... instrucción 2N`, *etc.*

### SIMD

Los computadores SIMD son bastante menos frecuentes que los SISD eso se debe a que su programación es bastante más compleja. Un computador SIMD está compuesto de una serie de unidades de ejecución, unos módulos de memoria y una red de interconexión. Gracias a las conexiones cualquier elemento de proceso puede acceder a cualquier módulo de memoria. El paralelismo se consigue ejecutando la misma instrucción sobre varios datos a la vez (cada elemento de proceso un dato). Aquí es donde se complica la programación pues antes de ejecutar la instrucción debe especificar dónde se encuentra los datos en memoria y la configuración de la red de conexión en cada instante.

Aunque se han intentado construir máquinas SIMD puras no han tenido demasiado éxito, no eran suficientemente flexibles y poco eficientes para algunas operaciones (por ejemplo operaciones condicionales). Hoy en día lo que se desarrollan son extensiones vectoriales de procesadores SISD, como ejemplos tenemos AltiVec de Motorola, MMX de Intel o 3DNow de AMD.

Motorola ha apostado por añadir más hardware: 32 registros de 128 bits para operaciones vectoriales. Dos unidades de ejecución independientes (una ALU vectorial y otra para permutaciones), puede mantener una operación por ciclo siempre que encuentre los datos en la cache de nivel 1.

El MMX de Intel en cambio usa registros de 64 bits que ya existían (registros del coprocesador para punto flotante) y añadió 57 instrucciones. Con SSE anteriormente conocida como MMX2 se añadieron 8 registros nuevos de 128 bits y una unidad independiente de suma vectorial. También hay una unidad de coma flotante vectorial pero comparte parte de su *pipeline* con la unidad de ejecución de coma flotante corriente. Cuando se quiere hacer una operación de 128 bits, se dividen en dos y se envían a las unidades de suma y multiplicación por lo que no siempre se obtiene una instrucción por ciclo, pero permite a Intel mantener sus buses internos del procesador a 64 bits.

3DNow implementa MMX, SSE y SSE2 de la misma manera que Intel y añade sus propias instrucciones para las que no pone registros nuevos (pero tiene registros ocultos al programador, para hacer renombramiento), no permite realizar operaciones de 128 bits pero a cambio las unidades de coma flotante son totalmente independientes.

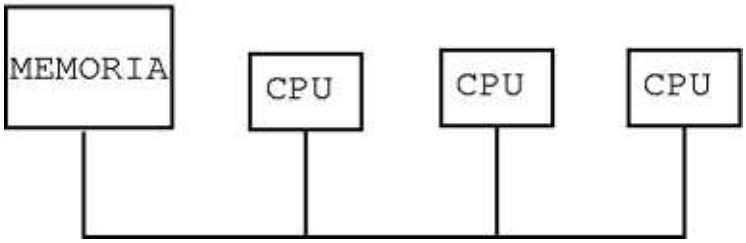



Figura 2.2: Arquitecturas. Multiprocesadores en bus

El problema de estas nuevas instrucciones es el mismo que tuvieron las máquinas SIMD: són más difíciles de programar. Hoy en día se usan mucho los lenguajes de alto nivel por lo tanto se deja la optimización y vectorización de los datos a los compiladores.

Como curiosidad, notar que el compilador de Intel, que es capaz de vectorizar aproximadamente la mitad de los bucles de un programa y que usa SSE2 para tantas operaciones en punto flotante como puede, aumenta los rendimientos por encima de los del procesador AMD K7 en instrucciones como FADD, FMUL y FSQRT. Tras ver el cuadro 2 se comprueba como realmente el AMD K7 es más potente en lo que se refiere a operaciones en coma flotante, no obstante por el momento no dispone de compilador específico en el mercado, esto lo pone en posición desfavorable frente al P4 dentro del uso del paralismo.

### MIMD

Pueden distinguirse multiprocesadores y multicomputadores.

Un sistema multiprocesador consta de muchos procesadores independientes unidos por algún mecanismo entre ellos y la memoria. La primera consideración que se hace al desarrollar un sistema multiprocesador es cómo conectar los procesadores entre sí y éstos a la memoria. Los procesadores deben estar conectados o al menos tener una zona de memoria común, pues están haciendo un trabajo cooperativo.

Según el tipo de acceso a memoria de todos los procesadores de la máquina MIMD se caracterizan en dos tipos, que son: UMA (acceso uniforme a memoria) y NUMA (accesso no uniforme a memoria).

- UMA significa que todos los procesadores acceden a la misma memoria a la misma velocidad, por ejemplo un sistema SMP es UMA, Por ejemplo los compatibles con x86 tienen una arquitectura con la memoria compartida, global para todos los procesadores, con un solo bus para acceder a memoria que comparten todos los procesadores.

- NUMA significa que no los procesadores no pueden acceder a toda la memoria con la misma velocidad. Habrá zonas de memoria a la que se accede más rápido y zonas a las que se accede a menos velocidad. Los procesadores tienen conexión directa con una memoria local, pero también tienen una conexión más lenta a memorias locales de otros procesadores (la memoria local no es memoria cache).

Los sistema UMA a su vez pueden ser de dos tipos dependiendo de la red de interconexión. Así en los **multiprocesadores en bus** (los procesadores comparten buses de dirección, datos y control) existe un árbitro en este bus que es quien decide quien puede en cada momento acceder a los buses. El problema de esta arquitectura es que los buses se saturan con facilidad, a partir de 64 CPUs el bus es el cuello de botella y aumentar el número de CPUs no aumenta el rendimiento.

Los **multiprocesadores con conmutador** són la solución para los problemas que conlleva el bus y, evidentemente, es una arquitectura más cara. La memoria se divide en módulos, varios procesadores pueden acceder a memoria de forma simultánea. Las CPUs y las memorias están conectadas a través de puntos de cruce. Si dos procesadores quieren acceder simultáneamente a la misma memoria, tendrán que esperar, en cualquier otro caso el acceso se hace de forma simultánea. El problema que conlleva esta arquitectura es que el número de conmutadores es alto `NxM` (si `N` es el número de procesadores y `M` el número de módulos de memoria) además estos conmutadores son caros pues tienen que ser de alta tecnología para que no se haga más lento el acceso a la memoria.

Para solucionar estos problemas se creó la red Omega que tiene un número menor de conmutadores. Pero en esta red, y por tener menos elementos de conmutación, ocurren más esperas que en una red de barras como

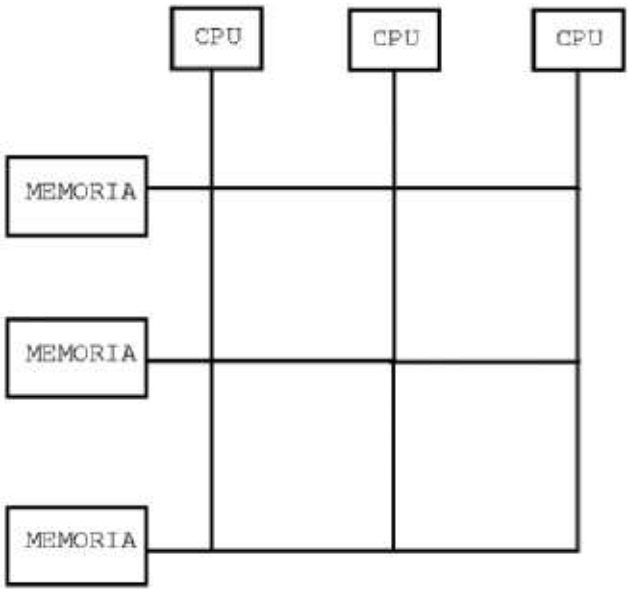



Figura 2.3: Arquitecturas. Multiprocesadores en conmutador

la anterior, aunque menos que en una estructura en bus. Para solucionar todos estos es para lo que se creó la estructura NUMA, pero esta estructura necesita algoritmos complejos.

Un problema que estos sistemas MIMD tienen que solventar es la coherencia de las memorias cache. Las memorias cache de cada procesador guardan los datos que este está accediendo más frecuentemente. En la ejecución normal de un procesador se leen los datos de memoria y se dejan en memoria cache, las próximas lecturas o escrituras se realizan en ella ahorrando tener que ir a memoria principal. El problema ocurre con las escritura en estos datos. Pongamos un ejemplo:

1. la CPU 1 lee de memoria la variable `varGlobal`.
2. la CPU 2 lee de memoria la misma variable.
3. la CPU 1 y la CPU 2 escriben en su copia local de cache un valor.
4. Cuando esos valores se actualizen en memoria (depende de si la cache es de postescritura o escritura directa) estaremos ante una *condición de carrera*. Además el valor escrito por un procesador no se ha propagado a los demás procesadores que están trabajando con un valor antiguo.

Se podría pensar que la solución sería **no permitir memoria compartida** o tener un mecanismo para controlar la situación en ese tipo de memoria, no obstante hay que tener en cuenta que también puede ocurrir el caso de que un proceso corriendo en el procesador 1 lea las variables propias (que pueden ser privadas), entonces se mueve al procesador 2 donde escribe en esas variables. Cuando vuelve a migrar al procesador 1 las variables que el proceso piensa que escribió (la migración es transparente al proceso) no están escritas.

Para evitar estas situaciones se han desarrollado una serie de técnicas. La primera de ellas es dividir la memoria principal y dejar parte de esa memoria como memoria local. Sólo se pasarían a cache los datos de esa memoria local y esto plantea bastantes problemas (copiar datos de la memoria global a la local, perdemos eficiencia en accesos a memoria compartida) y seguramente por esta razón no es una solución amplamente usada.

Tener la **cache compartida** por los procesadores implicaría una velocidad de bus muy elevada, no se implementa en la práctica por sus costes económicos.

Existen las **cache privadas** con directorio compartido donde se guarda el estado de cada uno de los datos, gracias a esta información se implementa un algoritmo que mantiene la cache coherentes. Se implementa en sistemas con multiprocesadores con redes de conexión (no en bus). Existen varios protocolos que se diferencian en la información que guardan referida a los bloques de memoria y como se mantiene esa información.

- Directorios completos.

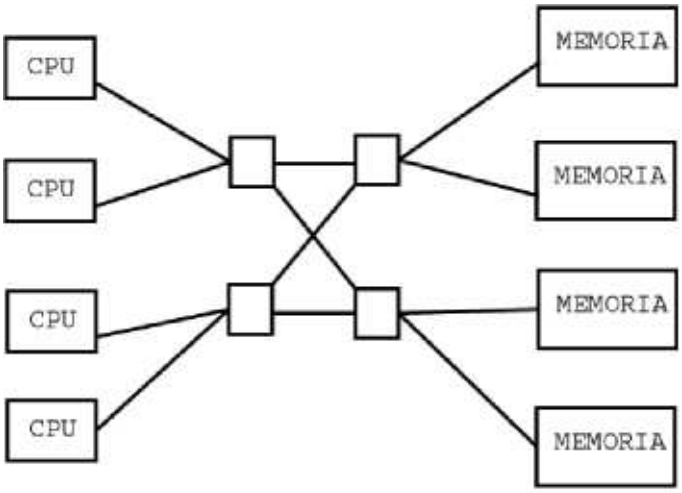



Figura 2.4: Arquitecturas. Red Omega

El directorio tiene un bit de procesador que indica si el bloque está o no en la cache del procesador y un bit de modificado si ha sido modificado, si ha sido modificado, sólo está en ese procesador y sólo él tiene permiso para modificarlo. También hay 2 bits por cada bloque de la memoria cache, uno indica si ese bloque es válido o no y el otro indica si se puede escribir en él.

- Directorios limitados.

  Cuando el número de procesadores es muy alto, la solución anterior requiere mucha memoria, para disminuir esta necesidad se usan directorios limitados que siguiendo el mismo esquema, imponen un límite al número de procesadores. Si se solicita el bloque en más cache de las que entradas tiene el directorio se aplica un algoritmo de reemplazo. El directorio debe especificar el procesador porque ahora no sabemos como antes que bits indican que procesadores.

- Directorios encadenados.

  El directorio sólo indica el primer procesador que tiene el bloque, este procesador indica el siguiente creando una cadena, hasta encontrar el procesador que tiene el bit que indica final de cadena activado.

Entre las soluciones más implantadas en multiprocesadores con sistema de memoria basado en bus se encuentran los protocolos de *snoopy*: los procesadores se mantienen a la escucha de lo que ocurre en los distintos bloques de cache. Ya que el problema ocurre cuando se escribe un dato en la cache, cuando se escribe uno hay dos posibilidades:

- Invalidar el dato de todas las demás cache cuando uno de los procesadores escribe en su cache ese mismo dato. El problema de esta solución es que puede ocurrir que un dato sea muy utilizado y se tenga que estar cogiendo de memoria constantemente pues fue invalidado.

- Actualizar los datos de los demás procesadores cuando modificamos un bloque, el problema de esta aproximación es que se puede generar mucho tráfico y el bus tiene que ser rápido.

A nivel de máquina se han desarrollado otros sistemas para conseguir paralelizar todo tipo de tareas a todos los niveles. El primero de ellos son las IRQs lo que permiten al procesador estar ejecutando programas a la vez que los periféricos están ejecutando sus funciones. Este es un paralelismo a nivel de máquina que permite que se lleve la cuenta del reloj a la vez que se acepta una pulsación de una tecla a la vez que el procesador está haciendo algún proceso pesado.

Tras las IRQs se implantó el DMA, esto es un chip especial dentro de la placa del ordenador que permite ser programado para que un periférico haga accesos a un bloque de memoria sin intervención de la CPU, esto permite mayor autonomía de los periféricos que ejecutan más funciones sin interrumpir a la CPU, con un bus compartido no podrán acceder más de un periférico a la vez a memoria.

Los sistemas multiprocesadores actuales están consiguiendo mejores políticas de gestión por parte del sistema operativo.

Para poner un ejemplo de hacia donde se dirigen las nuevas capacidades de los procesadores para trabajar en paralelo se presenta el caso del procesador MAJC de Sun.

MAJC es capaz de, cuando está en una configuración multiprocesador y detecta que en el *thread* actual se ha llegado a un bucle del que se tardará en salir, producir de manera especulativa (al igual que se realizan instrucciones) un nuevo *thread* que se ejecuta en otro procesador ejecutando las posibles instrucciones futuras en tiempo presente, a esto lo llaman STC (computación espacio temporal).

MAJC implementa también *vertical multithreading*. Así cuando un *thread* tiene un fallo de cache y tiene que ir a memoria a por el dato se hace un cambio de contexto y se ejecuta otro *thread* hasta que el dato ha llegado. Esto es posible porque mucho espacio del procesador MAJC se gasta en registros y es capaz de mantener la información de estado necesaria para hacer el cambio de contexto entre dos *threads* en ellos, con lo que el cambio de contexto es el más rápido que puede implementarse actualmente y da tiempo a ejecutar instrucciones antes de que el dato de memoria haya llegado.

**Multicomputadores:**

Ya se vió en el tema de introducción cómo aparecieron y evolucionaron estos sistemas. Es una arquitectura MIMD donde cada uno de los procesadores puede tener a su vez varios procesadores. En este apartado lo que antes era un procesador ahora será un ordenador entero. Cada uno tiene acceso directo a su memoria local. Entonces es la misma arquitectura que en multiprocesadores pero escalada. También aquí se puede tener una arquitectura en bus u otro tipo, podrán aplicarse exactamente las mismas explicaciones sobre arquitecturas de las que hablamos en multiprocesadores, solo se varía el esquema levemente:

- Bus: por ejemplo una red de ordenadores conectados a través de una LAN se adapta a este esquema. Las velocidades de transmisión son menores que en un sistema multiprocesador en base a buses y las latencias son mayores. Básicamente igual que su homólogo en multiprocesadores, excepto que esta arquitectura es NUMA (pues un ordenador accede a su memoria local mucho más rápidamente que a la memoria de otro ordenador), se usan algoritmos complejos para las operaciones con memoria.

- Conmutadores: ahora los conmutadores son los propios nodos, pues tienen la capacidad de conectarse a sus nodos vecinos (tienen tarjetas de red) se pueden dar en dos arquitecturas diferentes:

  1. Transputers: los ordenadores forman una retícula 2D. Cada nodo solo tiene que conectar con (como mucho) 4 nodos más en cualquier caso, esto es perfecto para escalar el cluster sin tener que añadir tarjetas de red a los nodos ya existentes. Nótese que no hay simetría pues los nodos en los bordes se conectan con menos nodos.
  2. Hipercubos: los ordenadores forman una retícula 3D. Es difícil de visualizar a partir del orden 4. Cada nodo se conecta a N nodos siendo N el grado del hipercubo. Es una configuración simétrica.

### 2.3.2. Soluciones software

Una vez que el hardware obtuvo la potencia suficiente se empezó a explotar el paralelismo a nivel de software, en el capítulo sobre sistemas operativos se verá cómo se comenzó dividiendo el tiempo de CPU entre varios procesos, esto permitía una mayor eficiencia e interacción entre usuarios y máquinas. Con la llegada de las IRQ el sistema operativo tuvo que gestionarlas, teniendo así la responsabilidad de mantener todos los periféricos trabajando a la vez el máximo tiempo posible.

Los sistemas operativos tienen la responsabilidad de utilizar eficientemente los mecanismos que le da el hardware para hacer trabajos paralelamente. Así por ejemplo en multiprocesadores es el sistema operativo quien tiene que distribuir los procesos por los procesadores de una forma que todos los procesadores estén ocupados. El kernel tiene que tener en cuenta que un proceso se ejecuta más eficientemente en un procesador donde se ejecutó recientemente, pues el procesador tiene en su cache datos de ese procesador, también tiene que tener en cuenta si todos los procesadores tienen acceso a entrada/salida o no. Por ejemplo Windows NT ha elegido una forma muy sencilla que es usar `N-1` procesadores para los `N-1` procesos más prioritarios y el último para el resto, el rendimiento de este esquema es más que discutible, pero desde luego es simple.

La evolución de las soluciones software ha seguido la misma que las soluciones hardware: hasta que el hardware no las soportaba las soluciones software no se hacían realmente eficientes. Así tradicionalmente todos

los recursos de una máquina los ha gestionado el núcleo y éste se ha ido adaptando a las nuevas posibilidades, aunque siempre ha habido hardware de un nivel más alto que controlaba su propio funcionamiento, como grandes sistemas gestores de bases de datos.

Con la llegada de los multicomputadores, se plantearon nuevos retos, pero esta vez en vez de ser superados por los diseñadores de sistemas operativos, se vió como un problema de las aplicaciones. Quizás esto fue así porque cuando aparecieron las primeras redes no eran algo barato ni tan común, por eso se consideraba tan puntero el sistema operativo Unix al soportarlas. Tampoco se disponía de hardware barato y potente. Por lo tanto aunque el núcleo daba funcionalidades básicas (creación de procesos, comunicación, manejo de red) todo el problema se dejaba al espacio de usuario. Afortunadamente esta aproximación está cambiando y ya existen varios sistemas de ficheros enfocados a tener un sistema de ficheros único en todo el cluster, pero aún queda mucho por hacer.

Hoy en día una red es muy barata y están muy extendidas por lo tanto las aplicaciones que funcionan sobre ellas se han extendido en todas las áreas del software. Al principio estas aplicaciones eran muy básicas, eran poco más que los protocolos que implementaban (que para la época implementar el protocolo era un logro suficiente), poco a poco se fueron haciendo más complejas. Por aquel entonces Sun hacía mucho desarrollo e implementó algo que hasta la fecha está siendo muy utilizado: RPC. Este sistema permite una comunicación de dos máquinas, una de las máquinas ejecuta unos procedimientos de la otra máquina.

## 2.4. SISTEMAS DISTRIBUIDOS

*The best way to accelerate a Macintosh is at 9.8m sec sec.*
*Marcus Dolengo*

### 2.4.1. Concepto de sistema distribuido y sistema operativo distribuido

Un sistema distribuido es un conjunto de ordenadores o procesadores independientes que cara al usuario funcionan como uno solo. Está formado por varios componentes, relativamente pequeños e independientes, que cooperan estrechamente para dar un servicio único.

Un sistema operativo distribuido es el encargado de cumplir lo anterior. Hay un particionado y cooperación entre todos los componentes, ninguno sobrevive solo, el propio kernel está distribuido por las máquinas. El hardware no da ninguna facilidad, es el software el que determina que un sistema sea distribuido o no. El usuario no sabe dónde se están ejecutando los procesos ni dónde están los recursos. Llevar a la práctica la idea de un kernel global distribuido es muy difícil, pues podría haber inconsistencias de datos en el kernel, además se necesita al menos el kernel mínimo para enviar y recibir información y un mecanismo robusto de comunicación y eficiente para evitar latencias demasiado elevadas.

Lo que se han desarrollado hasta ahora son los llamados **sistemas operativos en red.** En estos sistemas cada máquina tiene su propio kernel. Los sistemas están conectados por una red y se puede hacer *remote login* en ellos, en todo momento el usuario sabe donde se encuentran todos los recursos (ficheros, procesos, *etc.*). No es un sistema distribuido por lo que no tiene las ventajas que se verán en el siguiente apartado.

Actualmente se está caminando desde los sistemas operativos en red a los sistemas distribuidos, aunque aún no se han cumplido los objetivos de un sistema distribuido completamente tenemos ya algunos avances. Por ejemplo ya hay grandes avances en sistemas de ficheros para conseguir que exista un solo directorio raíz al igual que la existencia de una ubicación automática por parte del sistema de los ficheros. Se puede implementar un balanceo de la capacidad y redundancia en los datos para minimizar el impacto de posibles caídas de nodos.

Un cluster openMosix ya implementa una migración de procesos, que a ojos del usuario es transparente. El cluster se usa como un gran computador donde se ejecutan los procesos en todos sus nodos. La ubicación de los procesos la elige el sistema operativo o el usuario, en un intento de balancear la carga.

### 2.4.2. Necesidad de sistemas distribuidos

En un sistema operativo distribuido se cumplen todas los criterios de transparencia, con todas las ventajas que esto supone. Aparte también se deben tener en consideración las siguientes características:

1. Economía: la relación precio-rendimiento es mayor que en los sistemas centralizados sobretodo cuando lo que se busca son altas prestaciones.

2. Velocidad: llega un momento en el que no se puede encontrar un sistema centralizado suficientemente potente, con los sistemas distribuidos siempre se podrá encontrar un sistema más potente uniendo más nodos. Se han hecho comparativas y los sistemas distribuidos especializados en cómputo han ganado a los mayores mainframes.

3. Distribución de máquinas: podemos tener unas máquinas inherentemente distribuidas por el tipo de trabajo que realizan.

4. Alta disponibilidad: cuando una máquina falla no tiene que caer todo el sistema sino que este se recupera de las caídas y sigue funcionando con quizás algo menos de velocidad.

5. Escalabilidad: puede empezarse un cluster con unas pocas máquinas y según la carga aumenta, añadir más nodos. No hace falta tirar las máquinas antiguas ni inversiones iniciales elevadas para tener máquinas suficientemente potentes.

6. Comunicación: los ordenadores necesariamente estarán comunicados, para el correcto y eficaz funcionamiento del cluster se crean unas nuevas funcionalidades avanzadas de comunicación. Estas nuevas primitivas de comunicación pueden ser usadas por los programas y por los usuarios para mejorar sus comunicaciones con otras máquinas.

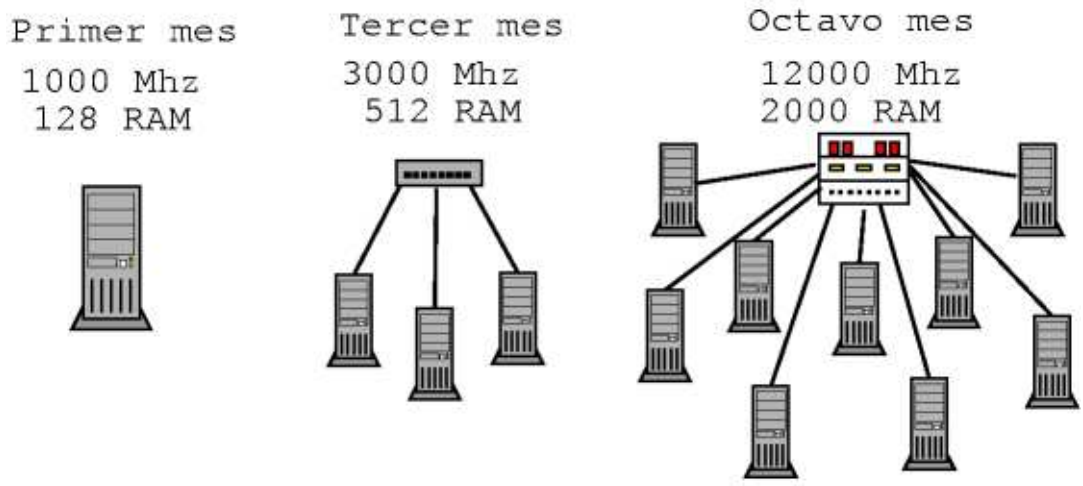



Figura 2.5: Sistemas distribuidos. Escalabilidad de servicios en una empresa

7. Sistema de ficheros con raíz única: este sistema de ficheros hace que la administración sea más sencilla (no hay que administrar varios discos independientemente) y deja a cargo del sistema varias de las tareas.

8. Capacidad de comunicación de procesos y de intercambio de datos universal: permite enviar señales a cualquier proceso del cluster, asimismo permite trabajar conjuntamente con cualquier proceso e intercambiar datos. Por lo tanto será posible tener todos los procesos trabajando en un mismo trabajo.

### 2.4.3. Desventajas: el problema de la transparencia

La principal desventaja de estos sistemas es la complejidad que implica su creación. Básicamente se tienen todos los problemas que se puedan tener en un nodo particular pero escalados. Vamos a ver los problemas que ocurren al intentar implantar las ventajas que hemos visto en el apartado anterior. Los puntos 1, 2 y 3 no tienen problemas de implantación porque son inherentes a los sistemas distribuidos.

- 4.- Alta disponibilidad: podemos conseguir alta disponibilidad pues al tener varios nodos independientes, hay muchas menos posibilidades de que caigan todos a la vez. Pero esto por sí sólo no nos da alta disponibilidad. Tenemos que implantar los mecanismos necesarios para que cuando una máquina caiga, se sigan dando todos los servicios.

  Normalmente se apuesta por la replicación de información. Si tenemos 3 servidores de ficheros, sirviendo los mismos ficheros, si uno de ellos cae podemos seguir obteniendo el servicio de alguno de los otros servidores (en el peor de los casis el servicio quizás se ralentice).

  Este caso además ilustra otro problema que es la necesidad de actualizar todas las réplicas de un servicio: si un nodo escribe en uno de los servidores éste debe mandar los nuevos datos a los otros servidores para mantenerlos todos coherentes. De otro modo al caer uno de estos servidores perderíamos toda la información que hubiéramos ido grabando en este servidor y no en el resto.

  También se tiene que disponer de los mecanismos adecuados para que el nodo que ve el fallo del servidor busque los servidores alternativos en busca de la información que necesita. Además también se debe disponer de los mecanismos necesarios para que los nodos que han caído, cuando vuelvan a conectarse al cluster puedan continuar con su trabajo normalmente.

- 5.- Escalabilidad: el problema es que más nodos suele implicar más comunicación, por lo que tenemos que diseñar un sistema lo más escalable posible. Por ejemplo elegir una comunicación *todos con todos* aunque tenga ciertas ventajas no es una solución en absoluto escalable pues cada nuevo nodo tiene que comunicarse con todos los demás, lo que hace que incluir un nodo no sea lineal.

  Una solución para este problema son los clusters jerárquicos propuestos por Stephen Tweedie: clusters divididos en niveles. Un cluster como lo conocemos es un cluster de primer nivel, los nodos forman una

lista de miembros. Hay un nodo privilegiado llamado líder. Para hacer clústeres más grandes se juntan todos los líderes en un metacluster. Se crean entonces dos listas, una de ellas contiene todos los nodos llamada *subcluster membership* y la otra contiene únicamente los líderes llamada *metacluster membership*. Si se cambia de líder se cambia también en el metacluster. Así podemos agrupar líderes de metaclusters en un cluster de tercer nivel, *etc.*

Esta disposición evita transiciones de todo el cluster, pues si hace falta añadir un nodo y crear una transición en el cluster sólo involucramos a los nodos de esa rama. Cuando el cluster necesite recuperación habrá nodos que participen y otros que no.

- 6.- Comunicación: un cluster tiene más necesidades de comunicación que los sistemas normales por lo tanto tenemos que crear nuevos métodos de comunicación lo más eficientes posibles. Un ejemplo de esto es Heartbeat.

- 7.-Sistemas de ficheros con raíz única: tenemos que independizar los sistemas de ficheros distintos de cada uno de los nodos para crear un sistema de ficheros general. Tenemos una fácil analogía con LVM, la cual abstrae al usuario de las particiones del disco duro y le hace ver un único volumen lógico.

  Entre los problemas que nos encontramos en los sistemas de sabor UNIX se encuentran por ejemplo cómo manejar los directorios */proc* y */dev*. Hacer un solo directorio con ellos significaría en */proc* tener PIDs independientes para cada nodo, algo que no es tan complicado de conseguir y que tiene beneficios como que cada nodo sabe a ciencia cierta y sin ningún retardo cual es el siguiente PID que tiene que asignar.

  Pero además incluye otra información como */proc/cpuinfo*, */proc/meminfo*, *etc.* que es específica del nodo y que o se modifican esos ficheros para poner la información de todos los nodos en ellos (lo cual rompería la compatibilidad con algunas aplicaciones de usuario) o poner la información en directorios distintos que tuvieran como nombre el número del nodo (lo que traería la duda de qué información colocar en */proc*). Aún peor es el caso de */dev* pues aquí están los dispositivos de cada nodo. Cuando los dispositivos de un nodo son totalmente distintos a los de otro no hay ningún problema (incluso poder acceder al dispositivo de otro nodo por este fichero sería perfecto) pero si todos los nodos tuvieran por ejemplo un disco duro IDE conectado en el primer slot como maestro con 3 particiones, tendríamos repetidos en cada nodo `hda1`, `hda2` y `hda3`; por lo tanto tendremos que tener un nuevo esquema para nombrar los dispositivos.

- 8.-Capacidad de comunicación de procesos y de intercambio de datos universal: para conseguir este objetivo necesitamos una forma de distinguir unívocamente cada proceso del cluster. La forma más sencilla es dando a cada proceso un PID único, que llamaremos CPID (cluster process ID). Este CPID podría estar formado por el número de nodo y el número de proceso dentro de ese nodo. Una vez podemos direccionar con que proceso queremos comunicarnos, para enviar señales necesitaremos un sencillo mecanismo de comunicación y seguramente el mismo sistema operativo en el otro extremo que entienda las señales. Para compartir datos, podemos enviarlos por la red o podemos crear memoria compartida a lo largo del cluster. Sobre como crear esta memoria compartida hablaremos en el capítulo de sistemas operativos.

Como se expone en el anterior apartado otras de las ventajas de los sistemas distribuidos es que cumple con todos los criterios de transparencia. Pero conseguir estos criterios implica ciertos mecanismos que debemos implementar:

1. **Transparencia de acceso.**

   Implica tener que mantener el viejo sistema para el nuevo cluster, por ejemplo mantener un árbol de directorios usual para manejar todos los dispositivos de almacenamiento del cluster. No tenemos que romper las APIs para introducir las nuevas funcionalidades.

2. **Transparencia de localización.**

   A nivel más bajo tenemos que implantar una forma de conocer donde se encuentran los recursos, tradicionalmente se han usado servidores centralizados que lo sabían todo, ahora ya se va consiguie ndo que esta información se distribuya por la red.

3. **Transparencia de concurrencia.**

   Esto que no es excesivamente complicado en un sistema local, se ha convertido en un quebradero de cabeza en un sistema distribuido. Se han desarrollado varios métodos para conseguirlo. El mayor problema es la

desincronización de los relojes pues es muy complejo que todos los relojes hardware lleven exactamente la misma temporización por tanto algunos ordenadores ven los acontecimientos como futuros o pasados respecto a otros ordenadores. Este tema se tratará con más profundidad en el capítulo de sistemas operativos.

4. **Transparencia de replicación.**

   Básicamente el problema es que el sistema sepa que esas réplicas están ahíy mantenerlas coherentes y sincronizadas. También tiene que activar los mecanismos necesarios cuando ocurra un error en un nodo.

5. **Transparencia de fallos.**

   Aunque haya fallos el sistema seguirá funcionando. Las aplicaciones y los usuarios no sabrán nada de estos fallos o intentarán ser mínimamente afectados, como mucho, el sistema funcionará más lentamente. Este punto está muy relacionado con la transparencia de replicación.

6. **Transparencia de migración.**

   Tenemos que solucionar problemas sobre las decisione s que tomamos para migrar un proceso, hay que tener en cuenta las políticas de migración, ubicación, *etc.* Además tenemos que ver otros aspectos prácticos como si al nodo que vamos encontraremos los recursos que necesitamos, *etc.* La aplicación no tiene que saber que fue migrada.

7. **Transparencia para los usuarios.**

   Implica una buena capa de software que de una apariencia similar a capas inferiores distintas.

8. **Transparencia para programas.**

   La más compleja. Implica que los programas no tienen porque usar llamadas al sistema nuevas para tener ventaja del cluster. Mosix hace esto muy inteligentemente tomando la natural división en procesos de los programas para migrarlos de forma transparentemente.

### 2.4.4. La tendencia a lo distribuido

Existen varios métodos para intentar distribuir a nivel de aplicación, son métodos que abstraen las capas inferiores y hacen la vida más fácil a los programadores de las aplicaciones, que no se tendrán que preocupar sobre las peculiaridades de las capas inferiores consiguiéndose una mayor efectividad en la programación. Las tecnologías que se verán, en este orden, son:

RPC: Remote Procedure Calls.
RMI: Remote Method Invocation.
CORBA: Estándar de comunicación de objetos.
Bonobo: Abstracción sobre CORBA de GNOME.
KDE: Desktop Enviroment. Veremos: KIO, XMLRPC, DCOP.
SOAP: Simple Object Access Protocol.

**RPC.-**

El concepto de RPC o llamada a procedimiento remoto ha sido explotado desde hace ya varias décadas, de hecho, existen varias implementaciones creadas por distintos grupos e incluso varios protocolos. El concepto de llamada a procedimiento remoto no es otra que la que su propio nombre indica, es decir, se pretende ejecutar funciones o procedimientos en otras máquinas distintas a la nuestra y que el resultado retorne a nuestra máquina, todo ello de manera transparente. Hasta aquí el concepto de llamada a procedimiento remoto parece bastante asequible. El problema surge como siempre a la hora de implementarlo, codificarlo y tener en cuenta consideraciones de sistema.

Para empezar se deben tener en cuenta cosas como que tipo de protocolo se va a utilizar en capas inferiores. Puede ser **conectivo**, perfecto para olvidarse de las retransmisiones y fallos o **no conectivos** y aprovechar al máximo la capacidad de la red. También hay que prestar atención al formato de representación entre distintas

máquinas de un mismo sistema, como podrían ser ASCII, EBCDIC o Little Endian y Big Endian, y en otros muchos problemas que implican las llamadas a procedimientos remotos[17].

Por último (como colofón de las pegas de los sistemas implementados de los RPC) está el que generalmente no son tan transparentes como debieran. En el caso ideal, se supone que, el programador no debe saber si los procedimientos de una biblioteca son locales o remotos, ni necesitar de ningún tipo de interface para poder llamar a dichos procedimientos. Esto no se cumple, no sólo en el caso de RPC, sino en prácticamente cualquier sistema distribuido.

El RPC por excelencia es el RPC diseñado e implementado por Sun Microsystems cuya versión 2 esta especificada y tipificada en el RFC 1831. En esta RFC se puede leer no sólo el modelo de llamada a procedimiento remoto, sino también ejemplos de programación de RPC, semánticas de transporte, sistema de autenticación y modelo de programación en general de esta implementación. Dicha implementación utiliza como modelo de comunicación IP/UDP ya que la mayoría de las aplicaciones suelen estar en una LAN y por tanto se obtiene más efectividad que con TCP. Se utiliza también XDR para hacer compatible entre varios formatos de representación. Los programas requieren de librerías y funciones especiales y al mismo tiempo se requiere rpcportmapper instalado como demonio para efectuar las transacciones. Existen varias referencias acerca de como podría implementarse un sistema distribuido de este tipo en el libro *Sistemas Operativos Distribuidos* de Andrew S. Tanembaum.

¿Por qué no utilizar RPC en sistemas distribuidos o clusters? Existen diversas causas, unas con más peso que otras. Por norma general el modelo RPC suele ser bastante lento a pesar de utilizar UDP, por ejemplo en lo que se refiere al *checksum*, cambio de referencias en memoria, traducción XDR, rellenado de cabeceras y otras tantas operaciones que hacen que dicho sistema sea compatible en un entorno de trabajo hetereogéneo.

Es decir, en entornos de trabajo en los cuales no se requiere de confiabilidad, autenticación (y seguridad en general), eficiencia, y consistencia; la solución RPC es aceptable. En el caso de los clusters, suele implementarse con unos requerimientos de sistema bastante exigentes en lo que se requiere a eficiencia, y es mejor considerada una macro en el programa que utilizar todo un sistema como puede ser XDR. Generalmente los sistemas distribuidos suelen basar su desarrollo en nuevas implementaciones o modelos, que no tienen por que ser el de RPC, y no utilizan RPC debido a la poca eficiencia que este reporta y a la mala fama en seguridad que este posee.

**RMI.-**
También desarrollado por Sun para Java, mucha gente considera a RMI el sucesor de RPC.
RMI es un concepto confuso, tiene dos aceptaciones:

- RMI es cualquier protocolo que invoque métodos de objetos remotamente. Aparte de la implementación de Java, existen otras implementaciones que se pueden considerar RMI como las implementaciones de CORBA, DCOM y DCOP.
- RMI como Java RMI, se suele abreviar como RMI, es específico de Java y es del que hablaremos aquí pues las demás tecnologías las vemos en los siguientes apartados.

RMI es un mecanismo que permite invocar un método de un objeto que no existe en el espacio de direccionamiento de la propia aplicación, este otro espacio de direcciones puede encontrarse en la propia máquina o en otra diferente. Básicamente podemos aplicar lo explicado en RPC pero teniendo en cuenta que este mecanismo es orientado a objetos. Comparando con el siguiente punto CORBA, aunque básicamente ambos métodos son un RPC para lenguajes orientados a objetos, CORBA es un estandard e independiente del lenguaje, RMI sólo es para Java. CORBA incluye muchos otros mecanismos en su estándar (lo que le hace ser una implementación más lenta y pesada) que no son parte de RMI. Tampoco existe el concepto de ORB (Object Request Broker) en RMI. Java RMI ha estado evolucionando y convirtiéndose en más compatible con CORBA, en particular hay ahora una nueva forma de RMI llamada RMI/IIOP (RMI sobre IIOP) que usa IIOP (Internet Inter-ORB Protocol de CORBA como protocolo de bajo nivel para las comunicaciones RMI. Por supuesto podemos imaginar las consecuencias para el rendimiento.

Hay 3 procesos que participan en que la invocación de métodos remotos se haga efectiva:

- El cliente: es el proceso que está invocando a un método en un objeto remoto.
- El servidor: es el proceso al que pertenece el objeto remoto. El objeto remoto es un objeto ordinario en el espacio de direcciones del proceso servidor.

[17] Como el paso de variables por referencia que implica el paso de un buffer, con lo que se tienen que tener en cuenta, fallos de consistencia o simplemente eficiencia del proceso.

- El registro de objetos: es el servidor de nombre que relaciona objetos con nombres. Objetos están registrados con el registro de objetos. Una vez que un objeto ha sido registrado, cualquiera puede usar el registro de objetos para obtener acceso a los objetos remotos usando el nombre del objeto.

Hay dos tipos de clases que pueden ser usadas en Java RMI.

- **Remote class:** es una clase cuyas instancias pueden ser usadas remotamente. Un objeto de esa clase puede ser referenciado de dos maneras:

  1. Dentro de su propio espacio de direcciones donde el objeto fue construido, este objeto es un objeto ordinario Java.
  2. Dentro de otro espacio de direcciones, el objeto puede ser referenciado usando un manejador de objeto. Aunque hay limitaciones en cómo se puede usar un manejador de objeto comparado con un objeto, la mayor parte del tiempo los manejadores de objetos se pueden usar de la misma forma que un objeto.

- **Serialized class:** las instancias de este objeto (*serializable object*) pueden ser copiadas de un espacio de direcciones a otro. Se puede usar para el protocolo entre objetos. Si uno de estos objetos es pasado como parámetro (o es un valor de retorno) de una invocación remota a un método, entonces el valor del objeto será copiado de un espacio de direcciones a otro. En cambio si un *remote object* es pasado como parámetro ( o valor de retorno) el manejador del objeto será copiado de un espacio de direcciones a otro.

**CORBA.-**

Es un estándar desarrollado por una fundación creada por distintas empresas llamada OMG (*Object Management Group*) para poder hacer fácilmente programación distribuida, reutilización de objetos y de código ya hecho. CORBA cumple correctamente estos objetivos, para ello la estructura de CORBA cuenta de un lenguaje estandar llamado IDL (Interfaz Definition Language) que sirve para determinar los interfaces entre objetos, la implementación de los objetos se hace con cualquier lenguaje que tenga un compilador capaz de enlazar IDL.

CORBA es un *middleware* entre el sistema operativo y las aplicaciones usuario, entre ese middleware se encuentra el ORB encargado de hacer las llamadas entre objetos. En resumidas cuentas, CORBA es un nivel más de abstracción.

**— Ventajas:**

- Abstrae a la aplicación de todo lo referente a las comunicaciones, el programa ni siquiera sabe donde esta el objeto al que llama.
- Se pueden reutilizar programas anteriores simplemente añadiendo algo de código
- Tiene todas las ventajas de la programación orientada a objetos.
- Se puede programas en varios lenguajes. Puedes crear un programa con varios lenguajes distintos ya que cada una de las partes solo verá el interfaz CORBA de la otra.
- Las simplificaciones en la programación deberían llevar a una programación con menos errores.
- Abstrae el Sistema Operativo.
- Es un estándar para todas las plataformas y abierto.

**—Desventajas:**

- Más capas de software, más carga.
- Usa RPC en su nivel más bajo, anteriormente explicamos los problemas que acarrea esto.
- En sus versiones Real Time o QoS necesita dar preferencia a unas llamadas sobre otras para conseguir el objetivo.

- CORBA es muy grande, un fallo en CORBA podría ser difícil de detectar.
- Aunque se necesite añadir poco código las aplicaciones se necesitan reescribir.
- Un objeto se ejecuta en el sistema que le alberga.

Aunque haya más ventajas que inconvenientes estos últimos son más relevantes puesto que no podemos suponer que los procesos con los que trabajemos vayan a estar desarrollados en CORBA, además CORBA no nos soluciona el problema de migrar procesos y si lo hiciera sería extremadamente lento.

Aún se podría pensar en usar CORBA para envío de mensajes, pero la sobrecarga que genera no merece la pena. CORBA es muy potente para aplicaciones distribuidas o no de nueva creación y seguramente se le pudieran añadir funciones de balanceo de carga, pero para que nuestra solución sea más general es preferible no usar CORBA y tratar todo a nivel de proceso, sin abstracciones mayores. Aunque el estándar CORBA no añade estas capacidades de balanceo, implementación es específicas si que tienen reconfiguración dinámica, por ejemplo, Dinamic TAO.

**Bonobo.-**

Abstracción por encima de CORBA desarrollada por el proyecto GNOME.

Bonobo es un conjunto de interfaces CORBA, algunos implementados por objetos contenedores y otros por los objetos que serán contenidos (por ejemplo cuando pulsamos sobre un pdf y este se abre en nuestro propio navegador, el navegador es el contenedor y el programa lector de pdf el contenido). Bonobo es también una implementación por defecto de estos interfaces CORBA que es exportado en forma de API C para evitar a la mayoría de los programadores de preocuparse por tener que codificar directamente los interfaces CORBA.

Aunque no lo hemos mostrado en el punto anterior la programación en CORBA es ciertamente compleja, por tanto dentro del esfuerzo del proyecto GNOME para incorporar CORBA se ha desarrollado una librería de un nivel más alto llamada Bonobo (el nombre es de ciertos monos en vías de extinción que sirven para ilustrar la teoría de conexión de componentes). Otro de estos desarrollos fue ORBit que es un ORB pero con la particularidad de estar altamente mejorado en velocidad, siendo el más rápido existente (casi el doble que su más cercano competidor Omniorb) y de los que menos memoria necesita.

A parte de ORBit tenemos Gnorba que facilita en algo algunas tareas de ORBit y es más específico de GNOME, por ejemplo permite la integración del bucle principal de ORBit con el bucle principal de Gtk+, añade seguridad a las comunicaciones y un método estándar para acceder al servicio de nombres de GNOME. Para permitir acceso a un objeto en concreto se dispone la información en GOAD (*GNOME Object Activation Directory*) donde tenemos el id del objeto, las interfaces que exporta y como crear un ejemplar, gracias a esto se pueden usar métodos a través de la red.

Bonobo se creó para intentar mantener las aplicaciones con poca complejidad para que a los nuevos programadores les cueste poco saber cómo funciona el código y sea más fácil el mantenimiento y desarrollo. Usando todas las facilidades de CORBA que es usado para la capa de comunicación y para unir todos los componentes Bonobo entre sí. Cada componente tiene su interfaz (definido en términos de las interfaces CORBA) que exporta, ahíes donde cualquier otro componente puede conseguir información o unirse al componente.

El interface CORBA se introduce en un objeto GTK+, esto quiere decir que los objetos GTK+ son la implementación del interface IDL. Las implementaciones contienen ciertas acciones por defecto para no molestar al programador con detalles de CORBA, pero se dan unos métodos para cambiar este comportamiento.

**KDE.-**

En este apartado vamos a ver tres tecnologías que utiliza KDE a partir de sus versiones 2.0 y posteriores:

1. KIO: entrada y salida , transparencia de red.
2. XML-RPC: manejo de programas remotamente.
3. DCOP: comunicación entre componentes Kparts.

Estas tres tecnologías permiten acercarnos a un proceso distribuido y a una transparencia de red enormes. KDE es muy buen ejemplo de cómo se intenta cumplir algunos de los objetivos de los sistemas distribuidos, y cómo usuarios que somos de este entorno podemos decir que esta tecnología hace el día a día más sencillo.

- 1.- KIO:

  Esta es la librería que implementa prácticamente todas la funciones de manejo de ficheros. Es la librería que se usa para proveer un manejo de ficheros transparente a red. La implementación de los distintos protocolos (FTP, HTTP, SMB, POP3, IMAP4, gopher, printer, man, gzip, *etc.*) es hecha por un proceso independiente llamado kioslave, uno por cada protocolo, kio_ftp implementa FTP, kio_http implementa HTTP *etc.* La aplicación indica el protocolo a usar y KIO decide que kioslave usar.

  Por tanto gracias a estos kioslaves por ejemplo podemos tener acceso transparent e a servidores ftp, subir y bajar información simplemente copiando y pegando o haciendo *drag'n drop*. Podemos tener acceso a discos remotos e información remota de forma totalmente transparente. Un usuario sin mucho conocimiento de informática podría incluso nunca darse cuenta de que no estaba accediendo a su propio disco duro.

- 2.-XML-RPC

  Esta tecnología no es específica de KDE sino que se usa en este entorno como en otros, vamos a ilustrar el ejemplo de KDE. En KDE existe el kxmlrpcd que permite que un cliente escrito en cualquier lenguaje, en cualquier plataforma pueda acceder a los servidores DCOP de KDE . Como casi todas las aplicaciones de KDE son servidores DCOP, se puede manejar este entorno remotamente, de forma transparente. Por tanto vemos una vez más que el límite entre local y remoto se difumina permitiéndonos ejecutar KDE como si estuviéramos ejecutando un telnet.

  XmlRpc es la forma estándar de implementar RPC usando XML y HTTP. Con XML se marcan todas las llamadas a funciones , sus parámetros y sus valores de retorno. Entonces se utiliza HTTP para transferir la llamada al método. Muchísimos lenguajes disponen de parsers XML y clientes HTTP por lo que la implementación es bastante sencilla. Como el proyecto KDE ya tenía el protocolo DCOP para hacer RPC e IPC, kxmlrpcd es básicamente un traductor (servidor web por soportar HTTP), que traduce las peticiones XmlRpc en llamadas a DCOP y viceversa. Tanto el cliente XmlRpc como el servidor DCOP creen que están comunicándose solamente en su protocolo nativo con el demonio.

- 3.- DCOP : Desktop COmunication Protocol

  Fue creado por la necesidad de comunicación entre aplicaciones, no podían usar X Atoms por ser demasiado simples y crear una dependencia con Xwindow. También estuvieron intentando implementar CORBA pero se dieron cuenta que para la simple tarea de intercomunicar tareas era demasiado lento y requería demasiada memoria, además no tenía ningún tipo de autentificación.

  Lo que realmente se quería era un protocolo simple con autentificación básica, aunque no fuera capaz de hacer todo lo que era capaz de hacer CORBA, pero que fuera suficientemente bueno para la tarea encomendada. Un ejemplo de estas llamadas es una aplicación recién iniciada que pregunta si hay otra aplicación con el mismo nombre abierta, si es así abrirá una nueva ventana en la antigua aplicación en vez de abrir una nueva.

  DCOP es un mecanismos IPC/RPC muy simple que funciona sobre sockets. Están soportados los sockets del dominio UNIX y TCP/IP. Como capa inferior utiliza el protocolo ICE (Inter Client Exchange), que viene estándar como parte de X11R6. También depende de Qt, pero más allá de estos requerimientos no necesita más librerías. Es una capa de software muy ligera.

  Las aplicaciones son clientes DCOP y se comunican entre sía través de un servidor DCOP, que funciona como director del tráfico, enviando los mensajes o las llamadas a los destinos apropiados. Todos los clientes son pares entre ellos. Hay dos tipos de acciones posibles:

  Mensajes: en estos mensajes se envía el mensaje y no se espera la vuelta, por lo que la llamada no bloquea y es muy rápida.

  Llamadas: se llama a una función y se debe esperar a que se devuelva algún dato.

  Para crear las llamadas se ha desarrollado un compilador al estilo del compilador IDL que vimos en CORBA, llamado dcopidl (y dcopidl2cpp) que generan los stubs y skels al estilo que CORBA lo hacía para ahorrar el trabajo del programador.

  En definitiva DCOP provee un método sencillo y eficiente que fue desarrollado exactamente para ese uso que evita la complejidad de CORBA pero que también permite la interconexión de objetos en una

máquina o en máquinas distintas. Además en unión con XML-RPC del capítulo anterior se permite usar este protocolo desde cualquier lenguaje que era otra de las características de CORBA.

Las ventajas para crear aplicaciones distribuidas usando este protocolo son las mismas que se vieron en el apartado de CORBA.

**SOAP.-**

Hoy en día lo que se le pide a un sistema que use RPC o RMI es que sea confiable, robusto, que tenga APIs desarrolladas para varios lenguajes, interoperabilidad entre lenguajes, plataformas y modelos de componentes.

Desafortunadamente no hay un único sistema que tenga todas esas características. El formato de los datos y el protocolo usado para intercambiarlos es un factor determinante en el grado de interoperabilidad que pueden alcanzar las aplicaciones.

XML(eXtensible Markup Language) se está convirtiendo en un estandar para representar datos en un formato independiente de la plataforma. XML es un lenguaje fácil de general y de leer. HTTP (HyperText Transfer Protocol) también se está convirtiendo en un protocolo muy usado gracias a las aplicaciones web, que está soportado por todos los sistemas.

Las peticiones/respuesta de HTTP se pasan a través de firewall y son manejadas de forma segura, todo lo contrario que una ejecución de una llamada RPC o RMI.

Por lo tanto XML sobre HTTP parece una forma bastante inteligente para las aplicaciones distribuidas de comunicarse entre ellas. SOAP hace exactamente eso.

Representando RPCs de forma independiente a la plataforma, abre la posibilidad de implementar otros protocolos específicos de arquitectura en SOAP. Por ejemplo Microsoft parece querer usar SOAP como protocolo intermedio de intercambio al que otros protocolos pueden ser fácilmente traducidos, para potenciar el uso de sus componentes COM.

RPC en SOAP son interacciones cliente servidor sobre HTTP donde la petición/resp uesta se hacen de acuerdo con las reglas SOAP. Para elegir el objeto se usa el URI (Universal Resource Identifier) que es propio de HTTP o la cabecera SOAPAction que indica el nombre del interface. Se especifica una convención de la llamada remota , que determina la representa ción y el formato de las llamadas y respuestas. Una llamada es un dato compuesto con varios campos, uno por cada parámetro. Un mensaje de retorno es un valor de retorno y unos parámetros.

# Capítulo 3

# Implementación de Sistemas Distribuidos

## 3.1. SISTEMAS OPERATIVOS

*Microsoft isn't evil, they just make really crappy operating systems.*
*Linus Torvalds*

## Introducción

Los ordenadores se están convirtiendo en una plataforma cada vez más abierta y los sistemas operativos tienen que dar los servicios necesarios para que esto pueda gestionarse de manera adecuada. Dando un breve repaso a la historia de los sistemas operativos puede verse que cada vez tienen más posibilidades de comunicación.

Cuando se construyeron las primeras redes de computadores se vio la potencia que tenían estos sistemas y se desarrolló el paradigma cliente/servidor. Se crearon los sistemas operativos de red con servicios como NFS o FTP para acceder a sistemas de ficheros de otros ordenadores. `Xwindow` para entorno gráfico, `lpd` para poder imprimir remotamente además de un conjunto de herramientas que permitían el acceso a recursos compartidos.

En los aurales de la informática el sistema operativo ni siquiera existía: se programaba directamente el hardware. Se vio que era muy pesado y se programó una capa de abstracción que evitaba al programa tener que tener un conocimiento total del hardware: el sistema operativo. Se descubrió también que con un solo proceso se desperdiciaba tiempo de procesador y como tales dispositivos eran carísimos se empezaron a inventar mecanismos que evitasen que el programa esperase bloqueado por ejemplo las operaciones de entrada/salida. Se llegó a la multiprogramación en el ambicioso MULTICS y el posterior UNIX. Estos dos sistemas se desarrollaron a finales de los años 60 y se basaban en el *MIT Compatible Timesharing System* realizado en 1958.

En los años 70 se añadieron las primeras capacidades de interconexión:

- Xerox inventó la red Ethernet
- IBM la Token Ring
- y surgió UNIX BSD 4.2

Este sistema operativo fue ampliamente usado porque integraba capacidades de comunicación: implementaba TCP/IP y tenía capacidades de intercomunicación entre procesos.

Un sistema operativo distribuido puede acceder a cualquier recurso transparentemente y tiene grandes ventajas sobre un sistema operativo de red. Pero es muy difícil de implementar y algunos de sus aspectos necesitan que se modifique seriamente el núcleo del sistema operativo, por ejemplo para conseguir memoria distribuida transparente a los procesos. También se tiene que tener en cuenta que ahora la *máquina* sobre la que corre todo el sistema es el sistema distribuido (un conjunto de nodos) por lo que tareas como planificación de procesos (*scheduling*) o los hilos (*threads*) y señales entre procesos toman una nueva dimensión de complejidad. Por todo esto aún no existe ningún sistema distribuido con todas las características de transparencia necesarias para considerarlo eficiente.

A partir de ahora no hablaremos de computadores, ordenadores ni PC. Cualquier dispositivo que se pueda conectar a nuestro sistema y donde se pueda desarrollar un trabajo útil para el sistema se referenciará como **nodo**. El sistema operativo tiene que controlar todos estos nodos y permitir que se comuniquen eficientemente.

Aunque tras leer este capítulo el lector podrá intuir esta idea, queremos hacer notar que un sistema distribuido se puede ver como el siguiente nivel de abstracción sobre los sistemas operativos actuales. Como hemos visto la tendencia es a dar más servicios de red y más concurrencia. Si pusiéramos otra capa por encima que aunara ambos, esta capa sería el sistema operativo distribuido. Esta capa debe cumplir los criterios de transparencia que veremos en el próximo apartado.

Podemos comprender esto mejor haciendo una analogía con el SMP: cuando sólo existe un procesador todos los procesos se ejecutan en él, tienen acceso a los recursos para los que tengan permisos, se usa la memoria a la que puede acceder el procesador, *etc*. Cuando hay varios procesadores todos ellos están corriendo procesos (si hay suficientes para todos) y los procesos migran entre procesadores.

En un sistema multiprocesador al igual que un sistema multicomputador hay una penalización si el proceso tiene que correr en un elemento de proceso (procesador o nodo) distinto al anterior: en multiprocesador por la contaminación de cache y TLB; en multicomputador por la latencia de la migración y pérdidas de cache. En un

sistema multicomputador se persigue la misma transparencia que un sistema operativo con soporte multiprocesador, esto es, que no se deje ver a las aplicaciones que realmente están funcionando en varios procesadores.

En este capítulo además se tratarán las zonas de un sistema operativo que se ven más afectadas para conseguirlo. Éstas son:

- **Scheduling.**

  El *scheduling* de los procesos ahora puede ser realizado por cada nodo de manera individual, o mediante algún mecanismo que implemente el cluster de manera conjunta entre todos los nodos.

- **Deadlocks.**

  Las técnicas usadas hasta ahora en un nodo local no se pueden generalizar de una forma eficiente en el sistema distribuido.

- **Manejo de memoria.**

  Hay que disponer de memoria distribuida: poder alojar y desalojar memoria de otros nodos de una forma transparente a las aplicaciones.

- **Intercomunicación de procesos.**

  Los procesos podrían estar ubicados en cualquier nodo del sistema. Se deben proveer de los mecanismos necesarios para permitir esta intercomunicación de tipo señales, memoria distribuida por el cluster u otros.

- **Sistemas de ficheros.**

  Pretenden que todo el sistema distribuido tenga la misma raíz para que no sea necesario explicitar en qué nodo se encuentra la información.

- **Entrada/salida.**

  Tendría que ser posible acceder a todos los recursos de entrada/salida globalmente, sin tener que indicar explícitamente a que nodo están estos recursos conectados.

### 3.1.1. Procesos y Scheduling

#### Utilidad de migrar procesos

La migración de un proceso consiste en mover el proceso desdel nodo donde se está ejecutando a un nuevo entorno. Aunque no se suela emplear en este caso, se podría hablar de migración cuando un proceso se mueve de procesador. Aquí consideraremos migración cuando un proceso se mueve de un nodo a otro.

En un sistema de tipo cluster lo que se pretende, como se verá en el capítulo *Clusters*, es compartir aquellos dispositivos (a partir de ahora serán los *recursos*) conectados a cualquiera de los nodos. Uno de los recursos que más se desearía compartir, aparte del almacenamiento de datos, es el procesador de cada nodo. Para compartir el procesador entre varios nodos lo más lógico es permitir que las unidades atómicas de ejecución del sistema operativo (procesos, con PID propio) sean capaces de ocupar en cualquier momento cualesquiera de los nodos que conforman el cluster.

En un sistema multiusuario de tiempo compartido la elección de qué proceso se ejecuta en un intervalo de tiempo determinado la hace un segmento de código que recibe el nombre de *scheduler*, el planificador de procesos. Una vez que el scheduler se encarga de localizar un proceso con las características adecuadas para comenzar su ejecución, es otra sección de código llamada *dispatcher* la que se encarga de substituir el contexto de ejecución en el que se encuentre el procesador, por el contexto del proceso que queremos correr. Las cosas se pueden complicar cuando tratamos de ver este esquema en un multicomputador. En un cluster, se pueden tener varios esquemas de actuación.

- *Scheduling* conjunto. En el cual todos los nodos del cluster hacen la elección de los procesos que se ejecutaran en el cluster y quizá incluso en cada nodo. Esto requiere una actuación conjunta de todos los nodos, lo que implica que el scheduling podría ser:

  - Centralizado: una o varias máquinas trabajando para efectuar las tareas de scheduler y organizar el resto del cluster. Es sencillo de implementar, pero tiene más inconvenientes que ventajas, aparte que los inconvenientes son de mucho peso y muy obvios.

  - Distribuido: requieren un modelo matemático adecuado y una implementación más complicada.

- *Scheduling* individual: cada nodo actúa de la manera comentada en sistemas individuales, pero añade la acción de migrar procesos a otros nodos, de manera que se optimice tiempo de ejecución global en el cluster. Para esto, los nodos comparan la carga de otros nodos y deciden cúal de los procesos que tienen en la cola de espera para ser ejecutados es el óptimo para ser enviado a otro nodo.

Conforme se avance en este manual se verá cuál puede convenir más para cada caso. Otra de las políticas que puede variar del scheduling en un ordenador individual a un cluster es la de diseminación de la información de los nodos. A la hora de elegir sobre qué nodo migrar un proceso, tanto en el *scheduling* conjunto como en el individual, se debe conocer la información de los nodos para balancear convenientemente el sistema. Esta diseminación puede estar basada en:

- Información global.
- Información parcial.

De estos dos, el primero no es muy escalable, pero probablemente sea más adecuado en ciertas condiciones. Lo más habitual es que un cluster deje las decisiones sobre cuándo ejecutar cada proceso a cada nodo particular, por tanto la principal novedad en un sistema distribuido es que hay que permitir que los procesos se ejecuten en cualquier computador no necesariamente en el que se crearon. Y transparentemente a los mismos procesos. Este ha sido un tema estudiado numerosas veces, con todo tipo de conclusiones, seguramente la más sorprendente es la de Tanenbaum[1]. Se equivocó (una vez más) porque openMosix ha conseguido la migración en la práctica. Esta migración de procesos tiene varias ventajas, como son:

- Compartición de carga: los procesos se trasladarán a la máquina con menos carga para evitar que haya sistemas sobrecargados. También se tiene en cuenta cuándo se está acabando la memoria de la máquina. En definitiva los procesos van donde tengan más recursos. En el caso óptimo se tendría la potencia de todas las máquinas sumadas, pues con un balanceo perfecto podrían tenerse a todos los nodos trabajando a máximo rendimiento, aunque el trabajo original se produjese en un nodo solo.

  Por supuesto este este es el caso ideal y no se puede dar en la realidad pues siempre se necesita potencia de cálculo para enviar y recibir los procesos así como para tomar las decisiones (*overhead*).

- Rendimiento de las comunicaciones: los procesos que necesitan entrada/salida en un nodo se desplazan a éste para evitar traer todos los datos a través de la red. Otra variante es mantener a todos los procesos que interactúan fuertemente entre ellos en una misma máquina.

Las principales decisiones que se deben tomar en un sistema distribuido para hacer esta migración eficiente las podemos englobar en tres políticas distintas:

- Políticas de localización
- Políticas de migración
- Políticas de ubicación

A parte de estas políticas también hay que hacer otras consideraciones que no tienen que ver con las decisiones que se toman sino cómo van a ser implementadas en la práctica, estas son:

- Parte del proceso a migrar
- Mecanismos de migración

#### Política de localización

Cubre las decisiones referentes a desde dónde se lanzaran los procesos.

Este tipo de planteamiento sólo se efectúa en los sistemas SSI de los que se hablará más tarde, ya que necesita un acoplamiento muy fuerte entre los nodos del cluster. Un ejemplo puede ser el requerimiento de un cluster que dependiendo del usuario o grupo de usuario elija un nodo preferente donde ejecutarse por defecto. En openMosix no se hace uso de esta política, ya que el sistema no está tan acoplado como para hacer que otro nodo arranque un programa.

[1] *La migración real de procesos en ejecución es trivial en teoría, pero cerca de lo imposible en la práctica*, del libro publicado junto a R. Renesse *Distributed Operating Systems*

### Política de migración

La política de migración plantea las decisiones que hay que hacer para responder a preguntas, a saber:

- ¿cuándo se inicia la migración?
- ¿quién inicia la migración?

Para saber cuándo se debe realizar una migración nuestro nodo debe estar en contacto con los demás nodos y recolectar información sobre su estado y así, y teniendo en cuenta otros parámetros como la carga de la red, se debe hacer una decisión lo más inteligente posible y decidir si es momento de migrar y qué es lo que se migra.

Las causas que hacen que se quiera realizar la migración van a depender del objetivo del servicio de la migración. Así si el objetivo es maximizar el tiempo usado de procesador, lo que hará que un proceso migre es el requerimiento de procesador en su nodo local, así gracias a la información que ha recogido de los demás nodos decidirá si los merece la pena migrar o en cambio los demás nodos están sobrecargados también.

La migración podría estar controlada por un organismo central que tuviera toda la información de todos los nodos actualizada y se dedicase a decidir como colocar los procesos de los distintos nodos para mejorar el rendimiento. Esta solución aparte de ser poco escalable, pues se sobrecargan mucho la red de las comunicaciones y uno de los equipos, es demasiado centralizada ya que si este sistema falla se dejarán de migrar los procesos.

El otro mecanismo es una toma de decisiones distribuida: cada nodo tomará sus propias decisiones usando su política de migración. Dentro de esta aproximación hay dos entidades que pueden decidir cuando migrar un proceso: el propio proceso o el kernel del sistema operativo.

- Si el proceso es quien va a decidirlo hay el problema de que él tiene que ser consciente de la existencia de un sistema distribuido.
- En cambio si es el kernel quien decide, la función de migración y la existencia de un sistema distribuido pueden ser transparentes al proceso.

Esta última es la política que usa openMosix.

### Política de ubicación

Encontrar el mejor nodo donde mandar un proceso que está sobrecargando el nodo de ejecución no es una estrategia fácil de implementar. Entran en juego una gran amplia gama de algoritmos generalmente basados en las funcionalidades o recursos a compartir del cluster, y dependiendo de cuáles sean estos, se implantarán unas medidas u otras. El problema no es nuevo y puede ser tratado como un problema de optimización normal y corriente, pero con una severa limitación. A pesar de ser un problema de optimización se debe tener en cuenta que este código va a ser código de kernel, por lo tanto y suponiendo que está bien programado, el tiempo de ejecución del algoritmo es crucial para el rendimiento global del sistema. ¿De qué sirve optimizar un problema de ubicación de un proceso que va a tardar tres segundos en ejecutarse, si dedicamos uno a decidir donde colocarlo de manera óptima y además sin dejar interactuar al usuario?

Este problema penaliza a ciertos algoritmos de optimización de nueva generación como las redes neuronales, o los algoritmos genéticos y beneficia a estimaciones estadísticas.

### Parte de los procesos a migrar

Una vez se sabe cuándo, de dónde y hacia dónde migrará el proceso, falta saber qué parte del proceso se va a migrar. Hay dos aproximaciones:

1. **Migrar todo el proceso.**

   Implica destruirlo en el sistema de origen y crearlo en el sistema de destino. Se debe mover la imagen del proceso que consta de, por lo menos, el bloque de control del proceso (a nivel del kernel del sistema operativo). Además, debe actualizarse cualquier enlace entre éste y otros procesos, como los de paso de mensajes y señales (responsabilidad del sistema operativo). La transferencia del proceso de una máquina a otra es invisible al proceso que emigra y a sus asociados en la comunicación.

   El movimiento del bloque de control del proceso es sencillo. Desde el punto de vista del rendimiento, la dificultad estriba en el espacio de direcciones del proceso y en los recursos que tenga asignados. Considérese primero el espacio de direcciones y supóngase que se está utilizando un esquema de memoria virtual (segmentación y/o paginación). Pueden sugerirse dos estrategias:

- Transferir todo el espacio de direcciones en el momento de la migración. No hace falta dejar rastro del proceso en el sistema anterior. Sin embargo si el espacio de direcciones es muy grande y si es probable que el proceso no necesite la mayor parte, este procedimiento puede ser costoso.
- Transferir sólo aquella parte del espacio de direcciones que reside en memoria principal. Cualquier bloque adicional del espacio de direcciones virtuales será transferido sólo bajo demanda. Esto minimiza la cantidad de datos que se transfieren. Sin embargo, se necesita que la máquina de origen siga involucrada en la vida del proceso, pues mantiene parte de su espacio de direcciones, nótese que este sistema no tiene nada que ver con el anterior, pues el sistema de origen es un mero almacenador de información.

  También es una estrategia obligada cuando lo que se migran son hilos en vez de procesos y no migramos todos los hilos de una vez. Esto está implementado en el sistema operativo Emerald[2] y otras generalizaciones de este a nivel de usuario.

2. **Migrar una parte del proceso.**

   En este sistema:

   - el contexto del nivel de usuario del proceso es llamado *remote*: contiene el código del programa, la pila, los datos, el mapeo de la memoria y los registros del proceso. El *remote* encapsula el proceso cuando éste está corriendo en el nivel de usuario.
   - El contexto del kernel es llamado *deputy*: contiene la descripción de los recursos a los cuales el proceso está unido, y la pila del kernel para la ejecución del código del sistema cuando lo pida el proceso.

     Mantiene la parte del contexto del sistema que depende del lugar por lo que se mantiene en el nodo donde se generó el proceso.

   Como en Linux la interfície entre el contexto del usuario y el contexto del kernel está bien definida, es posible interceptar cada interacción entre estos contextos y enviar esta interacción a través de la red. Esto está implementado en el nivel de enlace con un canal especial de comunicación para la interacción.

   El tiempo de migración tiene una componente fija que es crear el nuevo proceso en el nodo al que se haya decidido migrar; y una componente lineal proporcional al número de páginas de memoria que se vayan a transferir. Para minimizar la sobrecarga de esta migración, de todas las páginas que tiene mapeadas el proceso sólo se van a enviar las tablas de páginas y las páginas en las que se haya escrito.

   OpenMosix consigue transparencia de localización gracias a que las llamadas dependientes al nodo nativo que realiza el proceso que ha migrado se envían a través de la red al *deputy*. Así openMosix intercepta todas las llamadas al sistema, comprueba si son independientes o no, si lo son las ejecuta de forma local (en el nodo remoto) sino la llamada se emitirá al nodo de origen y la ejecutará el *deputy*. Éste devolverá el resultado de vuelta al lugar remoto donde se continua ejecutando el código de usuario.

**Mecanismos de migración**

Normalmente las políticas de ubicación y localización no suelen influir en estos mecanismo pues en general, una vez que el proceso migra no importa donde migre. Si se ha decidido que los mismos procesos son quienes lo deciden todo (automigración) se llega a un mecanismo de migración que es el usado en el sistema operativo AIX de IBM. Para este caso el procedimiento que se sigue cuando un proceso decide migrarse es:

1. Se selecciona la máquina destino y envía un mensaje de tarea remota. El mensaje lleva una parte de la imagen del proceso y de información de archivos abiertos.
2. En el nodo receptor, un proceso servidor crea un hijo y le cede esta información.
3. El nuevo proceso extrae los datos, los argumentos, la información del entorno y de la pila que necesita hasta completar su operación.
4. Se indica con una señal al proceso originario que la migración ha terminado. Este proceso envía un mensaje final para terminar la operación al nuevo proceso y se destruye.

[2] Mas información en *Migration of Light-Weight Processes in Emerald*. Carta de la sociedad de sistemas operativos de la IEEE.

| Instancia 1 | Instancia 2 |
|---|---|
| lee(contador) | |
| suma(contador,1) | |
| escribe(contador) | |
| | lee(contador) |
| | suma(contador,1) |
| | escribe(contador) |

Cuadro 3.1: Sistemas Operativos. Compartición de recursos (1)

| Instancia 1 | Instancia 2 |
|---|---|
| lee(contador) | |
| añade(contador,1) | lee(contador) |
| escribe(contador) | añade(contador,1) |
| | escribe(contador) |
| | |
| | |

Cuadro 3.2: Sistemas Operativos. Compartición de recursos (2)

Si en cambio es otro proceso el que comienza la migración en vez de ser el propio proceso tenemos el sistema que se usa en Sprite: se siguen los mismos pasos que en AIX pero ahora el proceso que maneja la operación lo primero que hace es suspender el proceso que va a migrar para hacerlo en un estado de no ejecución. Si el proceso que decide no está en cada máquina sino que hay solamente uno que hace decisiones globales, hay una toma de decisiones centralizada. En este caso se suelen usar unas instrucciones especiales desde esa máquina a las máquinas origen y destino de la migración, aunque básicamente se sigue el patrón expuesto del sistema AIX.

También puede ocurrir que se necesite un consenso de un proceso en la máquina origen y otro proceso en la máquina destino, este enfoque tiene la ventaja de que realmente se asegura que la máquina destino va a tener recursos suficientes, esto se consigue así:

1. El proceso controlador de la máquina origen (después de las decisiones oportunas) le indica al proceso controlador de la máquina destino que le va a enviar un proceso, este proceso le responde afirmativamente si está preparado para recibir un proceso.

2. Ahora el proceso controlador de la máquina origen puede hacer dos cosas: ceder el control al núcleo para que haga la migración o pedir al núcleo estadísticas del proceso y se las envía al proceso controlador destino (el núcleo hace lo mismo si toma el control).

3. El proceso controlador de la máquina destino, o su kernel, toman esa información (el proceso se la enviaría al kernel) y deciden si hay suficientes recursos en el sistema para el nuevo proceso. Si lo hubiera el núcleo reserva los recursos necesarios para el proceso y lo notifica al proceso controlador que notifica al proceso controlador de la máquina origen que proceda con la migración.

### 3.1.2. Compartición de recursos

**Problemas**

Para entender como compartir recursos en un sistema distribuido se plantean nuevos problemas respecto a compartirlos en un sólo sistema. Véase con un ejemplo: supónganse dos instancias del mismo proceso que comparten memoria y acceden a una misma variable `contador` contenida en esta e inicializada con valor 5. Luego se incrementa `contador`. Lo que se espera es: Si esto ocurriera así la primera instancia del programa pondría el contador a 6, entonces la segunda instancia del programa, leyendo ese valor, escribiría el valor 7.

Sin usar los mecanismos necesarios que controlen la ejecución, ésta es una de las trazas de lo que podría ocurrir: El programa parece fallar de forma aleatoria, puesto que no podemos garantizar cuál será su ejecución

real.

Esto es lo que se llama una *condición de carrera* y desde que GNU/Linux funciona en máquinas SMP (a partir de la versión 2.0 ) se ha convertido en un tema principal en su implementación. Por supuesto no es un problema único de este sistema operativo sino que atañe a cualquiera con soporte multitarea. La solución pasa por encontrar estos puntos y protegerlos por ejemplo con semáforos, para que sólo uno de los procesos pueda entrar a la vez a la región crítica.

Esta es seguramente la situación más sencilla: compartición de una variable en memoria. Para aprender como solucionar éstas y otras situaciones de conflicto se recomienda al lector consultar los autores de la bibliografía.

### 3.1.3. Comunicación entre procesos

En un cluster existe un nuevo problema: al mover un proceso a otro nodo, ese proceso debe seguir pudiendo comunicarse con los demás procesos sin problemas, por lo que es necesario enviar las señales que antes eran locales al nodo a través de la red.

En los clusters donde los procesos tienen consciencia de si estan siendo ejecutados locales o remotos, cada nodo tiene las primitivas de comunicación necesarias para enviar toda la comunicación a través de la red. En clusters donde solo el kernel puede conocer este estado de los procesos, estas primitivas se hacen innecesarias pues la transparencia suple esta capa. Éste es el caso de openMosix.

Hay mecanismos de comunicación más problemáticos que otros. Las señales no lo son demasiado pues se pueden encapsular en un paquete qué se envíe a través de la red. Aquí se hace necesario que el sistema sepa en todo momento el nodo dónde está el proceso con el que quiere comunicar.

Otros mecanismos de comunicación entre procesos son más complejos de implementar. Por ejemplo la memoria compartida: se necesita tener memoria distribuida y poder/saber compartirla. Los sockets también son candidatos difíciles a migrar por la relación que tienen los servidores con el nodo.

### 3.1.4. La importancia de los sistemas de ficheros

Los sistemas de ficheros son necesarios en nuestros sistemas para mantener la información y compartirla entre usuarios y programas. Un fichero no es más que una abstracción del dispositivo de almacenaje permanente.

El sistema de ficheros tradicional tiene como funciones la organización, almacenaje, recuperación, protección, nombrado y compartición de ficheros. Para conseguir nombrar los ficheros se usan los directorios, que no son más que un fichero de un tipo especial que provee una relación entre los nombre que ven los usuarios y un formato interno del sistema de ficheros.

Un sistema de ficheros distribuido es tan importante para un sistema distribuido como para uno tradicional: debe mantener las funciones del sistema de ficheros tradicional, por lo tanto los programas deben ser capaces de acceder a ficheros remotos sin copiarlos a sus discos duros locales. También proveer acceso a ficheros en los nodos que no tengan disco duro. Normalmente el sistema de ficheros distribuido es una de las primeras funcionalidades que se intentan implementar y es de las más utilizadas por lo que su buena funcionalidad y rendimiento son críticas. Se pueden diseñar los sistemas de ficheros para que cumplan los criterios de transparencia, esto es:

- **Transparencia de acceso:** los programas no tienen por qué saber como están distribuidos los ficheros. Se deben usar las mismas instrucciones para acceder a los fichero locales o remotos por lo tanto los programas escritos para usar ficheros locales podrán usar ficheros remotos sin ninguna modificación.

- **Transparencia de localización:** los programas deben ver un espacio de nombres de ficheros uniforme, los ficheros o grupos de ficheros podrían ser cambiados de lugar sin que cambien su *path* ni sus nombres. El programa podrá ser ejecutado en cualquier nodo y verá el mismo espacio de nombres.

- **Transparencia de concurrencia:** los cambios a un fichero ocasionados por un cliente no deberían interferir con las operaciones de otros clientes que estén simultáneamente accediendo o cambiando el mismo fichero.

- **Transparencia a fallos:** se debe conseguir una operación correcta tanto si falla el cliente como el servidor, haya perdida de mensajes o interrupciones temporales.

- **Transparencia de réplica:** un fichero podría estar representado por varias copias de sus contenidos en lugares diferentes. Esto tiene varias ventajas, permite a muchos servidores compartir la carga si un número

de clientes están accediendo al mismo conjunto de ficheros aumentando la escalabilidad, permite tener copias en cache, más cercanas al lugar donde se están utilizando, aumentando el rendimiento y permitiendo a los clientes seguir accediendo a los ficheros aunque alguno de los servidores haya tenido algún error aumentando la tolerancia a fallos.

- **Transparencia de migración:** de todo lo anterior se concluye que un proceso o un fichero puede migrar en el sistema sin tener que preocuparse por un nuevo *path* relativo.

Para comprender mejor el problema vamos a ver cuatro sistemas de ficheros que desde el más simple a otros más complejos han intentado solucionar estos problemas hasta cierto grado. Estos sistemas de ficheros son:

- NFS, Network File System.
- MFS, Mosix File System. Necesario en openMosix para proveer de acceso directo al FS y mantener consistencias de cache.
- GFS. Es un sistema de ficheros para discos compartidos.

A continucación se enumeran sus propiedades básicas y se hará una pequeña comparación entre ellos. La elección de estos sistemas de ficheros entre los muchos que existen no es casual: NFS es bien conocido y aunque existen otros sistemas similares y más avanzados (como AFS, Coda o Intermezzo que como NFS dependen de un servidor central) sus características avanzadas (cache en los clientes de AFS y la adaptación al ancho de banda, reintegración de una comunicación perdida y múltiples peticiones RPC de Coda, simpleza y distinción kernel/programa de usuario de Intermezzo) hacen que sea más complejo comprender las características que se quiere destacar en esta comparativa.

- **NFS**

  Es uno de los primeros sistemas de archivos de red, fue creado por Sun basado en otra obra de la misma casa, RPC y parte en XDR para que pudiese implementarse en sistemas hetereogéneos. El modelo NFS cumple varias de las transparencias mencionadas anteriormente de manera parcial. A este modelo se le han puesto muchas pegas desde su creación, tanto a su eficiencia como a su protocolo, como a la seguridad de las máquinas servidoras de NFS.

  El modelo de NFS es simple: máquinas servidoras que exportan directorios y máquinas clientes[3] que montan este directorio dentro de su árbol de directorio y al que se accederá de manera remota.

  De esta manera el cliente hace una llamada a *mount* especificando el servidor de NFS al que quiere acceder, el directorio a importar del servidor y el punto de anclaje dentro de su árbol de directorios donde desea importar el directorio remoto. Mediante el protocolo utilizado por NFS[4] el cliente solicita al servidor el directorio exportable (el servidor en ningún momento sabe nada acerca de dónde esta montado el sistema en el cliente), el servidor en caso de que sea posible la operación, concede a el cliente un *handler* o manejador del archivo, dicho manejador contiene campos que identifican a este directorio exportado de forma única por un i-node, de manera que las llamadas a lectura o escritura utilizan esta información para localizar el archivo.

  Una vez montado el directorio remoto, se accede a él como si se tratase del sistema de archivos propio, es por esto que en muchos casos los clientes montan directorios NFS en el momento de arranque ya sea mediante scripts `rc` o mediante opciones en el fichero */etc/fstab*. De hecho existen opciones de *automount* para montar el directorio cuando se tenga acceso al servidor y no antes, para evitar errores o tiempos de espera innecesarios en la secuencia de arranque.

  NFS soporta la mayoría de las llamadas a sistema habituales sobre los sistemas de ficheros locales así como el sistema de permisos habituales en los sistemas UNIX. Las llamadas *open* y *close* no están implementadas debido al diseño de NFS. El servidor NFS no guarda en ningún momento el estado de las conexiones establecidas a sus archivos [5], en lugar de la función *open* o *close* tiene una función *lookup* que no copia información en partes internas del sistema. Esto supone la desventaja de no tener un sistema de bloqueo en el propio NFS, aunque esto se ha subsanado con el demonio **rpc.lockd** lo que implicaba que podían darse

[3] Si bien una misma máquina puede ser cliente y servidor a la vez.
[4] Que tiene implementación en varios sistemas operativos no solo en UNIX o derivados, por ejemplo MSDOS.
[5] De hecho no guarda ni las conexiones con los clientes ya que son UDP, por el poco control de coherencia de las caches que tiene.

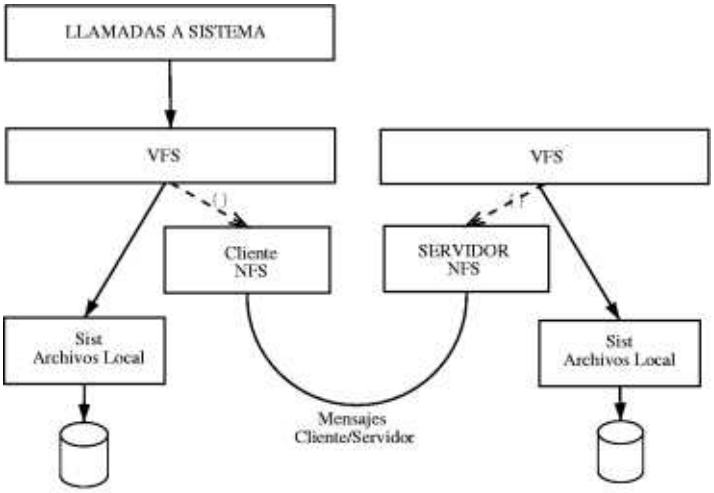



Figura 3.1: Sistemas operativos. NFS

situaciones en las cuales varios clientes estuviesen produciendo inconsistencias en los archivos remotos. Por otro lado, el no tener el estado de las conexiones, implica que si el cliente cae, no se produce ninguna alteración en los servidores.

La implementación de NFS de manera general es la siguiente. Se efectúa una llamada al sistema del tipo *open(), read(), close()* después de analizar la llamada, el sistema pasa ésta al VFS que se encarga de gestionar los archivos abiertos mediante tablas de *v-nodes*. Estos v-nodes apuntan a archivos de varios tipos, locales, remotos, de varios sistemas de archivos, y especifican a su vez que operaciones deben hacer en cada caso.

De esta manera para el caso de NFS, VFS guarda en el v-node un apuntador al nodo remoto en el sistema servidor,que define a la ruta del directorio compartido y a partir de este todos son marcados como v-nodes que apuntan a r-nodes o nodos remotos.

En el caso de una llamada *open()* a un archivo que se encuentra en parte de la jerarquía donde está un directorio importado de un servidor NFS, al hacer la llamada, en algún momento de la comprobación de la ruta en el VFS, se llegara al v-node del archivo y se verá que este corresponde a una conexión NFS, procediendo a la solicitud de dicho archivo mediante opciones de lectura, se ejecuta el código especificad o por VFS para las opciones de lectura de NFS.

En cuanto al manejo de las caches que se suelen implementar en los clientes depende de cada caso específico. Se suelen utilizar paquetes de 8KB en las transferencias de los archivos, de modo que para las operaciones de lectura y escritura se espera a que estos 8KB estén llenos antes de enviar nada, salvo en el caso de que se cierre el archivo. Cada implementación establece un mecanismo simple de *timeouts* para las caches de los clientes de modo que se evite un poco el trafico de la red.

La cache no es consistente, los ficheros pequeños se llaman en una cache distinta a esos 8KB y cuando un cliente accede a un fichero al que accedió poco tiempo atrás y aún se mantiene en esas caches, se accede a esas caches en vez de la versión del servidor. Esto hace que en el caso de ficheros que no hayan sido actualizados se gane el tiempo que se tarda en llegar hasta el servidor y se ahorre tráfico en la red. El problema es que produce inconsistencias en la memoria cache puesto que si otro ordenador modificó esos ficheros, nuestro ordenador no va a ver los datos modificados sino que leerá la copia local.

Esto puede crear muchos problemas y es uno de los puntos que más se le discute a NFS, este problema es el que ha llevado a desarrollar MFS pues se necesitaba un sistema que mantuviera consistencia de caches.

- **MFS.**

  Este es el sistema de ficheros que se desarrolló para openMosix en espera de alguno mejor para poder hacer uso de una de sus técnicas de balanceo, DFSA. Este sistema funciona sobre los sistemas de ficheros

| Método de acceso | 64 | 512 | 1K | 2K | 4K | 8K | 16K |
|---|---|---|---|---|---|---|---|
| Local | 102.6 | 102.1 | 100.0 | 102.2 | 100.2 | 100.2 | 101.0 |
| MFS con DFSA | 104.8 | 104.0 | 103.9 | 104.0 | 104.9 | 105.5 | 104.4 |
| MFS sin DFSA | 1711.0 | 169.1 | 158.0 | 161.3 | 156.0 | 159.5 | 157.5 |
| NFS | 184.3 | 169.1 | 158.0 | 161.3 | 156.0 | 159.5 | 157.5 |

Cuadro 3.3: Sistemas Operativos. MFS

locales y permite el acceso desde los demás nodos.

Cuando se instala, se dispone de un nuevo directorio que tendrá varios subdirectorios con los números de los nodos, en cada uno de esos subdirectorios se tiene todo sistema de ficheros del nodo en cuestión[6]. Esto hace que este sistema de ficheros sea genérico pues */usr/src/linux/* podría estar montado sobre cualquier sistema de ficheros; y escalable, pues cada nodo puede ser potencialmente servidor y cliente.

A diferencia de NFS provee una consistencia de cache entre procesos que están ejecutándose en nodos diferentes, esto se consigue manteniendo una sola cache en el servidor. Las caches de GNU/Linux de disco y directorio son sólo usadas en el servidor y no en los clientes. Así se mantiene simple y escalable a cualquier número de procesos.

El problema es una pérdida de rendimiento por la eliminación de las caches, sobre todo con tamaños de bloques pequeños. La interacción entre el cliente y el servidor suele ser a nivel de llamadas del sistema lo que suele ser bueno para la mayoría de las operaciones de entrada/salida complejas y grandes.

Este sistema cumple con los criterios de transparencia de acceso, no cumple con los demás. Tiene transparencia de acceso pues se puede acceder a estos ficheros con las mismas operaciones que a los ficheros locales. Los datos del cuadro 3.3 son del Postmark[7] benchmark que simula grandes cargas al sistema de ficheros, sobre GNU/Linux 2.2.16 y dos PCs Pentium 550 MHz[8]:

- GFS:

  Se basa en que las nuevas tecnologías de redes (como fibra óptica) permiten a muchas máquinas compartir los dispositivos de almacenamiento. Los sistemas de ficheros para acceder a los ficheros de estos dispositivos se llaman sistemas de ficheros de dispositivos compartidos. Contrastan con los sistemas de ficheros distribuidos tradicionales donde el servidor controla los dispositivos (físicamente unidos a él). El sistema parece ser local a cada nodo y GFS sincroniza los acceso a los ficheros a través del cluster. Existen dos modos de funcionamiento, uno con un servidor central (asimétrico) y otro sin él (simétrico).

  En el modo que necesita servidor central, éste tiene el control sobre los metadatos (que es un directorio donde están situados fechas de actualización, permisos, etc.), los discos duros compartidos solamente contienen los datos. Por tanto todo pasa por el servidor, que es quien provee la sincronización entre clientes, pues estos hacen las peticiones de modificación de metadata al servidor (abrir, cerrar, borrar, etc.) y leen los datos de los discos duros compartidos, es similar a un sistema de ficheros distribuido corriente. En la figura 3.2 se muestra una configuración típica de este sistema:

El modo que no necesita servidor central es llamado modo simétrico, los discos contienen datos y metadatos, que son controlados por cada máquina al ser accedidos, estos accesos son sincronizados gracias a locks globales, que se apoyan en la ayuda del hardware tanto por parte de SCSI (DMEP) como por parte del switch (DLM). Esto hace esta alternativa aún más cara. En la figura 3.3 se muestra un gráfico con una configuración típica de este sistema. Se puede ver que la Network Storage Pool no tiene procesadores, pues está directamente conectada a la red, gracias a fibra óptica y SCSI. La *Storage Area Network* típicamente es un *switch* de alta velocidad). Entre las características que este sistema de ficheros se encuentra que no hay un solo punto de fallo, si un cliente falla, o si incluso muchos clientes quedasen inutilizables, el sistema de ficheros estaría todavía ahíaccesible a todos los clientes que aún estuvieran funcionando. Gracias a que los discos están compartidos por todos los clientes todos los datos a los que los clientes que dieron error estaban accediendo están todavía a disposición de los clientes que estén funcionando.

[6] Por ejemplo si tenemos MFS montado en */mfs*, entonces */mfs/3/usr/src/linux/* es el directorio */usr/src/linux/* del nodo 3.

[7] `http://www.netapp.com`

[8] *The MOSIX scalable cluster file systems for LINUX* de Lior Amar, Amnon Barak, Ariel Einzenberg y Amnon Shiloh.

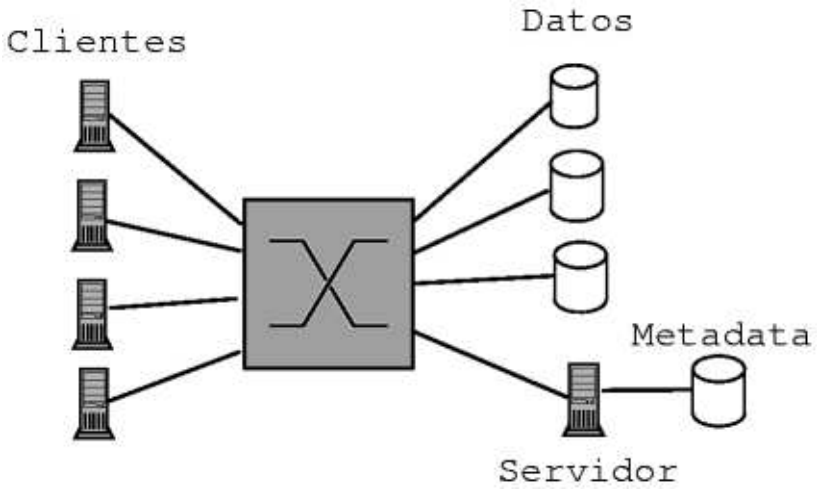



Figura 3.2: Sistemas operativos. GFS con servidor central

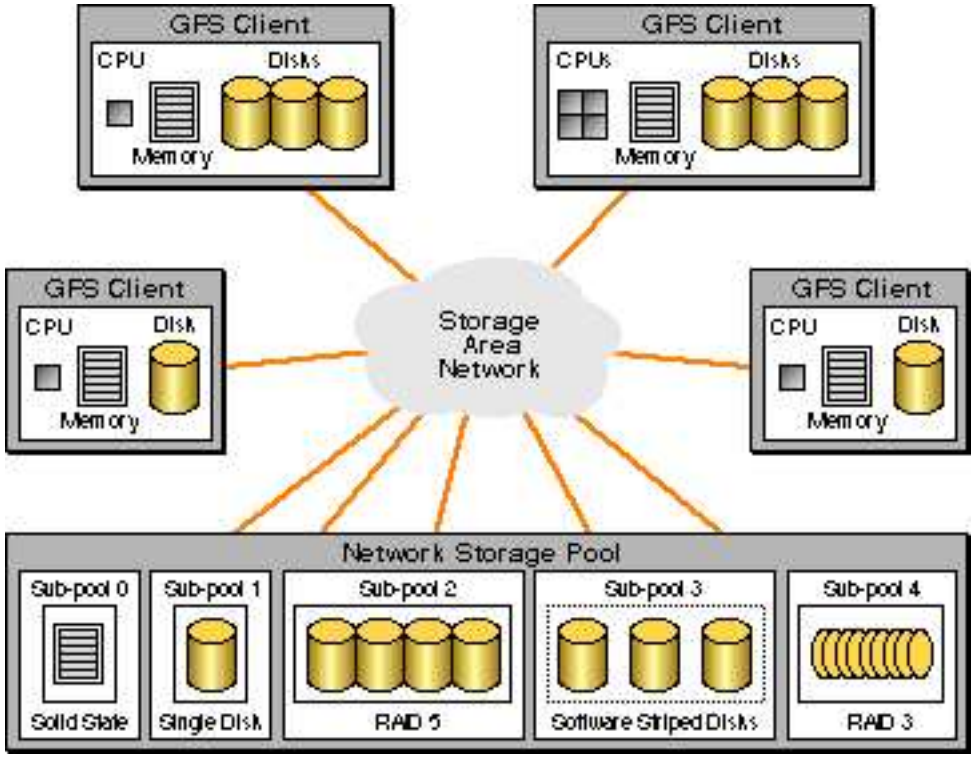



Figura 3.3: Sistemas operativos. SAN

- Cada cliente puede acceder a cualquier parte de los datos (cualquier disco), por lo tanto se facilita la migración de procesos pues ya no hay que tener en cuenta si llevar con el proceso los ficheros abiertos o no y no hay que dejar en el nodo origen información sobre ficheros abiertos.
- Usando técnicas de LVM (*Logic Volume Management*) se pueden unir varios de los discos duros en uno solo, simplificando el control de los dispositivos de almacenamiento y reduciendo la complejidad de la administración.
- Aumenta la escalabilidad en capacidad, conectividad y ancho de banda con respecto a otros sistemas como NFS que están basados en servidores centrales.
- Es un sistema de ficheros *journaled*, un espacio *journaled* para cada cliente para evitar que sea tan ineficiente, además se envía la información por grupos y se intenta mejorar el I/O clustering, y tiene capacidad de una recuperación rápida de los fallos de los clientes, gracias a los distintos espacios *journaled*, sólo los tiene que reparar secuencialmente , no importa que caigan gran número de nodos. No se pierde información.
- Requiere hardware muy caro, fibra óptica, *switch/hub* de alta velocidad, SCSI (normalmente RAID SCSI)

Aunque de todos estos puntos solo el último sea negativo es una razón bastante fuerte y relega al uso de este sistema para empresas con gran presupuesto que no quieran usar un servidor centralizado.

Este sistema de ficheros cumple todas las transparencias explicadas al principio de la lección, en el caso de haber un servidor central este es el que no cumple los criterios de transparencia pero en la parte de los clientes si los cumple pues no saben dónde están los ficheros (transparencia de acceso), se podrían cambiar de disco duro sin problema (transparencia de localización), si un nodo falla el sistema se recupera (transparencia a fallos), varios clientes pueden acceder al mismo fichero (transparencia de concurrencia) y se mantienen caches en los clientes (transparencia de réplica).

### 3.1.5. Entrada salida

En un sistema tradicional, la entrada/salida es local al nodo en el que se produce, pero desde la aparición de las redes se han venido aprovechando éstas para acceder a determinados recursos de entrada/salida colocados en un ordenador distante.

Por ejemplo es típico en las empresas comprar una única impresora cara para obtener la mejor calidad posible y dejar que esa impresora sea accedida desde cualquier ordenador de la intranet de la empresa, aunque esto significa el desplazamiento físico de los empleados. Puede ser un ahorro considerable a instalar una impresora en cada uno de los ordenadores.

El problema es que para este ejemplo se ha desarrollado una solución específica que necesita un demonio escuchando peticiones en un determinado puerto. Desarrollar una solución general es mucho más complejo y quizás incluso no deseable. Para que cualquier nodo pueda acceder a cualquier recurso de entrada/salida, primero se necesita una sincronización que como ya se ha visto en una sección anterior de este capítulo puede llegar a ser complejo. Pero también se necesita conocer los recursos de entrada/salida de los que se dispone, una forma de nombrarlos de forma única a través del cluster, etc.

Para el caso concreto de migración de procesos el acceso a entrada/salida puede evitar que un proceso en concreto migre, o más convenientemente los procesos deberían migrar al nodo donde estén realizando toda su entrada/salida para evitar que todos los datos a los que están accediendo tengan que viajar por la red. Así por ejemplo un proceso en openMosix que esté muy vinculado al hardware de entrada/salida no migrará nunca (Xwindow, lpd, etc.). Los sockets como caso especial de entrada/salida también plantean muchos problemas porque hay servicios que están escuchando un determinado puerto en un determinado ordenador para los que migrar sería catastrófico pues no se encontrarían los servicios disponibles para los ordenadores que accedieran a ese nodo en busca del servicio.

## 3.2. LA IMPORTANCIA DE LA RED

*Victory is the beautiful, bright coloured flower.*
*Transport is the stem without which it could never have blossomed.*
*Winston Churchill*

### 3.2.1. La importancia del sistema de comunicación

En los clusters la eficacia del sistema de comunicación es crítica. Si las comunicaciones fallan el cluster dejará de ser un cluster y se convertirá en un conjunto de máquinas que no cooperarán, algo lejos del objetivo. Por lo tanto es usual disponer en sistemas cluster de alta disponibilidad de una red alternativa por si la red principal fallara. Cabe decir que una red es un elemento bastante fiable a nivel físico: es difícil que una vez instalada y probada, falle. Sobre las **topologías** y **tecnologías** de red que existen se hablará en las próximas secciones.

Otro tema importante de la red es la eficiencia: de nada sirve una red si está congestionada. Hay que tener en cuenta que todas las comunicaciones entre nodos pasan a través de la red; dependiendo de la cantidad de nodos la red puede ser la culpable directa de mermar la eficacia computacional del cluster. Es por esta razón que la inversión en una red tecnológicamente moderna es habitual en estos sistemas.

### 3.2.2. Topologías de red

Existen muchas topologías de red que se han impuesto en diversos entornos, cada una de estas topologías tienen sus propios puntos fuertes.

Las topologías estáticas son topologías donde una vez hechas las conexiones, éstas no cambian. Las redes dinámicas están construidas con elementos que se pueden conectar entre varios caminos, esto hace que si un camino está siendo usado se pueda usar otro, permitiendo más paralelismo. La topología de la red influye mucho en el grado de paralelismo que se pueda alcanzar.

#### Redes estáticas

Las redes estáticas fueron las primeras en aparecer. En estas redes se distingue entre redes punto a punto y redes con acceso a medio compartido. Van a verse los ejemplos más usados, explicando las ventajas y los inconvenientes de cada una de ellos. Entre las redes punto a punto destacan:

- **Lineal.**

  Todos los nodos están conectados de forma lineal; como la comunicación es de punto a punto el peor caso es cuando un nodo de una esquina quiera conectar con un nodo que esté en la otra esquina. En este caso se necesitan `N-1` pasos para llegar hasta el destino, siendo `N` el número de nodos en la red.

- **Anillo.**

  Es similar al caso de la red lineal pero los nodos de las esquinas están unidos entre síÈsto tiene la ventaja que añadiendo un solo enlace más se consigue que el diámetro de la red pase de `N-1` a $\frac{N}{2}$ pues ahora los dos nodos más alejados son los que estén en dos puntos extremos de un anillo con la mitad de los nodos en cada lado hasta llegar hasta ellos.

- **Estrella.**

  Hay un nodo central que se conecta con todos los demás nodos. Todos los nodos sólo se conectan al nodo central. Esto tiene la ventaja de que en 2 pasos se llega desde cualquier nodo a cualquier otro (excepto el central que solo necesita uno). Pero la desventaja que el nodo central tiene que soportar mucha sobrecarga, pues tiene que llevar todas las comunicaciones que están ocurriendo. Además si ese nodo central cayera, la red dejaría de funcionar.

- **Árbol.**

  Cada nodo puede tener unido a él varios nodos hijos y está unido a un nodo padre, menos el nodo raíz y los nodos hoja.

Una variante muy usada es el árbol binario en el que cada nodo tiene como máximo 2 nodos hijos. Esto hace que se puedan hacer varios cálculos (recorridos, ...) de forma más efectiva.

Una mejora de este árbol es el árbol jerárquico que consta de más enlaces según se sube en las ramas del árbol, esto es así porque los padres tienen que llevar a cabo todas las comunicaciones que quieran hacer sus hijos que no tengan como destinatarios a sus hermanos y como a cuanta más altura más hijos tiene cada padre (entendiendo por hijos: hijos, nietos, bisnietos, etc.) hay más comunicaciones de las que el padre se tiene que hacer cargo por lo que se necesitan más posibilidades de comunicación. Los pasos que tiene que seguir un dato de uno de los nodos para que llegar a su nodo más lejano es $2 * log(n + 1) - 2$ .

- **Malla.**

  Los nodos están conectados en una red 2D. Cada uno se conecta con otros cuatro nodos excepto los nodos que están en los bordes de la malla que se conectan a 2 o 3 nodos. El aspecto que tiene esta forma de conectar una red es como un tablero de ajedrez donde los puntos donde se cruzan las líneas son los nodos y las líneas son los medios de comunicación. Los pasos que se tienen que seguir para que un dato llegue de un nodo a su nodo más alejado son $2 * r - 2$ siendo `r` el número de nodos por lado que hay. Suele implementarse en circuitos para soportar multiprocesadores.

- **Sistólica.**

  Muy parecida a la malla 2D pero existen más conexiones entre los elementos: 6 conexiones. Es una red más cara de construir pero que necesita menos pasos para llegar los datos de un nodo a otro nodo.

- **Totalmente conectada.**

  Esta red es la más cara de construir porque cada nodo está conectado a todos los demás. Esto permite que un dato llegue de un nodo a cualquier otro en 1 paso, minimizándose al máximo el tiempo gastado en el traspaso de información. Esta configuración sólo es viable para un número de nodos pequeño.

- **Hipercubo.**

  Es una malla de `N` dimensiones, este número de dimensiones es el grado del hipercubo. En esta malla cada nodo conecta con un número de nodos igual al grado del hipercubo, el número de nodos del que dispone el hipercubo es $2^N$. En esta configuración todos los nodos son iguales, conectan con el mismo número de nodos.

  Aunque es un poco difícil de visualizar con grados mayor que el grado tres, es una configuración bastante usada (Intel hizo en su día una red de este tipo con procesadores 286).

### Redes dinámicas

Estas redes cambian su topología dinámicamente: cuando dos nodos necesitan conectarse la red puede cambiar de tal forma que se puedan conectar, así no se necesita pasar por todos los nodos o crear una complicada estructura de conexión. En el caso anterior los nodos tenían que proveer de las capacidades que necesitaba la red (por ejemplo el nodo central de una red tipo estrella tenía que tener características especiales) . Este es el caso donde la red es controlada por los nodos. La red está formada de cajas de conmutación: cambiando las cajas de conmutación se cambia la red.

Hay dos tipos de redes dinámicas:

- **Redes monoetapa:** sólo necesitan un paso para conectar dos nodos diferentes (una sola etapa).
- **Redes multietapa:** realizan las conexiones entre los nodos en más de un paso.

En las redes monoetapa si no se puede acceder desde cualquier elemento a otro cualquiera se necesita recircular la información.

Dentro de este tipo de redes es de especial interés el caso de la red de barras cruzadas: los procesadores tanto formando una linea en horizontal como en vertical y unas vias los conectan; cuando un procesador se quiere comunicar con otro, la posición (proc X, proc Y) se activa con lo que se hace una conexión.

Esto permite que todos los procesadores puedan mantener conexiones independientes. Mientras que no haya dos procesadores que quieran comunicarse con un mismo procesador no habrá ningún problema. Este esquema

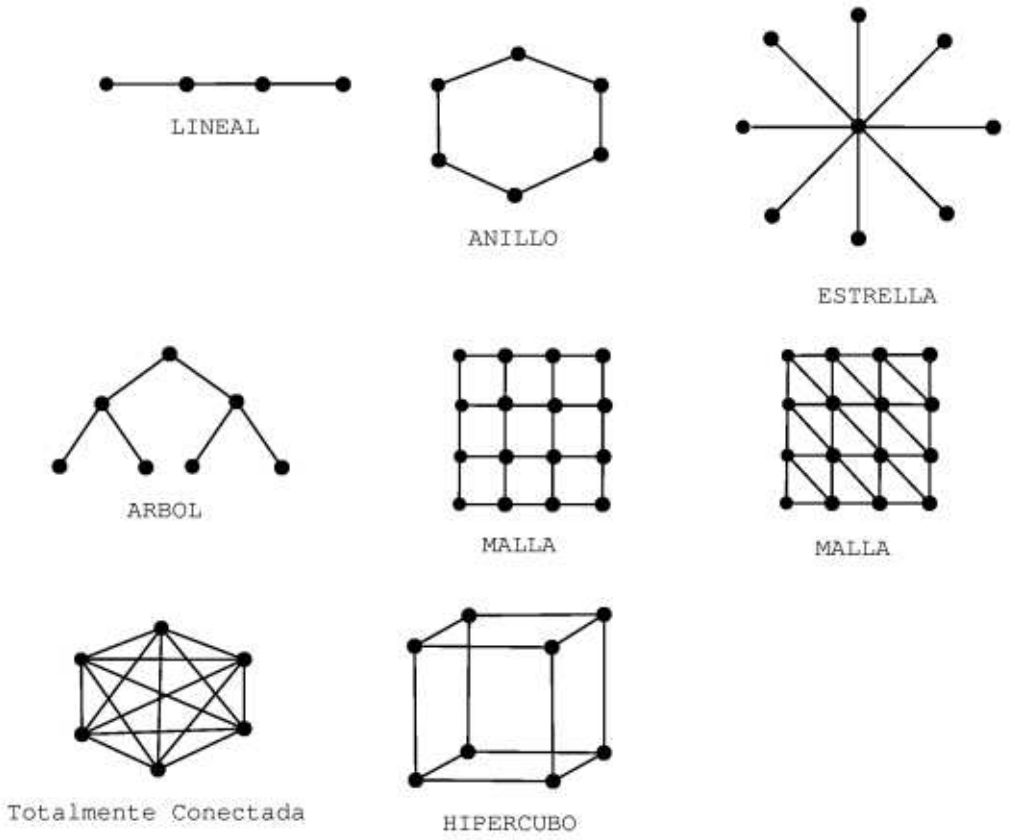



Figura 3.4: La importancia de la red. Topología de redes estáticas

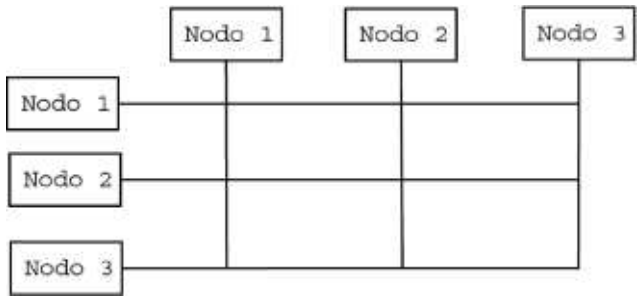



Figura 3.5: La importancia de la red. Barras cruzadas

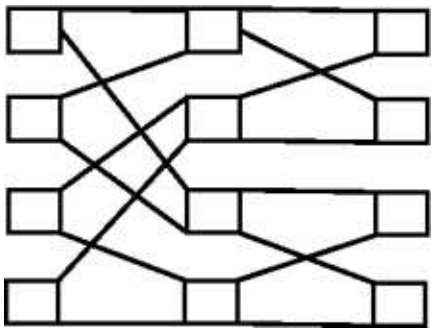


Figura 3.6: La importancia de la red. Red dinámica con bloqueo

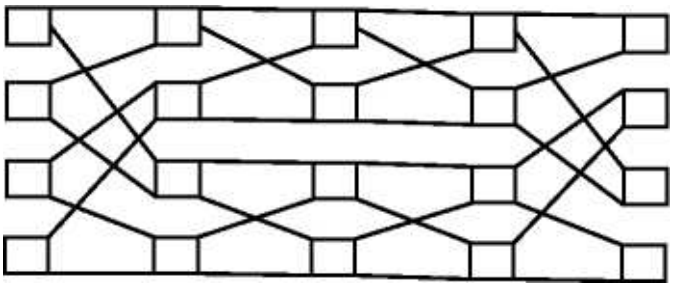


Figura 3.7: La importancia de la red. Red dinámica reordenable

es bastante caro y no merece la pena si no se hacen muchas conexiones simultáneas. Las redes multietapa son más complejas. Para comprenderlas primero tenemos que comprender las cajas de conmutación. Las cajas de conmutación tienen dos entradas y dos salidas. Para un ejemplo las entradas se llamaran `e1` y `e2` y las salidas `s1` y `s2`. Se pueden formar cuatro posibles configuraciones, según se relacionan las entradas con las salidas:

- Paso directo: las dos entradas pasan directamente a las 2 salidas. La entrada `e1` sale por la salida `s1` y la entrada `e2` sale por la entrada `s2`.
- Cruce: las salidas salen en sentido inverso a las entradas. La entrada `e1` se conecta con la salida `s2` y la entrada `e2` se conecta con la salida `s1`.
- Difusión inferior: ésta y la siguiente configuración se usan para hacer broadcast. En esta configuración la entrada `e1` se conecta a la salida `s1` y `s2` y la entrada `e2` se deja sin conectar. Por lo tanto se asegura que por `e2` no está pasando una comunicación para realizar esta operación.
- Difusión superior: similar al caso anterior, pero ahora es la entrada `e2` la que se conecta a las salidas `s1` y `s2` y la entrada `e1` la que no se conecta a nada.

Se pueden dividir las redes multietapa en tres tipos:

- Con bloqueo: en estas redes para llegar de un nodo a otro solo hay un camino, para ahorrar conexiones, gracias a las cajas de conmutación varios caminos van a una caja por lo que puede ocurrir que la conexión simultánea de varios nodos produzca conflictos. Tras realizar una asociación entre dos nodos, intentar hacer otra asociación puede bloquear. En este ejemplo si los dos primeros nodos quieren conectar con el primer nodo, tendrán que esperar porque los dos necesitan el mismo enlace.
- Reordenables: existen varios caminos para llegar al mismo lugar, siempre se puede conectar otro camino, pero quizás habría que reordenar la situación. Estas redes son más complejas pero aunque se necesite reordenación, podemos conectar simultáneamente los nodos. Además estas redes se pueden construir de un tamaño arbitrario usando el método de red de benes, solo hay que unir redes de este tipo de potencia de 2 para conseguirlo.

- Sin bloqueo: estas redes pueden manejar todas las conexiones posibles sin que se produzcan bloqueos, para conseguir esto, todas las cajas de conmutación de un nivel (una columna) se conectan a todas las cajas de conmutación de la siguiente columna.

### 3.2.3. Tecnologías de red

Van a verse las tecnologías de red más usadas en el clustering. Estas tecnologías son las mismas que imperan en el mundo de las redes locales de medio y alto rendimiento y son:

- Serie
- Ethernet
- Fast Ethernet
- Gigabit Ethernet
- ATM

Por supuesto estas no son las únicas tecnologías existentes, pero si que son las más comunes. Las comunicaciones a través de Token Ring o sus derivados quedan fuera de este tema.

- Serie: puede ser un poco sorprendente ver este método de comunicación aquí pero al menos una aplicación muy importante puede usarlo por lo que puede ser un buen método de comunicación si se necesita una forma sencilla de comunicarse con otro nodo.

  La aplicación que usa este método es Heartbeat del que se habla con más detalle en el capítulo de clusters HA. La comunicación serie es perfecta para esta aplicación pues sus pequeños pulsos no necesitan gran ancho de banda pero en un medio compartido como Ethernet pueden ocasionar colisiones que afecten en gran medida al rendimiento de otras comunicaciones que se lleven a cabo en el cluster.

- Ethernet: la velocidad máxima teórica es de 10 Mb/segundo y aunque hace pocos años era un estandar y estaba muy extendido el uso de esta tecnología en redes locales, la caída de los precios de los equipos de 100 Mb/segundo (Fast Ethernet) ha hecho que esta tecnología quede desfasada.

  Ethernet está especificado en el estandard IEEE 802.3, es *half duplex*. Existe un medio compartido que físicamente puede ser topología tipo bus (por ejemplo de cable coaxial) o en topología tipo estrella con un *hub* o un *switch* como elemento central. El acceso al medio compartido se gestiona con CSMA/CD (*Carrier Sense Medium Access/Colision Detection*) Esto quiere decir que la tarjeta de red antes de enviar la información *escucha la linea* mirando el voltaje de la línea sabe si hay datos o no en ella.

  Si no hay datos, envía sus datos y se pone a escuchar la línea, si el voltaje es superior a un determinado umbral, más de una tarjeta está intentando enviar datos a la línea a la vez. En este caso, se considera que los datos que se enviaron se enviaron con errores y se vuelve a intentar enviarlos tras un tiempo aleatorio ($2^i - 1$, siendo $i$ el número de intento). Este método de acceso al medio impone ciertas condiciones a las redes Ethernet. Por ejemplo tiene que haber un tamaño mínimo de paquete (46 bytes) y una longitud máxima de segmento.

  Es de especial mención la gran diferencia que existe entre usar un *hub* y un *switch*. El *hub* simplemente conecta todos los cables que le entran con todos, sirve pues para tener una topología física de estrella. A su vez pero se obtiene una topología lógica de bus; esto tiene la ventaja de que si se corta un cable sólo se incomunica un ordenador, no todos. El *switch* en cambio, a parte de tener la misma función que el *hub*, sólo conecta los cables de los ordenadores que se están comunicando.

  Hay 2 tipos de *switch*:

  - *Cut-through* (al vuelo): en cuanto lee la dirección de destino del paquete lo reenvía por el cable adecuado, así varios ordenadores pueden estar comunicándose simultáneamente siempre que las parejas de ordenadores que se comunican sean distintas.
  - Almacenamiento y reenvío: similar al anterior pero no envía un paquete hasta que lo tiene completo. Esto evita algunas colisiones y elimina los paquetes erróneos lo antes posible, además aporta mejora en la gestión de la red, como conteo de paquetes, filtros y otras opciones que dependen de cada fabricante en cuestión, el problema es que la latencia de la red aumenta.

- *Fast Ethernet*: Utiliza la misma técnica que Ethernet pero con velocidades de 100 Mb/segundo, puede usar el cableado existente por lo que los cambios se reducen a las tarjetas de red (hoy en día las mismas tarjetas de 10 Mb/segundos también tienen soporte para esta tecnología) y a los concentradores (en los que sí que hay que hacer mayores inversiones pues son bastante más caros). Existen concentradores con puertos 10/100 para facilitar la migración a esta tecnología.

  Como la trama mínima sigue del mismo tamaño, el tamaño máximo de estas redes tiene que ser 10 veces inferior para poder detectar cualquier colisión. Está normalizado en el 802.3u.

- *Gigabit Ethernet*: esta es la tecnología Ethernet de más reciente aparición, que consigue velocidades de 1 Gbps o superiores y es compatible con las dos tecnologías anteriores. Se disminuye la distancia para detectar colisiones. Se puede usar cables de categoría 5 pero para grandes distancias se recomienda el uso de fibra óptica. Cuando lo que se necesita es una red LAN de altísimas prestaciones (como la que puede necesitar un cluster) y se dispone del dinero necesario, la elección está entre esta tecnología y ATM.

- ATM: se basa en una tecnología de conmutación de celdas. Las celdas son de un tamaño fijo (53 bytes, 5B cabecera + 48B datos), esto simplifica en gran medida el procesamiento de las celdas, acelerando la conmutación y disminuyendo el retardo. Da muchos servicios de valor añadido de los que no dispone Ethernet, por ejemplo permite que la velocidad de transferencia sea fijada dinámicamente bajo demanda.

  No sólo eso, sino que puede garantizar otros factores, como el retardo, por lo que la gran ventaja es el QoS (calidad de servicio) que de forma inherente proporciona. Es una red orientada a conexión y tiene un control del tráfico y de la congestión muy eficiente. Los problemas que plantea frente a otras soluciones es que es una alternativa muy cara: la tecnología es compleja y aún no existen muchas aplicaciones desarrolladas.

### 3.2.4. Protocolos utilizados a nivel de red

Otro tema importante aparte de las topologías o tecnologías de red utilizadas son los protocolos que se adaptan mejor a sistemas cluster. Generalmente la tecnología de red de cluster más utilizada suele ser ATM o Ethernet y dado el precio de los dispositivos ATM suele ser normalmente la segunda.

En cualquier caso, el protocolo más extendido en el uso de las redes de cualquier tipo actualmente es IP (Internet Protocol). IP permite trabajar sobre cualquier tecnología de red, ya que es independiente del medio de comunicación físico. IP es un protocolo no orientado a conexión y no fiable que se basa en técnicas de *el mejor esfuerzo* (best effort) para encaminar sus paquetes. Esto es en cierto modo una ventaja, en el caso de que se utilice sobre redes no conectivas como pueden ser Ethernet, o una desventaja en redes ATM. En cualquier caso la adaptabilidad del protocolo IP a cualquier red mediante capas que actúan de interfaz permite situar este protocolo por encima de ATM, Frame Relay, Ethernet o redes LAN en general e incluso sobre líneas serie o paralelo.

El protocolo IP se encarga de cumplir los siguientes objetivos en una comunicación:

- Definir el formato de bloques de datos que será enviado, para evitar errores de cabecera o tipo de archivos.
- Esquema de direccionamiento.
- Movimiento de datos entre nivel de transporte y nivel de red, dependiente de cada implementación.
- Encaminamiento de los datagramas.
- Fragmentación y agrupación de los datagramas en el caso de que sea necesario al pasar el paquete por redes de MTU más pequeña.

Es un protocolo ampliamente utilizado y con muchos años de uso. La práctica totalidad de los sistemas que se conectan a redes tienen alguna implementación IP ya sea completa o incompleta, lo que nos permite tener redes de equipos hetereogéneos. IP no es sólo un protocolo aislado, sino que trabaja en conjunto con otros protocolos como TCP para realizar transmisiones. Estos protocolos son ICMP e IGMP en el nivel de red[9] y TCP y UDP en el nivel de transporte.

El protocolo ICMP (Internet Control Message Protocol) se encarga de mandar mensajes de control: errores en host, fallos de encaminamiento y mensajes similares, entre los host de la red. IGMP (Internet Group Message Protocol) se encarga de mantener actualizada la información relativa a la creación de grupos para la multidifusión,

[9] Aunque estos esten en el nivel de red, son encapsulados en datagramas IP

broadcast o multicast. El conjunto de estos protocolos dotan a la solución de una completa solidez en uso y funcionamiento.

El formato de bloques de IP no interviene en el factor de diseño de clusters, además, en el caso de IP, permite reducir bastante el tamaño de los paquetes que hay que enviar o ampliarlos al máximo de lo que nos permita la red. El esquema de direccionamiento tampoco es un problema para los clusters, aunque empieza a serlo a nivel mundial por la escasez de direcciones. Se espera que en poco tiempo la red vaya migrando a IPv6, en cualquier caso desde el punto de vista particular de los clusters el uso de IPv6 o IPv4 no tiene por que ser drástico e incluso puede favorecernos esta migración por hacer que los paquetes sean más configurables.

Respecto al encaminamiento y la fragmentación de los datagramas, permiten tener clusters diseminados por una geografía más extensa. Generalmente los clusters suelen localizarse en un mismo aula, edificio o entorno, pero existen problemas para los que pueden ser necesarias redes de área extensa para los que IP ya esta ampliamente comprobado y utilizado, como por ejemplo organizaciones internacionales o empresas multinacionales.

Este documento no pretende ser muy duro en cuanto a la teoría de redes. Lo básico para configurar la necesaria para interconectar nuestros nodos. Lo que sí es necesario es dar una explicación de porqué utilizar IP en clusters.

- Ampliamente utilizado, conocido y comprobado.
- Efectividad en la mayoría de las redes probadas.
- Perfecto para ambientes hetereogéneos.

El principal motivo para utilizar IP es que en la mayoría de los casos y a pesar de que pueda consumir en muchos casos demasiado ancho de banda o capacidad de proceso, es mejor utilizar un protocolo comprobado y válido que uno a desarrollar para un sistema concreto.

Existen otros protocolos de red que se han utilizado para entornos distribuidos como pueden ser IPX/SPX, Appletalk u otros; e incluso algunas soluciones se han basado directamente sobre capas de nivel de enlace para efectuar las comunicaciones entre los nodos. GNU/Linux soporta sin problemas todos estos protocolos.

### 3.2.5. Protocolos utilizados a nivel de transporte (UDP/TCP)

**Protocolos de transporte**

Los protocolos de red están divididos por capas. En la subsección anterior se ha visto la capa de red, ahora se dará paso a las capas que existen a nivel de transporte: UDP y TCP.

La mayoría de los protocolos de transporte tienen elementos comunes. Los protocolos orientados a conexión deben tener mecanismos para establecer y liberar conexiones. Cuando hay múltiples conexiones abiertas en una misma máquina, las entidades de transporte tendrán que asignar un número a cada conexión y ponerlo en el mensaje.

En sistemas operativos UNIX y para la red ARPANET existen dos enfoques para saber a que puerto acceder a un servicio:

- Un servidor de puertos escuchando en el UDP/TCP 111 atendiendo solicitudes, los servidores registran el nombre del servicio y el puerto en donde está escuchando, permaneciendo dormidos hasta que les llega una solicitud.
- Los puertos los conocen los clientes que tienen el fichero */etc/services*; hay un servidor (*inetd* o *xinetd*) que escucha en todos los puertos que se quieran mantener abiertos, cuando llega una solicitud crea un nuevo proceso que se encarga de atender la solicitud. Al cliente le da la impresión de que el servicio está siempre activo.

Otras de las características comunes de todos estos protocolos de nivel de red y que hacen este nivel tan útil es el control de flujo. Si por debajo del protocolo de red hay una red fiable (X.21) o nuestro protocolo no añade fiabilidad a la red (UDP) entonces no será necesario usar el control de flujo de la capa de transporte.

En otro caso el control de flujo es fundamental. El emisor tiene una copia local de cada mensaje que se envía por la red hasta el momento que es asentido, el receptor quizás no desee enviar un ACK (*acknowledgement* o asentimiento) de ese mensaje, o no recibió el mensaje correctamente; en esos casos el ordenador que está enviando los mensajes tiene que reenviar el mensaje, por ello la necesidad de tener una copia local.

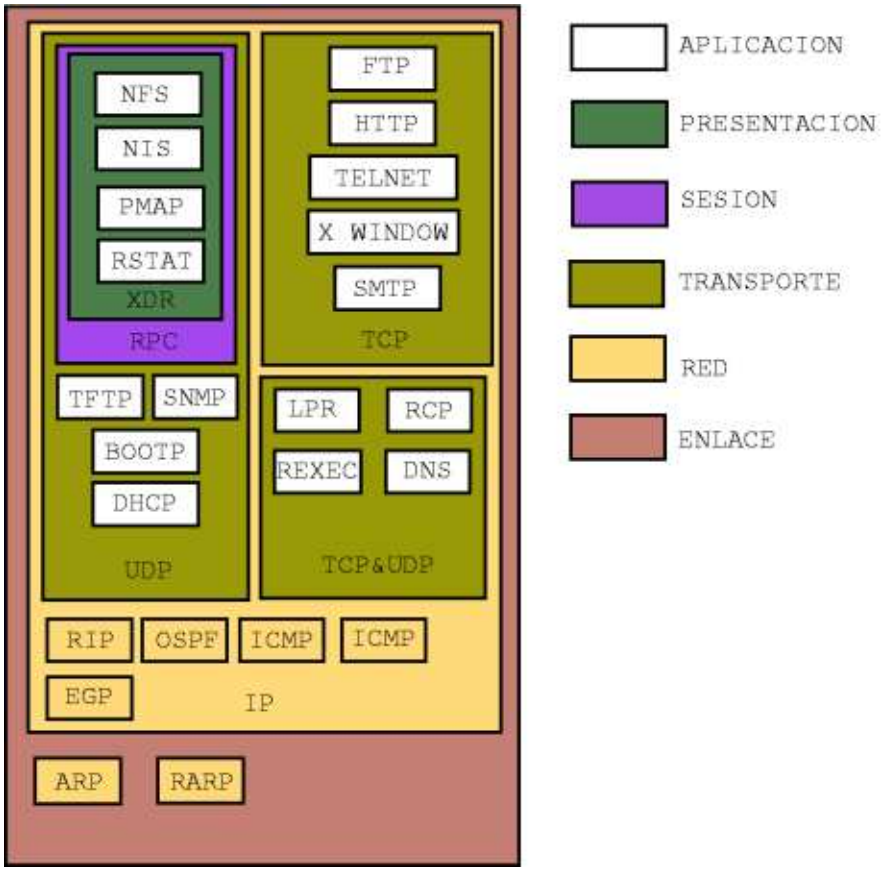



Figura 3.8: La importancia de la red. Encapsulamiento IP

Con este sistema el receptor puede limitar el tráfico que recibe simplemente tardando más tiempo en asentir los datos recibidos. Así cada uno de estos protocolos define unos temporizadores y unas reglas para no sólo controlar el flujo sino controlar que no se produzca congestión en la red.

Otro punto en común de todos los protocolos de transporte es que permiten la multiplexión y demultiplexión de las conexiones, para aumentar el ancho de banda y aprovechar mejor los recursos de los que se disponga.

### UDP

Es el protocolo de transporte no fiable estándar en Internet. Como protocolo de transporte que es solamente está implementado en ordenadores finales, no en routers. Es un protocolo no orientado a conexión (de ahísu nombre, se envían datagramas) y no es fiable. La cabecera que incluye al mensaje es la mínima para poder distinguir a que puerto se dirige y un pequeño *checksum* de la cabecera. Es usado por las aplicaciones de los servicios no conectivos, para enviar pocos datos (se consigue más rendimiento puesto que no se necesita gastar tiempo en conectar y desconectar) y en multidifusión.

UDP al contrario de TCP no puede realizar fragmentaciones de los mensajes puesto que no tiene campos para ello. Por tanto es la aplicación quien debe dividir los mensajes como crea oportuno. Como la tecnología física de red también impone una restricción de tamaño máximo de paquete, otra división se realiza en el nivel IP; por tanto una aplicación que conociera la MTU evitaría esta segunda división. Cuando uno de los fragmentos de un mensaje UDP se pierde se tendría que retransmitir.

En el caso de UDP será la aplicación la encargada, gracias a unos temporizadores, de retransmitirlo. En el caso de TCP es él mismo el que detecta y soluciona e problema.

Como sabemos IP si no puede recomponer un datagrama, lo desecha, no tratando el problema y dejando la solución a capas superiores.

Algunos capas superiores que usan UDP son RPC y todos los aplicaciones que dependen de estos (DNS, NIS, etc.) y otras aplicaciones como pueden ser TFTP, BOOTP, DHCP.

### TCP

Este protocolo junto con UDP son los dos protocolos de transporte estándar en Internet, es el protocolo fiable de la familia. Es un protocolo orientado a conexión. TCP recibe mensajes de cualquier tamaño y los divide en trozos de un máximo de 64KB se envía cada uno de estos trozos como un mensaje TCP separado. Para evitar fragmentaciones extra, TCP conoce la MTU de la red, con lo que se evita que el nivel de red tenga que dividir a su vez para acoplarse al MTU de la red.

También es misión de TCP ordenar todos los segmentos que hayan llegado separados y ensamblarlos para obtener el mensaje inicial. Tiene la ventaja de que las aplicaciones no tienen que preocuparse de tratar los errores de la comunicación puesto que para la aplicación el canal de comunicación es perfecto y siempre llega la información integra.

El que sea un servicio orientado a conexión quiere decir que hay 3 fases en la comunicación, que forman un mínimo de 7 mensajes sin contar los mensajes de datos:

- Establecimiento de la conexión: es la fase inicial, para conectarse los paquetes tienen un bit especial llamado SYN. Para conectarse se debe enviar un segmento con SYN activado y recibir una confirmación ACK. Se toman unos nuevos números de secuencia para enviar y para recibir que empiezan siendo un número muy superior al último número de secuencia recibido por si llegan segmentos de la conexión anterior y se empieza la conexión, esto son 3 mensajes.

- Traspaso de información: típicamente aquíse envían todos los datos que se quieren enviar, el nivel de aplicación verá como todos los datos que envía, llegan correctamente (si es físicamente posible) y en orden.

- Liberación de la conexión: se realiza gracias a un bit que indica que se desea finalizar con la conexión actual llamado FIN. Cuando un extremo activa su bit FIN, su interlocutor tiene que hacer un ACK y hasta que no lo desee puede posponer el cierre de la comunicación, cuando esté listo envía un paquete con el bit FIN activado y el primer ordenador realiza un ACK, liberando la conexión.

TCP cada vez que envía un segmento inicia un temporizador. Si este temporizador cumple y no se ha recibido confirmación se vuelve a enviar el mismo segmento suponiendo que el anterior segmento enviado no llegó o

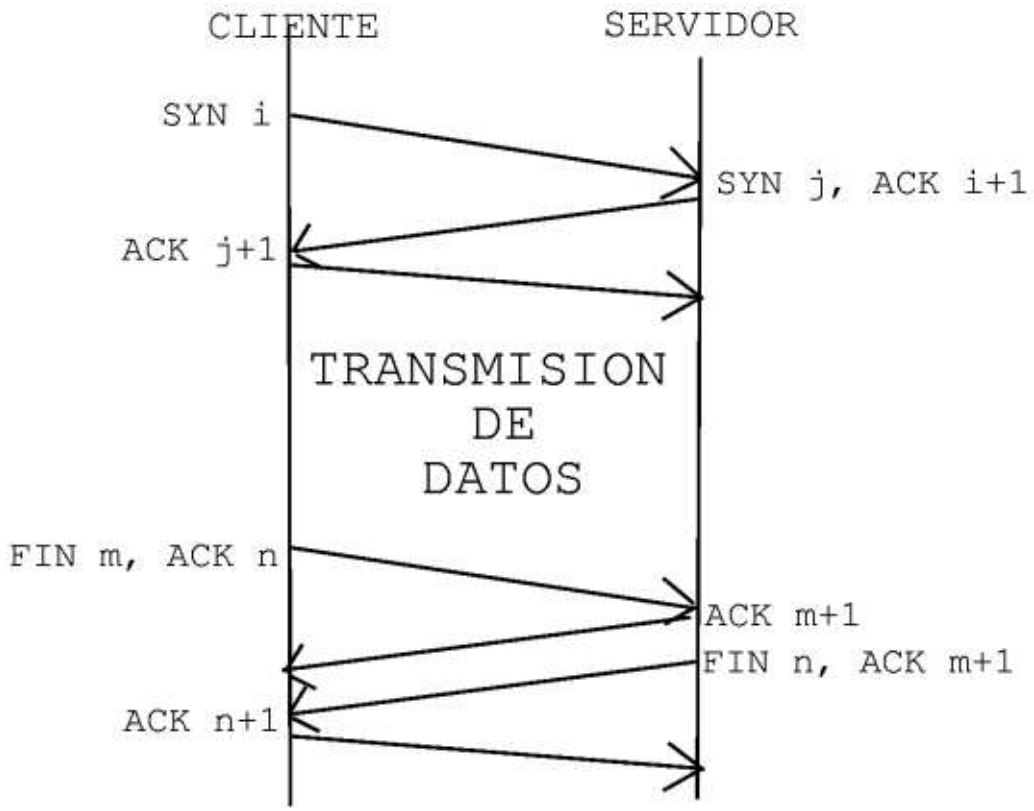



Figura 3.9: La importancia de la red. Conexión TCP

llegó con errores. Como no se puede saber a ciencia cierta cuánto tardará un segmento a llegar a su destino, pues dependiendo del paquete podría pasar por una redes u otras, haber congestiones momentáneas, etc. se implementa un temporizador variable que irá cambiando según lo que tardaron los ACK de los anteriores segmentos.

La peor situación que le puede ocurrir a una red físicamente bien es que tenga congestión, por eso esta situación se intenta evitar a toda costa, y los protocolos orientados a conexión son mejores para evitarla, por eso TCP intenta hacer un control de congestión en Internet. Quien más rápidamente puede reaccionar ante la congestión es el emisor, por tanto cuando según acabamos de ver se cumple el temporizador de retransmisión hay error o hay congestión, como las redes actuales son de muy buena calidad es muy difícil que el error sea a causa del medio de transmisión por lo tanto el emisor supone que existe congestión con lo que envía los datos de forma más lenta, siguiendo un algoritmo, con esto se intenta que se alivie la congestión de la red.

Para controlar el flujo y la congestión se usa la ventana deslizante que permite enviar varios paquetes antes de que se envíe un ACK, enviar un ACK por cada paquete es realmente lento por lo tanto cuando se quiere disminuir la velocidad por el peligro de congestión o porque el receptor no puede recibir tantos datos la ventana disminuye dejando enviar menos paquetes por ACK, pudiendo llegar a ser 0, lo que significa que el receptor no puede en esos momentos recibir nada más. Cuando se quiere más velocidad se deja enviar más paquetes antes de parar para esperar un ACK.

### 3.2.6. Diseño de redes

A la hora de poner en marcha un cluster ya sea a nivel de producción como a nivel de investigación, hemos de tener en cuenta que uno de los factores críticos a la hora de configurar dicho sistema va a ser la red.

Antes de comenzar a instalar el sistema propiamente dicho, se debe hacer un estudio de que servicios o programas debe correr, y tener en cuenta cómo éstas van a afectar a la carga en la red. Este apartado es bastante importante en la instalación de redes por los siguientes motivos.

1. El coste de instalacíon puede llegar a ser insignificante respecto al coste que puede llevar una ampliación en un sistema que ha sido mal diseñado desde un principio, tanto a nivel físico en la instalación (en la cual

no entraremos) como a nivel de capacidad de la red.

2. La mayoría de los clusters están montados sobre redes LAN, con protocolos UDP/IP o TCP/IP, generalmente no tienen ninguna manera de controlar la congestión de la red, lo que implica que la red debe estar lo suficientemente bien diseñada y montada como para que se produzca congestión sólo en casos excepcionales.

3. Se debe tener en cuenta, que cuanto más acoplado sea un sistema en su funcionamiento, más crítica se vuelve la comunicación, por tanto mejor diseñada debe estar la red de comunicación.

4. El coste que se produce al escalar un cluster, tanto económicamente como en tiempo no debe estar muy ligado al coste de escalar la red.

Como se puede ver, a lo largo de todo el proceso el funcionamiento de la red es crítico en cualquier sistema de tipo cluster, es más, no sólo es necesario un entorno de red adecuado para el sistema, sino que este debe estar en cierta manera infrautilizado para poder asegurar que no se va a llegar a condiciones de congestionamiento.

**Factores a tener en cuenta al diseñar una red**

A la hora de diseñar una red tenemos que tener claros ciertos conceptos acerca del sistema que queremos instalar, el funcionamiento que éste debe tener, los servicios que debe tener y sobre todo, la funcionalidad que se le va a dar.

Hay que conocer la funcionalidad del sistema:

- Funcionalidad del sistema
  - Funcionalidad general para la que el sistema debe funcionar.
  - Servicios que deberá correr el sistema.
  - Factores que externos que pueden afectar a nuestro sistema.

Todos estos puntos quizá entren en el ámbito de la administración del sistema, pero son necesarios en el caso de partir de cero, con la primera instalación de un cluster. Lo que pretenden es que no haya problemas a la hora de tener claro que servicios o funcionalidades están por encima de que otras.

No existe ningún sistema cluster de carácter general, la mayoría de ellos se compone de varias aplicaciones trabajando juntas. Si estas aplicaciones deben acceder a recursos compartidos, hay que tener claras dos cosas: cuáles tienen más prioridad, y cómo obtener la prioridad. Por otro lado están los factores técnicos que afectan a los servicios que utilizará la red. Una vez que se sabe los servicios que poseerá el cluster es necesario describir el tipo de funcionamiento que tiene cada servicio y demás características.

- Factores técnicos de cada servicio relativos a la red.
  - Funcionalidad del servicio.
  - Prioridad del servicio en el sistema.
  - Rrioridad del servicio en la red.
  - Factor de carga del sistema.
  - Factor de carga de la red.
  - Características de la comunicación que realiza el servicio.
    - Tiempos de latencia
    - Tipo de protocolos UDP, TCP.
    - Carga sobre la red a nivel general.

Todos los factores técnicos deben estar referidos a cada servicio particular. Para usuarios expertos puede ser obvio que programas o servicios acaparan más la red o deben tener más prioridad, para alguien profano a ese servicio, o parte del sistema deberá rellenar solamente los factores teóricos que el crea oportunos.

A partir de este apartado, el diseñador de la red debe conocer cada uno de los servicios que la red va a utilizar tanto con conocimiento teórico como práctico para poder tener una visión generalizada. Esta visión debe ser tanto

de como puede afectar este servicio a otros y al sistema en general como para tener los conocimientos necesarios para instalar el servicio sin que este provoque problemas de ningún tipo. Cuando un diseñador inexperto no haya instalado un sistema o un servicio de este tipo, lo conveniente es hacer uso de un entorno de simulación reducido en el cual hacer las instalaciones pruebas y verificaciones para después proceder a realizar extrapolaciones a un sistema grande, se ha de recordar siempre que la depuración de un error en un sistema distribuido crece exponencialmente con el número de nodos que tiene el sistema, y en el caso de que éste utilice varios servicios a la vez, el crecimiento de la complejidad crece más aún a medida que escala el cluster.

Cuando intervienen varios sistemas, todos ellos complejos y completos en un entorno distribuido, se debe tener el suficiente conocimiento de cada uno de los sistemas a montar como para asegurar que el conjunto de todos ellos no desestabilizara el funcionamiento de los otros.

El siguiente punto a tener en cuenta es hacer aproximaciones a las necesidades de red que se han extrapolado de cada servicio, y saber de que recursos se cuentan para satisfacer estas necesidades. En este punto, se debe exigir un equilibrio entre los recursos que se tienen y la efectividad de dicho funcionamiento.

Por ejemplo, en un entorno de trabajo con openMosix, NFSROOT y XWindow en un laboratorio de investigación científica probablemente se dé más importancia a la red openMosix que a el sistema de archivos NFS o XWindow. Así que probablemente se separen dichas dos redes como redes aisladas de manera que NFSROOT o X Window no interfieran sobre openMosix y al mismo tiempo openMosix esté aislado: sus paquetes de información de estado de cada nodo no interrumpirán las comunicaciones largas debido a colisiones cuando openMosix está en un estado sin migraciones.

- Localizar recursos de los que se dispone o saber con que medios se cuentan.
  - equipos o nodos
  - *switches/hubs*
  - routers
  - otros servicios como impresoras en red y demás que puedan afectar la carga de la red
- Estimar la carga de cada red que puede llegar cada servicio contando con los recursos que se poseen[10].
- Equilibrar la carga dependiendo de:
  - prioridad
  - servicio
  - recurso

En este caso no cuenta tanto el conocimiento del sistema, una vez que se conoce como puede llegar a cargar un servicio y de los recursos que se posee, el diseñador, coloca cada servicio en redes aisladas o interconectadas, intentando que interfieran cuanto menos mejor los servicios entre sí, en este apartado, se evalúan las tecnologías de red aplicadas a cada servicio asícomo de los medios de que se dispone para asegurar el funcionamiento y disponibilidad de los servicios. Por ejemplo, en el caso de Heartbeat, se suele utilizar como ya hemos visto un cable de serie de modem nulo, para asegurar que se produce una doble vía de comunicación entre los dos ordenadores que se desea replicar o controlar, e incluso el propio Heartbeat se encarga de comprobar la vía secundaria periódicamente para asegurar que esta estará disponible en el caso de que el sistema principal caiga.

### 3.2.7. Conclusiones

Como conclusiones de este tema hay que el sistema de comunicación es crítico en cualquier sistema paralelo, ya que al tener varios elementos de proceso (entiéndase por elementos de proceso a los procesadores, nodos, procesos, módulos, etc. que componen el sistema paralelo o distribuido), es necesaria la comunicación entre los elementos de proceso para realizar un trabajo conjunto.

En el caso de sistemas implementados en el kernel de un sistema operativo el fallo del sistema puede provocar el malfuncionamiento de una máquina y por tanto se puede requerir del reseteo de la misma.

Por último se han dado los puntos necesarios para diseñar una red correctamente, ya que generalmente cuesta más escalar un sistema mal diseñado con fallos respecto a los requerimientos iniciales que un sistema que sea bien diseñado y quizá sobredimensionado desde el principio.

[10] A la hora de estimar dicha carga, es necesario que ésta sea siempre estimada a la alza, de manera que en el momento de la puesta a punto del sistema, no sea necesario escalar éste inmediatamente después de instalarlo, lo cual sería síntoma de haber diseñado mal el sistema.

# Capítulo 4

# Clusters

## 4.1. CLUSTERS. NOCIONES GENERALES

*It is a capital mistake to theorize*
*before one has data.*

*Sir Arthur Conan Doyle*

### 4.1.1. El concepto de cluster

Aunque parezca sencillo de responder no lo es en absoluto. Podría incluirse alguna definición de algún libro, pero el problema es que ni los expertos en clusters ni la gente que los implementa se ponen de acuerdo en qué es aquello en lo que trabajan.

Un cluster podemos entenderlo como:

**Un conjunto de máquinas unidas por una red de comunicación trabajando por un servicio conjunto. Según el servicio puede ser dar alta disponibilidad, alto rendimiento, etc...**

Por supuesto esta definición no es estricta, de hecho uno de los problemas que tiene es que es demasiado vaga porque por ejemplo dos consolas de videojuegos conectadas para jugar en red ¿se considera cluster? Pero si en vez de estar jugando se está usando el kit de GNU/Linux haciendo procesamiento paralelo ¿entonces se podría considerar cluster?

Realmente el cambio de ambiente es mínimo, desde luego a nadie se le ocurriría definir cluster en base al contenido de los programas que se ejecuten y de hecho es posible que los juegos tengan más capacidades de procesamiento distribuido que los propios programas, entonces ¿qué es un cluster?

Hay definiciones que distinguen entre cluster de máquinas SMP y clusters formados por nodos monoprocesadores. Hay arquitecturas clusters que se denominan *constelaciones*[1] y se caracterizan por que cada nodo contiene más procesadores que el número de nodos. A pesar de todo, las constelaciones siguen siendo clusters de componentes o nodos aventajados y caros. De hecho entre las máquinas que aparecen en el top500 existen unos pocos clusters que pertenecen a organizaciones que no son gigantes de la informática, lo cual indica el precio que pueden llegar a tener estos sistemas.

Por poner unos ejemplos de la disparidad de opiniones que existen, se adjuntan las definiciones que dan ciertas autoridades de esta materia:

*Un cluster consiste en un conjunto de máquinas y un servidor de cluster dedicado, para realizar los relativamente infrecuentes accesos a los recursos de otros procesos, se accede al servidor de cluster de cada grupo* del libro *Operating System Concepts* de Silberschatz Galvin.

*Un cluster es la variación de bajo precio de un multiprocesador masivamente paralelo (miles de procesadores, memoria distribuida, red de baja latencia), con las siguientes diferencias: cada nodo es una máquina quizás sin algo del hardware (monitor, teclado, mouse, etc.), el nodo podría ser SMP. Los nodos se conectan por una red de bajo precio como Ethernet o ATM aunque en clusters comerciales se pueden usar tecnologías de red propias. El interfaz de red no está muy acoplado al bus I/O. Todos los nodos tienen disco local.*
*Cada nodo tiene un sistema operativo UNIX con una capa de software para soportar todas las características del cluster* del libro *Scalable Parallel Computing* de Kai Hwang y Khiwei Xu.

*Es una clase de arquitectura de computador paralelo que se basa en unir máquinas independientes cooperativas integradas por medio de redes de interconexión para proveer un sistema coordinado, capaz de procesar una carga* del autor Dr. Thomas Sterling.

### 4.1.2. Características de un cluster

Si no hay acuerdo sobre lo que es un cluster poco podrá acertarse en sus características. En este apartado se explican los requisitos que deben cumplir un conjunto de computadoras para ser consideradas cluster, tal y como

[1] Aquí hemos dado con el origen del logotipo de openMosix.

se conocen hasta el momento.

Para crear un cluster se necesitan al menos dos nodos. Una de las características principales de estas arquitecturas es que exista un medio de comunicación (red) donde los procesos puedan migrar para computarse en diferentes estaciones paralelamente. Un solo nodo no cumple este requerimiento por su condición de aislamiento para poder compartir información. Las arqutecturas con varios procesadores en placa tampoco son consideradas clusters, bien sean máquinas SMP o mainframes, debido a que el bus de comunicación no suele ser de red, sino interno.

Por esta razón se deduce la primera característica de un cluster:

***i.*- Un cluster consta de 2 o más nodos.**

Los nodos necesitan estar conectados para llevar a cabo su misión. Por tanto:

***ii.*- Los nodos de un cluster están conectados entre sí por al menos un canal de comunicación.**

Por ahora se ha referenciado a las características físicas de un cluster, que son las características sobre las que más consenso hay. Pero existen más problemas sobre las características del programario de control que se ejecuta, pues es el software el que finalmente dotará al conjunto de máquinas de capacidad para migrar procesos, balancear la carga en cada nodo, etc.

***y iii.*- Los clusters necesitan software de control especializado.**

El problema también se plantea por los distintos tipos de clusters, cada uno de ellos requiere un modelado y diseño del software distinto.

Como es obvio las características del cluster son completamente dependientes del software, por lo que no se tratarán las funcionalidades del software sino el modelo general de software que compone un cluster.

Para empezar, parte de este software se debe dedicar a la comunicación entre los nodos. Existen varios tipos de software que pueden conformar un cluster:

- Software a nivel de aplicación.

  Este tipo de software se sitúa a nivel de aplicacion, se utilizan generalmente bibliotecas de carácter general que permiten la abstracción de un nodo a un sistema conjunto, permitiendo crear aplicaciones en un entorno distribuido de manera lo más abstracta posible. Este tipo de software suele generar elementos de proceso del tipo rutinas, procesos o tareas, que se ejecutan en cada nodo del cluster y se comunican entre sí a través de la red.

- Software a nivel de sistema.

  Este tipo de software se sitúa a nivel de sistema, suele estar implementado como parte del sistema operativo de cada nodo, o ser la totalidad de éste.

  Es más crítico y complejo, por otro lado suele resolver problemas de carácter más general que los anteriores y su eficiencia, por norma general, es mayor.

A pesar de esta división existen casos en los cuales se hace uso de un conjunto de piezas de software de cada tipo para conformar un sistema cluster completo. Son implementaciones híbridas donde un cluster puede tener implementado a nivel de kernel parte del sistema y otra parte estar preparada a nivel de usuario.

### Acoplamiento de un cluster

Dependiendo del tipo de software, el sistema puede estar más o menos acoplado.

Se entiende por acoplamiento del software a la *integración que tengan todos los elementos software que existan en cada nodo*. Gran parte de la integración del sistema la produce la comunicación entre los nodos, y es por esta razón por la que se define el acoplamiento; otra parte es la que implica cómo de crítico es el software y su capacidad de recuperación ante fallos.

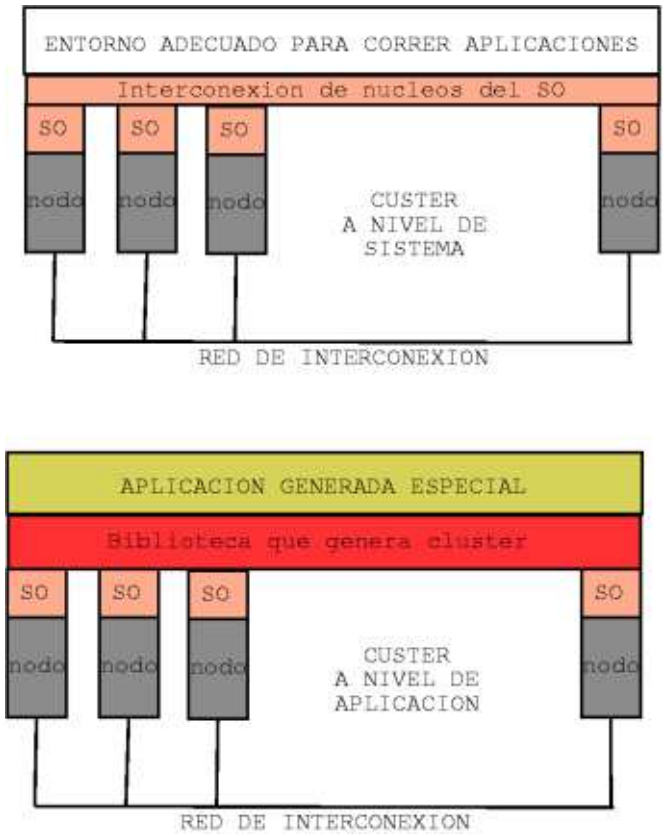



Figura 4.1: Clusters. Cluster a nivel de sistema y nivel de aplicación

Aquí hay que hacer un pequeño inciso para destacar que todo esto depende de si el sistema es centralizado o distribuido. En cualquier caso, el acoplamiento del software es una medida subjetiva basada en la integración de un sistema cluster a nivel general.

Se distingue entre 3 tipos de acoplamiento:

- Acoplamiento fuerte
- Acoplamiento medio
- Acoplamiento débil

Tras algunos ejemplos explicativos de estos tipos de acoplamiento quedará más clara la idea.

**Acoplamiento fuerte.**

El software que entra en este grupo es software cuyos elementos se interrelacionan mucho unos con otros y posibilitan la mayoría de las funcionalidades del cluster de manera altamente cooperativa. El caso de acoplamiento más fuerte que se puede dar es que solamente haya una imagen del kernel del sistema operativo, distribuida entre un conjunto de nodos que la compartirán. Por supuesto algo fundamental es poder acceder a todas las partes de este sistema operativo, estrechamente relacionadas entre sí y distribuidas entre los nodos.

Este caso es el que se considera como más acoplado, de hecho no está catalogado como cluster, sino como sistema operativo distribuido.

Otro ejemplo son los cluster SSI, en estos clusters todos los nodos ven una misma imagen del sistema, pero todos los nodos tienen su propio sistema operativo, aunque estos sistemas están estrechamente relacionados para dar la sensación a las aplicaciones que todos los nodos son idénticos y se acceda de una manera homogénea a los recursos del sistema total.

Si arranca o ejecuta una aplicación, ésta verá un sistema homogéneo, por lo tanto los kernels tienen que conocer los recursos de otros nodos para presentarle al sistema local los recursos que encontraría si estuviera en otro nodo. Por supuesto se necesita un sistema de nombres único, manejado de sistema distribuida o centralizada y un mapeo de los recursos físicos a este sistema de nombres.

**Acoplamiento medio.**

A este grupo pertenece un software que no necesita un conocimiento tan exhaustivo de todos los recursos de otros nodos, pero que sigue usando el software de otros nodos para aplicaciones de muy bajo nivel. Como ejemplos hay openMosix y Linux-HA.

Un cluster openMosix necesita que todos los kernels sean de la misma versión. Por otro lado no está tan acoplado como el caso anterior: no necesita un sistema de nombres común en todos los nodos, y su capacidad de dividir los procesos en una parte local y otra remota consigue que por un lado se necesite el software del otro nodo donde está la parte del fichero que falta en el nodo local y por otro que no se necesite un SSI para hacer otras tareas.

**Acoplamiento débil.**

Generalmente se basan en aplicaciones construidas por bibliotecas preparadas para aplicaciones distribuidas. Es el caso de por ejemplo PVM, MPI o CORBA. Éstos por sí mismos no funcionan en modo alguno con las características que antes se han descrito (como Beowulf) y hay que dotarles de una estructura superior que utilice las capacidades del cluster para que éste funcione.

### Esquema y otras características

Las características básicas de un cluster de carácter general podrían resumirse en el siguiente esquema:

1. Un cluster consta de 2 o más nodos conectados entre sípor un canal de comunicación funcional.
2. En cada nodo es imprescindible un elemento de proceso, memoria y un interfaz para comunicarse con la red del cluster.
3. Los clusteres necesitan software especializado. Este software y las máquinas conforman el cluster. El software puede ser:

   *a*) aplicación
   *b*) sistema

4. Se define acoplamiento de un cluster como nivel de colaboración que une los elementos del cluster. De este modo se categorizan en:
   *a*) Acoplamiento fuerte
   *b*) Acoplamiento medio o moderado
   *c*) Acoplamiento débil
5. Todos los elementos del cluster trabajan para cumplir una funcionalidad conjunta, sea esta la que sea. Es la funcionalidad la que caracteriza el sistema.

Muchos libros dan otra serie de características necesarias, por ejemplo el *Scalable Parallel Computing* de Kai Hwang y Zhiwei Xu incluye entre estas características las siguientes:

- Mejora sobre la disponibilidad
- Mejora del rendimiento

En general la catalogación de los clusters se hace en base a cuatro factores de diseño bastante ortogonales entre sí :

- Acoplamiento
- Control
- Homogeneidad
- Seguridad

De estos factores en este tema ya se ha visto el que quizás es más importante, el de acoplamiento.

Por otro lado está el factor de control del cluster. El parámetro de control implica el modelo de gestión que propone el cluster. Este modelo de gestión hace referencia a la manera de configurar el cluster y es dependiente del modelo de conexionado o colaboración que surgen entre los nodos. Puede ser de dos tipos:

- **Control centralizado:** se hace uso de un nodo maestro desde el cual se puede configurar el comportamiento de todo el sistema. Este nodo es un punto crítico del sistema aunque es una ventaja para una mejor gestión del cluster.
- **Control descentralizado:** en un modelo distribuido donde cada nodo debe administrarse y gestionarse. También pueden ser gestionados mediante aplicaciones de más alto nivel de manera centralizada, pero la mayoría de la gestión que hace el nodo local es leer archivos de configuración de su propio nodo.

  Es propio de sistemas distribuidos, como ventaja tiene que presenta más tolerancia a fallos como sistema global, y como desventajas que la gestión y administración de los equipos requiere más tiempo.

En lo que se refiere a homogeneidad de un cluster cabe decir que Tanenbaum no llevaba razón, a pesar de que sus conclusiones después de haber creado el sistema Amoeba. Se ha demostrado que es posible crear sistemas de una sola imagen o hetereogéneos con una implementación práctica. En cualquier caso, hay que entender por homogeneidad del cluster a la homogeneidad de los equipos y recursos que conforman a éste. Los clusters hetereogéneos son más difíciles de conseguir ya que se necesitan notaciones abstractas de transferencias e interfaces especiales entre los nodos para que éstas se entiendan, por otro lado los clusters homogéneos obtienen más beneficios de estos sistemas y pueden ser implementados directamente a nivel de sistema.

**Homogeneidad de un cluster**

- **Homogéneos:** formados por equipos de la misma arquitectura. Todos los nodos tienen una arquitectura y recursos similares, de manera que no existen muchas diferencias entre cada nodo.
- **Hetereogéneos:** formados por nodos con distinciones que pueden estar en los siguientes puntos.

- Tiempos de acceso distintos
- Arquitectura distinta
- Sistema operativo distinto
- Rendimiento de los procesadores o recursos sobre una misma arquitectura distintos

El uso de arquitecturas distintas, o distintos sistemas operativos, impone que exista una biblioteca que haga de interfaz, e incluso una sintaxis de notación abstracta del tipo ASN.1 o XDR en la capa de presentación que utilice la interfaz de comunicación de nuestro sistema distribuido o cluster. Esto hace que este tipo de clusters se consideren implementados a nivel de aplicación.

Existen otros muchos factores de diseño que limitan el comportamiento y modelado de un cluster. La imposibilidad de llegar a clusters que paralelicen cualquier proceso se basa en que la mayoría de las aplicaciones hacen uso, en mayor o menor medida, de algoritmos secuenciales no paralelizables.

### 4.1.3. Clasificación según el servicio prioritario

Generalmente el diseño de un cluster se realiza para solucionar problemas de tipo:

- Mejora de rendimiento
- Abaratamiento del coste
- Distribución de factores de riesgo del sistema
- Escalabilidad

El punto inicial ha sido explicado anteriormente: el coste para doblar las prestaciones de un equipo no suele ser habitualmente a costa de pagar el doble, sino unas cuantas veces más. El modelo de los clusters permite que la mejora de rendimiento sea evidente respecto a grandes mainframes a un precio realmente asequible. Lo que explica a su vez el segundo punto, acerca del coste de los clusters, que permite relaciones rendimiento precio que se acercan a un margen lineal dependiendo del cluster implementado.

Por otro lado esta la distribución de riesgos. La mayoría de los usuarios tienen sus servicios, aplicaciones, bases de datos o recursos en un solo ordenador, o dependientes de un solo ordenador. Otro paso más adelante es colocar las bases de datos replicadas sobre sistemas de archivos distribuidos de manera que estos no se pierdan por que los datos son un recurso importante. Actualmente el mercado de la informática exige no solo que los datos sean críticos sino que los servicios estén activos constantemente. Esto exige medios y técnicas que permitan que el tiempo en el que una máquina esté inactiva sea el menor posible. La distribución de factores de riesgo a lo largo de un cluster (o la distribución de funcionalidades en casos más generales) permite de una forma única obtener la funcionalidad de una manera más confiable: si una máquina cae otras podrán dar el servicio.

Por último está el factor de escalabilidad, del cual se habló en el tema de introducción. Cuanto más escalable es un sistema menos costará mejorar el rendimiento, lo cual abarata el coste, y en el caso de que el cluster lo implemente distribuye más el riesgo de caída de un sistema.

En cualquier caso, todas estas características dan pie a los tipos de clusters que se van a ver.

**Alto rendimiento (HP, high performance)**

Los clusters de alto rendimiento han sido creados para compartir el recurso más valioso de un ordenador, es decir, el tiempo de proceso. Cualquier operación que necesite altos tiempos de CPU puede ser utilizada en un cluster de alto rendimiento, siempre que se encuentre un algoritmo que sea paralelizable.

Existen clusters que pueden ser denominados de alto rendimiento tanto a nivel de sistema como a nivel de aplicacion. A nivel de sistema tenemos openMosix, mientras que a nivel de aplicación se encuentran otros como MPI, PVM, Beowulf y otros muchos. En cualquier caso, estos clusters hacen uso de la capacidad de procesamiento que pueden tener varias máquinas.

**Alta disponibilidad (HA, high availability)**

Los clusters de **alta disponibilidad** son bastante ortogonales en lo que se refieren a funcionalidad a un cluster de alto rendimiento. Los clusters de alta disponibilidad pretenden dar servicios 7/24 de cualquier tipo,

son clusters donde la principal funcionalidad es estar controlando y actuando para que un servicio o varios se encuentren activos durante el máximo periodo de tiempo posible. En estos casos se puede comprobar como la monitorización de otros es parte de la colaboración entre los nodos del cluster.

**Alta confiabilidad (HR, high reliability)**
Por ultimo, están los clusters de alta confiabilidad. Estos clusters tratan de aportar la máxima confiabilidad en un entorno, en la cual se necesite saber que el sistema se va a comportar de una manera determinada. Puede tratarse por ejemplo de sistemas de respeusta a tiempo real.

Pueden existir otras catalogaciones en lo que se refiere a tipos de clusters, en nuestro caso, solamente hemos considerado las tres que más clusters implementados engloban, si bien existe alguno de ellos que puede ser considerar o como cluster de varios tipos a la vez.

### 4.1.4. Clusters HP: alto rendimiento

Se harán distinciones entre los que se implementan a nivel de sistema operativo y los que se implementan a nivel de librerías, y se explicarán qué tipo de problemas resuelven ambos.

#### La misión

La misión o el objetivo de este tipo de clusters es, como su propio nombre indica mejorar el rendimiento en la obtención de la solución de un problema, en términos bien del tiempo de respuesta bien de su precisión.
Dentro de esta definición no se engloba ningún tipo de problema en concreto. Esto supone que cualquier cluster que haga que el rendimiento del sistema aumente respecto al de uno de los nodos individuales puede ser considerado cluster HP.

#### Problemas que solucionan

Generalmente estos problemas de computo suelen estar ligados a:

- Cálculos matemáticos
- Renderizaciones de gráficos
- Compilación de programas
- Compresión de datos
- Descifrado de códigos
- Rendimiento del sistema operativo, (incluyendo en él, el rendimiento de los recursos de cada nodo)

Existen otros muchos problemas más que se pueden solucionar con clusters HP, donde cada uno aplica de una manera u otra las técnicas necesarias para habilitar la paralelización del problema, su distribución entre los nodos y obtención del resultado.

#### Técnicas que utilizan

Las técnicas utilizadas dependen de a qué nivel trabaje el cluster.
Los clusters implementados a nivel de aplicacion no suelen implementar balanceo de carga. Suelen basar todo su funcionamiento en una política de localización que sitúa las tareas en los diferentes nodos del cluster, y las comunica mediante las librerías abstractas. Resuelven problemas de cualquier tipo de los que se han visto en el apartado anterior, pero se deben diseñar y codificar aplicaciones propias para cada tipo para poderlas utilizar en estos clusters.
Por otro lado están los sistemas de alto rendimiento implementados a nivel de sistema. Estos clusters basan todo su funcionamiento en comunicación y colaboración de los nodos a nivel de sistema operativo, lo que implica generalmente que son clusters de nodos de la misma arquitectura, con ventajas en lo que se refiere al factor de acoplamiento, y que basan su funcionamiento en compartición de recursos a cualquier nivel, balanceo de la carga

de manera dinámica, funciones de planificación especiales y otros tantos factores que componen el sistema. Se intentan acercar a sistemas SSI, el problema está en que para obtener un sistema SSI hay que ceder en el apartado de compatibilidad con los sistemas actuales, por lo cual se suele llegar a un factor de compromiso.

Entre las limitaciones que existen actualmente está la incapacidad de balancear la carga dinámica de las librerías PVM o la incapacidad de openMosix de migrar procesos que hacen uso de memoria compartida. Una técnica que obtiene mayor ventaja es cruzar ambos sistemas: PVM + openMosix. Se obtiene un sistema con un factor de acoplamiento elevado que presta las ventajas de uno y otro, con una pequeña limitación por desventajas de cada uno.

### 4.1.5. Clusters HA: alta disponibilidad

Son otro tipo de clusters completamente distintos a los anteriores. Son los más solicitados por las empresas ya que están destinados a mejorar los servicios que éstas ofrecen cara a los clientes en las redes a las que pertenecen, tanto en redes locales como en redes como Internet. En este apartado se darán las claves que explican tanto el diseño de estos clusters asícomo algún factor de implementación.

#### La misión

Los clusters de alta disponibilidad han sido diseñados para la máxima disponibilidad sobre los servicios que presenta el cluster. Este tipo de clusters son la competencia que abarata los sistemas redundantes, de manera que ofrecen una serie de servicios durante el mayor tiempo posible. Para poder dar estos servicios los clusters de este tipo se implementan en base a tres factores.

- Fiabilidad
- Disponibilidad
- Dotación de servicio

Mediante estos tres tipos de actuaciones y los mecanismos que lo implementan se asegura que un servicio esté el máximo tiempo disponible y que éste funcione de una manera fiable. Respecto al tercer punto, se refiere a la dotación de uno de estos clusters de un servicio que provea a cliente externos.

#### Problemas que solucionan

La mayoría de estos problema están ligados a la necesidad de dar servicio continuado de cualquier tipo a una serie de clientes de manera ininterrumpida. En una construcción real se suelen producir fallos inesperados en las máquinas, estos fallos provocan la aparición de dos eventos en el tiempo: el tiempo en el que el servicio está inactivo y el tiempo de reparación del problema.

Entre los problemas que solucionan se encuentran:

- Sistemas de información redundante
- Sistemas tolerantes a fallos
- Balanceo de carga entre varios servidores
- Balanceo de conexiones entre varios servidores

En general todos estos problemas se ligan en dos fuentes de necesidad de las empresas u organizaciones.

- Tener un servicio disponible
- Ahorrar económicamente todo lo que sea posible

El servicio puede ser diverso. Desde un sistema de ficheros distribuidos de carácter muy barato, hasta grandes clusters de balanceo de carga y conexiones para los grandes portales de Internet. Cualquier funcionalidad requerida en un entorno de red puede ser colocada en un cluster y implementar mecanismos para hacer que esta obtenga la mayor disponibilidad posible.

**Técnicas que utilizan**

Como se ha visto en el apartado anterior los servicios y el funcionamiento de los mismos suelen ser de carácter bastante distinto, en cualquier caso, se suelen proponer sistemas desde SSI que plantean serias dudas en lo que se refiere a localización de un servidor, hasta balanceo de carga o de conexiones. También suelen contener secciones de código que realizan monitorización de carga o monitorización de servicios para activar las acciones necesarias para cuando estos caigan.

Se basan en principios muy simples que pueden ser desarrollados hasta crear sistemas complejos especializados para cada entorno particular. En cualquier caso, las técnicas de estos sistemas suelen basarse en excluir del sistema aquellos puntos críticos que pueden producir un fallo y por tanto la pérdida de disponibilidad de un servicio. Para esto se suelen implementar desde enlaces de red redundantes hasta disponer de `N` máquinas para hacer una misma tarea de manera que si caen `N-1` máquinas el servicio permanece activo sin pérdida de rendimiento.

La explicación de estas técnicas ha sido muy somera, se darán con más detalle en el caítulo perteneciente a clusters HA.

### 4.1.6. Clusters HR: alta confiabilidad

Este tipo de clusters son los más difíciles de implementar. No se basan solamente en conceder servicios de alta disponibilidad, sino en ofrecer un entorno de sistema altamente confiable. Esto implica muchísima sobrecarga en el sistema, son también clusters muy acoplados.

Dar a un cluster SSI capacidad de alta confiabilidad implica gastar recursos necesarios para evitar que aplicaciones caigan. Aquí hay que hacer de nuevo un inciso.

En los clusters de alta disponibilidad generalmente una vez que el servicio ha caído éste se relanza, y no existe manera de conservar el estado del servidor anterior, más que mediante puntos de parada o *checkpoints*, pero que en conexiones en tiempo real no suelen ser suficientes. Por otro lado, los clusters confiables tratan de mantener el estado de las aplicaciones, no simplemente de utilizar el ultimo checkpoint del sistema y relanzar el servicio.

Generalmente este tipo de clusters suele ser utilizado para entornos de tipo empresarial y esta funcionalidad solamente puede ser efectuada por hardware especializado. Por elmomento no existe ninguno de estos clusters implementados como software. Esto se debe a limitaciones de la latencia de la red, así como a la complejidad de mantener los estados.

Se hacen necesarias características de cluster SSI, tener un único reloj de sistema conjunto y otras más. Dada la naturaleza asíncrona actual en el campo de los clusters, este tipo de clusters aún será difícil de implementar hasta que no se abaraten las técnicas de comunicación.

## 4.2. CLUSTERS HA

*Just in terms of allocation of time resources, religion is not very efficient.*
*There's a lot more I could be doing on a Sunday morning.*
otra muestra de inteligencia de *Bill Gates*

### 4.2.1. Introducción

Los clusters HA estan diseñados especialmente para dar un servicio de alta disponibilidad. Esto tiene muchas aplicaciones en el mundo actual donde existen gran cantidad de servicios informáticos que debe funcionar 24x7x365. Estos clusteres son una alternativa real a otros sistemas usados tradicionalmente para estas tareas de hardware redundante que son mucho más caros, por ejemplo los ordenadores Tandem.

### 4.2.2. El interés comercial

Desde ya hace unos años Heartbeat está en las distribuciones SuSE, Conectiva y Mandrake; incluso Mission Critical Linux se ha interesado en él. Todo esto es así porque el mercado de clusters HA es un mercado con muchos potenciales clientes y, lo que es quizás más importante, para los intereses comerciales de muchas empresas.

Piénsese como ejemplo una empresa de grandes almacenes que tiene ordenadores carísimos validando las operaciones de tarjeta de crédito. Estos ordenadores no deben cancelar el servicio nunca porque si lo hicieran todo el sistema de tarjetas de créditos se vendría abajo, con lo que se podrían ocasionar grandes pérdidas económicas.

Por todo esto se desarrollan proyectos que intentan conseguir esta disponibilidad pero no gracias a un soporte hardware carísimo, sino usando clusters. Las empresas que necesitan alta disponibilidad suelen pagar a la empresa que le ofrece este servicio aun cuando los programas sean de libre distribución, quieren unas garantías. Esto está haciendo que muchas empresas estén colaborando en proyectos libres de HA, cosa que no deja de ir en pro de la mejora del software en cuestión.

Las enormes diferencias entre el precio del hardware de las soluciones tradicionales y estas nuevas soluciones hacen que las empresas tengan un buen margen de beneficio dando un servicio de soporte.

Es bastante obvio por qué estos clusters estan más solicitados que los de alto rendimiento (HP): la mayoría de las empresas se pueden permitir en cierto modo máquinas potentes que les solucionen las necesidades de cómputo, o simplemente contar con el tiempo suficiente para que sus actuales equipos procesen toda la información que necesitan. En la mayoría de los casos el tiempo no es un factor crítico y por tanto la velocidad o la capacidad de cómputo de las máquinas no es importante. Por otro lado, que se repliquen sistemas de archivos para que estén disponibles, o bases de datos, o servicios necesarios para mantener la gestión de la empresa en funcionamiento, o servicios de comunicación interdepartamental en la empresa y otros servicios, son realmente críticos para las empresas en todos y cada uno de los días en los que estas están funcionando (e incluso cuando no están funcionando).

### 4.2.3. Conceptos importantes

Un buen cluster HA necesita proveer fiabilidad, disponibilidad y servicio RAS (explicado más adelante). Por tanto debe existir una forma de saber cuándo un servicio ha caído y cuándo vuelve a funcionar.

Esto se puede conseguir de dos maneras, por hardware y por software. No se van a tratar aquí los mecanismos que existen para conseguir alta disponibilidad por hardware, pues están más allá de los objetivos de este documento. Baste decir que construir estos ordenadores es muy caro pues necesitan gran cantidad de hardware redundante que esté ejecutando paralelamente en todo momento las mismas operaciones que el hardware principal (por ejemplo una colección de placas base) y un sistema para pasar el control o la información del hardware con errores a hardware que se ejecute correctamente.

Los clusters HA solucionan el problema de la disponibilidad con una buena capa de software. Por supuesto mejor cuanto más ayuda se tenga del hardware: UPS, redes ópticas, etc. Pero la idea tras estos sistemas es no

tener que gastarse millones de euros en un sistema que no puede ser actualizado ni escalado. Con una inversión mucho menor y con software diseñado específicamente se puede conseguir alta disponibilidad.

Para conseguir la alta disponibilidad en un cluster los nodos tienen que saber cuándo ocurre un error para hacer una o varias de las siguientes acciones:

- **Intentar recuperar los datos del nodo que ha fallado.**

  Cuando un nodo cae hay que recuperar de los discos duros compartidos por los nodos la información para poder seguir con el trabajo. Generalmente hay *scripts* de recuperación para intentar recuperarse del fallo.

- **Continuar con el trabajo que desempeñaba el nodo caído.**

  Aquí no se intenta recuperar del fallo sino que cuando se descubre que ocurrió un fallo otro nodo pasa a desempeñar el puesto del nodo que falló.

  Esta es la opción que toma por ejemplo Heartbeat: permite que 2 ordenadores mantengan una comunicación por un cable serie o Ethernet, cuando un ordenador cae el ordenador que no recibe respuesta ejecuta las órdenes adecuadas para ocupar su lugar.

Las ventajas de escalabilidad y economía de los clusters tienen sus desventajas. Una de ellas es la seguridad. Cuando se diseña un cluster se busca que haya ciertas facilidades de comunicación entre las estaciones y en clusters de alta disponibilidad el traspaso de información puede ser muy importante.

Recordando el ejemplo anterior de las tarjetas de crédito, se ha visto que se podría crear un cluster de alta disponibilidad que costara varias veces menos que el ordenador centralizado. El problema podría sobrevenir cuando ese cluster se encargara de hacer operaciones con los números de las tarjetas de crédito y transacciones monetarias de la empresa. Las facilidades de comunicación podrían ocasionar un gravísimo problema de seguridad. Un agente malicioso podría hacer creer al cluster que uno de los nodos ha caído, entonces podría aprovechar el traspaso de la información de los nodos para conseguir los números de varias tarjetas de crédito.

Este es uno de los motivos por los que, en según qué entornos, estos sistemas no se hayan implantado.

#### Servicio RAS

En el diseño de sistemas de alta disponibilidad es necesario obtener la suma de los tres términos que conforman el acrónimo RAS.

- **Reliability.**

  El sistema debe ser confiable en el sentido de que éste actúe realmente como se ha programado. Por un lado está el problema de coordinar el sistema cuando éste está distribuido entre nodos, por otro lado hay el problema de que todo el software que integra el sistema funcione entre sí de manera confiable.

  En general se trata de que el sistema pueda operar sin ningún tipo de caída o fallo de servicio.

- **Availability.**

  Es lógicamente la base de este tipo de clusters. La disponibilidad indica el porcentaje de tiempo que el sistema esta disponible en su funcionalidad hacia los usuarios.

- **Serviceability.**

  Referido a cómo de fácil es controlar los servicios del sistema y qué servicios se proveen, incluyendo todos los componentes del sistema.

La disponibilidad es el que prima por encima de los anteriores. La disponibilidad de un sistema es dependiente de varios factores. Por un lado el tiempo que el sistema está funcionando sin problemas, por otro lado el tiempo en el que el sistema esta fallando y por último el tiempo que se tarda en reparar o restaurar el sistema.

Para medir todos estos factores son necesarios fallos. Existen dos tipos de fallos: los fallos que provocan los administradores (para ver o medir los tiempos de recuperación y tiempos de caídas) y los fallos no provocados, que son los que demuestran que los tiempos de reparación suelen ser mucho más grandes de los que se estimó en los fallos provocados.

La naturaleza de los fallos puede afectar de manera diferente al sistema: pudiendolo ralentizar, inutilizar o para algunos servicios.

**Técnicas para proveer de disponibilidad**

Cualquier técnica deberá, por definición, intentar que tanto el tiempo de fallo del sistema como el tiempo de reparación del mismo sean lo más pequeños posibles.

Las que tratan de reducir el tiempo de reparación se componen a base de *scripts* o programas que detectan el fallo del sistema y tratan de recuperarlo sin necesidad de un técnico especializado. En general son técnicas de automatización de tareas basadas en sistemas expertos (SE). Al reducir el tiempo de recuperación, el sistema puede no solamente funcionar activamente sin fallos durante más tiempo, sino que también se aumenta su confiabilidad.

***i.*- Técnicas basadas en redundancia**

Por un lado hay las técnicas basadas en reducir el tiempo de fallo o caída de funcionamiento, estas técnicas se basan principalmente en efectuar algún tipo de redundancia sobre los dispositivos críticos. Saber cuáles son estos dispositivos suele ser cuestión de conocimiento acerca del sistema y de sentido común.

Las técnicas basadas en la redundancia de recursos críticos permiten que cuando se presenta la caída de uno de estos recursos, otro tome la funcionalidad. Una vez esto sucede el recurso maestro puede ser reparado mientras que el recurso *backup* toma el control. Entre los tipos de redundancia que pueden presentar los sistemas hay:

- Redundancia aislada.

  Es la redundancia más conocida donde existen 2 copias para dar una funcionalidad o servicio. Por un lado hay la copia maestro y por otro lado la copia esclavo. Cuando cae el servicio o recurso la copia redundante pasa a ser la utilizada, de esta manera el tiempo de caída es mínimo o inexistente.

  Los recursos pueden ser de cualquier tipo: procesadores, fuentes de alimentación, raids de discos, redes, imágenes de sistemas operativos...

  Las ventajas son cuantiosas: para empezar no existen puntos críticos de fallo en el sistema, es decir, el sistema al completo no es tomado como un sistema con puntos de fallos que bajen la confiabilidad del mismo. Los componentes que han fallado pueden ser reparados sin que esto cause sobre el sistema una parada.

  Por último, cada componente del sistema puede comprobar de manera periódica si se ha producido algún tipo de fallo en los sistemas de backup, de modo que se compruebe que éstos están siempre funcionales. Otra opción es que además de comprobar, presenten habilidades para localizar fallos en sistemas y los intenten recuperar de manera automatizada.

- N-Redundancia.

  Es igual que la anterior pero se tienen `N` equipos para ofrecer un mismo servicio, con lo cual presenta más tolerancia a fallos.

Así por ejemplo para dotar de sistema redundante a una red como la que muestra el esquema A de la figura debería haber el doble de recursos necesarios para construirla, e implementarlos como en el sistema B. En este caso se replicaron:

- LAN
- LAN para los servidores
- Los servidores
- El bus SCSI de acceso a discos duros
- Discos duros

Como se puede ver la replicación proporciona rutas alternativas, discos alternativos y en general recursos alternativos, y es aplicada sobre todos aquellos recursos que se consideren críticos en una red.

Otro apartado a la hora de considerar la instalación de dispositivos redundantes es la configuración o el modelo de funcionamiento de los mismos. Depende de como se haya implementado el software y como se haya dispuesto el hardware y define el modo de comportamiento de la redundancia. Esta redundancia puede ser del tipo:

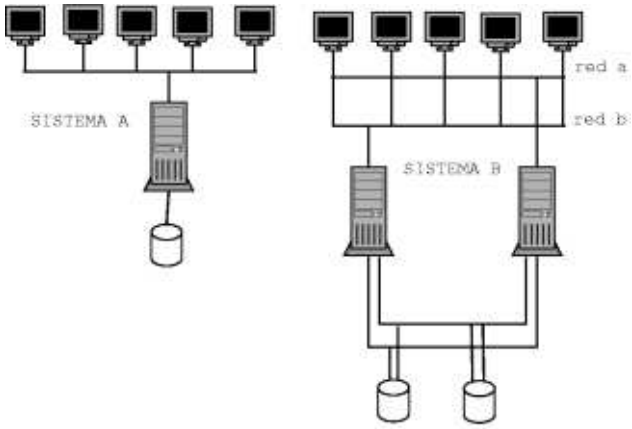



Figura 4.2: Clusters HA. Redundancia

1. **Hot Standby.**

   Este tipo de configuración es la que se ha visto hasta ahora. En cuando el nodo maestro cae existe un nodo backup que toma el control de sus operaciones. Hasta ahora no se ha tenido en cuenta un punto importante: el doble gasto en recursos que en un principio y si las cosas van bien se están desperdiciando.

   El servidor de backup simplemente monitoriza sus conexiones, la normal y la redundante en el caso de que esta exista, para asegurar que cuando el nodo maestro caiga tome correctamente el control de las operaciones. Exceptuando estas operaciones, el nodo backup no hace nada.

2. **Toma de cargos mutua.**

   La toma de cargos mutua es una configuración que soluciona la desventaja del apartado anterior. Mientras el nodo principal se ocupa de proveer de servicio, el nodo de backup hace las mismas operaciones que en apartado anterior y además puede efectuar otro tipo de operaciones. De este modo, la capacidad de este nodo se está aprovechando más que en el anterior y el costo del sistema se ve recompensado con un equipo más que se utiliza para efectuar trabajo útil. El problema está cuando el nodo maestro cae. En este caso, el comportamiento del backup puede tomar dos vías.

   - la primera es mantener sus actuales trabajos y tomar los trabajos que el maestro ha dejado sin hacer. Esta manera de comportarse presenta una bajada de rendimiento del sistema crítico, pero hace que este esté disponible. Depende del tipo de trabajo que se presente sobre el backup que ahora ha pasado a ser maestro el considerar la prioridad de que trabajos son más críticos.
   - la segunda opción es dejar estos trabajos postergados hasta que el antiguo maestro sea reparado (lo cual puede ser hecho por el nodo de backup, es decir el nuevo maestro) y cuando éste tome las tareas de backup, que continúe con el trabajo que en un principio estaba efectuando él antes de tomar el control.

     Esta es una solución mucho más elegante y más difícil de implementar. Presenta mejor rendimiento coste que la anterior.

3. **Tolerante a fallos**

   Los sistemas redundantes a fallos se basan en la N-Redundancia. Si se tienen `N` equipos y caen `N-1` el sistema sigue en funcionamiento. Este sistema se puede cruzar con la toma de cargos mutua para no perder rendimiento ni elevar el costo del sistema, sin embargo configurar un sistema de este tipo es bastante complejo a medida que aumenta `N`.

***ii.-* Técnicas basadas en reparación**

Por otro lado están las técnicas basadas en reducir el tiempo de reparación. Este tipo de técnicas se componen a base de scripts o programas que detectan donde fallo el sistema y tratan de recuperarlo sin necesidad de un técnico especializado. En general son técnicas de automatización de tareas basadas en sistemas expertos.

Al reducir el tiempo de recuperación, el sistema puede no solamente funcionar activamente sin fallos más tiempo, sino que también aumentamos la confiabilidad. Se pueden separar en dos tipos de acciones que realizan que pueden ser independientes o dependientes entre si:

- **Diagnóstico.**

  La parte de diagnosis simplemente trata de conocer las posibles causas que han provocado el error y principalmente localizar el error.

  Una técnica muy utilizada en este campo es una especie de *piggybacking* aplicada a los pulsos o latidos entre ambos nodos. En esta técnica, se envía junto con el latido o pulso, la suficiente información como para prever cual será el estado de los componentes en próximos tiempos o incluso actualmente, lo cual es una ventaja a la hora de saber en todo momento el estado del sistema. La implementación Heartbeat de Linux-HA hace una implementación muy coherente y correcta de esta técnica.

- **Reparación.**

  Son técnicas mediante las cuales a través de sistemas expertos o cualquier otro tipo de actuación, el sistema caído puede llegar a ser restaurado desde una copia del sistema. En la mayoría de los casos basa su funcionamiento en puntos de comprobación o *checkpoints* que se efectúan sobre el sistema cada cierto tiempo, de manera que el servidor caído es restaurado al último *checkpoint* existente. Los puntos críticos de este apartado son:

  - aislar los componentes que fallan y sustituirlos o repararlos. Los componentes que fallan pueden ser localizados mediante programas que implementen sistemas de comprobación o sistemas expertos.
  - recuperación mediante puntos de comprobación o puntos de restauración.
  - acoplamiento al cluster actual tomando las acciones que tenía el nodo backup e informando al nodo maestro de que ya existe un nuevo backup en el sistema.

  Los puntos de comprobación son importantes ya que introducen un factor de sobrecarga en el sistema y al mismo tiempo son un factor crítico a la hora de efectuar restauraciones del mismo. Un sistema al máximo confiable debería guardar el estado de todos los programas que está corriendo y comunicárselos a su backup en tiempo real para que de este modo la caída de uno guardase el estado del complementario. Serían sistemas simétricos.

  Este tipo de sistemas solamente son implementados en hardware, ya que exigen medios de comunicación muy rápidos (aparte de la sobrecarga al procesador que genera estar controlando este tipo de labores). Los clusters implementan a un nivel mucho menos eficiente este tipo de *checkpoints*, y la eficiencia depende generalmente de la capacidad de los procesadores y de la capacidad de la red que une maestro y backups. Debido a esto los clusters solamente guardan configuraciones y servicios activos, pero no el estado de conexiones y demás componentes que harían que un usuario externo observase la caída del sistema de modo realmente transparente, como si este no existiese.

  Esta es una de las grandes diferencias entre entornos de alta disponibilidad y entornos de alta confiabilidad, de los cuales no se ha visto ninguno implementado debido a que la tecnología actual los hace inviables.

  Existen varias maneras de efectuar el punto de comprobación. Por ejemplo en los sistemas implementados a nivel de kernel o sistema, el sistema operativo se encarga de efectuar este de manera completamente transparente al usuario o administrador. En los sistemas a nivel de aplicación son generalmente las bibliotecas de funciones las que proveen de estas características.

  Otro factor a tener en cuenta acerca de los checkpoints, que marca el rendimiento del sistema, es su intervalo. Éste debe ser óptimo: no crear congestión y permitir copias de restauración lo suficientemente actuales como para que los servicios cuenten con la máxima disponibilidad. En ocasiones, cuando el checkpoint es muy grande puede provocar congestiones. En cualquier caso, el principal problema de un checkpoint es la información que necesita para hacer la colaboración eficiente, y esto como hemos visto depende siempre del tipo de sistemas.

  Como última característica de los checkpoints, hacer una pequeña mención en los sistemas con más de un nodo de redundancia, en los cuales se pueden imaginar dos modos lógicos de hacer los checkpoints:

1. Como checkpoints aislados, donde cada nodo se encarga de hacer los checkpoints de otro nodo al que replica cada intervalo de tiempo o por una política propia del sistema (puede caer en el denominado efecto domino, en el cual se guardan copias de sistema que no corresponden con el estado actual de los nodos).
2. Frente a los checkpoints en grupo o checkpoints organizados, en los cuales todos los nodos se ponen de acuerdo para hacer un sistema de checkpoints efectivo. A cambio requiere más dificultad de implementación, y quizá sobrecarga del sistema.

### 4.2.4. Soluciones libres

**Linux-HA**

Este es el mayor proyecto de software libre de clusters HA que existe, parte de este proyecto es Heartbeat y trabajan conjuntamente con el grupo encargado de LVS.

Han desarrollado varias aplicaciones comerciales sobre este proyecto y se está utilizando en varios servicios con éxito. Como parte de los objetivos que se persiguen se encuentran:

- *Servicios de membership.*

  Estos servicios permiten añadir y quitar miembros a un cluster. El problema es que la llegada de un miembro a un cluster orientado a estados puede hacer cambiar de estado a todo el cluster (esto suele ser lo que ocurre en este tipo de clusters) con lo que se envían demasiados paquetes de sobrecarga demasiado a menudo por tanto ante esto se plantean soluciones como clusteres jerárquicos. Es la solución que dentro de Linux-HA ha sido apoyada por Stephen Tweedie.

- *Servicios de comunicación.*

  Comunicar información crítica de forma que una caída en un sistema no haga que se pierda la información y a la vez enviar la información de una forma suficientemente segura para evitar posibles ataques externos. Como se ha visto esto es especialmente importante en clusters HA.

- *Manejo del cluster.*

  Una serie de servicios que hagan sencillo el manejo del cluster en general y de los nodos y procesos en particular. Al igual que un sistema operativo provee de servicios para administrarlo, un cluster también debe proveer de instrucciones para gestionar su funcionamiento.

- *Monitorización de los recursos.*

  Este punto está muy unido al anterior. Para que el administrador detecte prematuramente posibles fallos y pueda ver qué ocurre en el cluster necesita algunas facilidades de monitorización. Por supuesto estos dos puntos no son exclusivos de los clustersHA ni del proyecto Linux-HA sino que son necesarios en todos los clusters.

- *Replicación y/o compartición de datos.*

  Para conseguir que los datos que estuviera modificando uno de los nodos no se pierda cuando caiga se puede replicar la información y/o mantenerla en lugares compartidos por todos los nodos con lo que cualquier nodo podría continuar con los datos compartidos.

  Para conseguir tener unos discos compartidos se necesita un hardware caro como es SCSI y fibra óptica.

  La replicación de información no necesita un hardware caro (solamente una red tan rápida como el coste permita) pero se necesita mantener un equilibrio entre los periodos de actualización de las copias y el uso de la red. Un cluster HA no suele necesitar demasiado ancho de banda por lo que se puede dedicar gran parte para este uso.

**HeartBeat**

Esta tecnología implementa *heartbeats*, cuya traducción directa sería *latidos de corazón*. Funciona enviando periódicamente un paquete que si no llegara indicaría que un servidor no está disponible, por lo tanto se sabe que el servidor ha caído y se toman las medidas necesarias.

Dichos *latidos* se pueden enviar por una linea serie, por UDP o por PPP/UDP. De hecho los desarrolladores de Heartbeat recomiendan el uso de puertos serie por varias razones, entre las que destacan que están aislados de las tarjetas de red.

También incluye toma de una dirección IP y una modelo de recursos incluyendo grupos de recursos. Soporta múltiples direcciones IP y un modelo servidor primario/secundario. Por ahora ya se ha probado útil para varias aplicaciones incluyendo servidores DNS, servidores proxy de cache, servidores web y servidores directores de LVS. El proyecto LVS recomienda HeartBeat para aumentar la disponibilidad de su solución, pero no es parte de LVS.

En Linux-HA Heartbeat es un servicio de bajo nivel. Cuando un ordenador se une al cluster se considera que el ordenador se ha unido al canal de comunicaciones (por lo tanto *late*) y cuando sale que ha dejado el canal de comunicaciones.

Cuando un ordenador deja de *latir* y se considera *muerto* se hace una transición en el cluster. La mayoría de los mensajes de manejo del cluster que no son heartbeats se realizan durante estas transiciones.

Los mensajes de Heartbeat se envían por todas las lineas de comunicación a la vez, así si una linea de apoyo cae, se avisará de ese problema antes de que la linea principal caiga y no haya una línea secundaria para continuar el servicio.

Heartbeat también se preocupa por la seguridad permitiendo firmar los paquetes CRC de 32 bits, MD5 y SHA1. Esto puede evitar el desastre que podría provocarse si un nodo no miembro se enmascarase como nodo miembro del cluster. Hay varias operaciones de mantenimiento de seguridad que necesitan ser efectuadas en ese tiempo, como pueden ser cambio de claves y de protocolos de autentificación. Heartbeat está preparado para esos cambios disponiendo de ficheros para la configuración.

Heartbeat tiene el problema que si no se dispone de una línea dedicada, aunque ésta sea una línea serie, al tener un tráfico que aunque pequeño es constante, suele dar muchas colisiones con otros tráficos que puedan ir por la misma red. Por ejemplo openMosix y Heartbeat en una misma red que no tenga gran ancho de banda no funcionan bien, sobre todo si hay bastantes nodos, pues los heartbeats se envían de cualquier nodo a cualquier nodo, por lo que podrían llegar a ser un tráfico voluminoso.

**Ldirectord.**

Pensado especialmente para ser usado junto con LVS, utiliza Heartbeat. Monitoriza que los servidores reales sigan funcionando periódicamente enviando una petición a una url conocida y comprobando que la respuesta contenga una cadena concreta. Si un servidor real falla entonces el servidor es quitado del conjunto de servidores reales y será reinsertado cuando vuelva a funcionar correctamente. Si todos los servidores fallan se insertará un servidor de fallos, que será quitado una vez que los servidores vuelvan a funcionar.

Típicamente este servidor de fallos es el propio host desdel que se realiza el monitoraje.

**LVS.**

Sobre este proyecto se dedica la siguiente subsección, por tanto de momento basta decir que éste es seguramente el proyecto más implantado y uno de los pilares sobre los que se apoyan las soluciones comerciales.

### 4.2.5. LVS (Linux Virtual Server)

#### Introducción

LVS es un proyecto que incluye los programas y documentación necesaria parar montar un cluster de servidores bajo GNU/Linux. El proyecto LVS es utilizado principalmente para aumentar rendimiento y escalabilidad de servicios ofrecidos sobre la red, es ampliamente utilizado por grandes sites como `SouceForge.net` o `Linux.com`.

La principal idea es proveer de un mecanismo de *migración de sockets*. El mecanismo se basa en utilizar una máquina servidora a la que se dirigen las peticiones de los usuarios clientes. El interfaz público (en Internet) de esta máquina normalmente tiene asociada una dirección conocida como VIP. El cometido de esta primera computadora es direccionar dichas peticiones a otros servidores reales mediante varias técnicas, de este modo los usuarios clientes ven un único servidor. No ostante éste opera con varias máquinas para conceder un servicio único al exterior.

A todo el conjunto de nodos que conforman el servicio y se comportan como si fuese un único servidor se le denomina Servidor Virtual. El cluster está formado por dos tipos de máquinas:

- por un lado están los nodos o **servidores reales**, que corren con los servicios habituales que estos suelen proveer,
- por otro lado están los **nodos directores**, de los cuales uno de ellos será el principal y el resto estarán preparados para hacer de refuerzo de éste (mediante técnicas o protocolos como heartbeat) para cuando caiga. Estas técnicas no son propias de LVS, como ya puede saberse a partir de las seccines anteriores.

En general se puede considerar LVS como una suma de herramientas que permiten efectuar la función ya especificada. Para cosneguirlo se requiere:

- el código de **ipvs**, un parche al kernel para el nodo director
- el programa **ipvsadm**, encargado de configurar las tablas internas y algoritmos del kernel del nodo director

Ambos constituyen el código principal del proyecto LVS, pero se requieren otras muchas herramientas como ipchains, iptables o Netfilter (dependiendo de la versión del núcleo utilizada ), Ldirectord, Heartbeat, Piranha, MON, LVS-gui, *etc.*

El Servidor Virtual[2] requiere de la configuración de todos estos servicios y herramientas para un funcionamiento adecuado, y no *solamente* del código de LVS. Es más, dentro del tipo de programas que conforman el Servidor Virtual no hay que olvidar los programas o demonios servidores habituales que proveerán realmente de servicio a los clientes finales (HTTPD, FTPD, SMTP, *etc.*).

El funcionamiento de LVS es una aproximación a resolver el problema de la escalabilidad y el alto rendimiento muy elegante puesto que permite que cliente y servidor trabajen de la manera transparente permitiendo que el balanceo se lleve a cabo por el director. Comparado con métodos como el ofrecido por RR-DNS es mucho menos intrusivo y más confiable en su servicio.

Existen otras técnicas para ofrecer mayor escalabilidad y rendimiento en servicios de Internet. La alternativa RR-DNS es uno de los métodos más utilizados actualmente ya que permite independencia en arquitectura o sistema utilizado en el servidor Se basa en un algoritmo Round Robin en el servidor DNS, de manera que cuando un cliente solicita la dirección IP de un servidor ésta le es concedida según el algoritmo. Así por ejemplo si existen 4 máquinas servidoras que proporcionan el mismo servicio, a la primera conexión entrante que solicite el servicio se le asignará la dirección IP del primer servidor, a la segunda conexión la IP del segundo servidor y a la quinta conexión la IP del primero otra vez.

Uno de los defectos que tiene este método es que si uno de los servidores cae los clientes que asociaban el dominio a esa dirección IP lo seguirán haciendo, obteniendo un fallo en sus peticiones.

El servicio RR-DNS puede ser utilizado complementariamente a LVS, es decir, se utilizar RR-DNS para solicitar la IP de un Servidor Virtual LVS. Otra alternativa es el uso de clientes no transparentes, clientes que utilicen algún algoritmo para decidir que servidor utilizan en cada petición o sesión, lógicamente estos métodos son muy poco utilizados. Esto puede conseguirse mediante applets Java, por ejemplo.

Otras alternativas más conocidas son los proxies inversos, esta alternativa supone el mismo funcionamiento de un proxy, pero en el caso de los servidores. El Proxy recive las peticiones y gestiona una nueva conexión con uno de los servidores reales mediante algún tipo de algoritmo de balanceo, el servidor responde al proxy y este envía las peticiones de nuevo al cliente. Esta alternativa es muy utilizada, aunque presente menos índice de escalabilidad y más sobrecarga en los equipos que RR-DNS o LVS. El proxy acaba por resultar un cuello de botella ya que éste abre 2 conexiones TCP por cada conexión que se realiza al Proxy, esta solución a nivel de aplicacion se puede realizar mediante programas como SWEB[3].

Por último, están las alternativas propietarias de empresas como CISCO, IBM, COMPAQ y demás empresas que tienen bastante código tanto a nivel de aplicación, kernel y hardware específico para este tipo de tareas.

A pesar de que el director LVS se comporta como un conmutador (*switch*) de nivel 4 no actúa como un Proxy inverso. El modo de actuar es bien distinto. El nodo director utiliza un kernel especial parcheado que permite el

[2] Es decir, el conjunto de nodo director y batería de servidores reales.
[3] `http://www.cs.ucsb.edu/Research/rapid_sweb/SWEB.html`

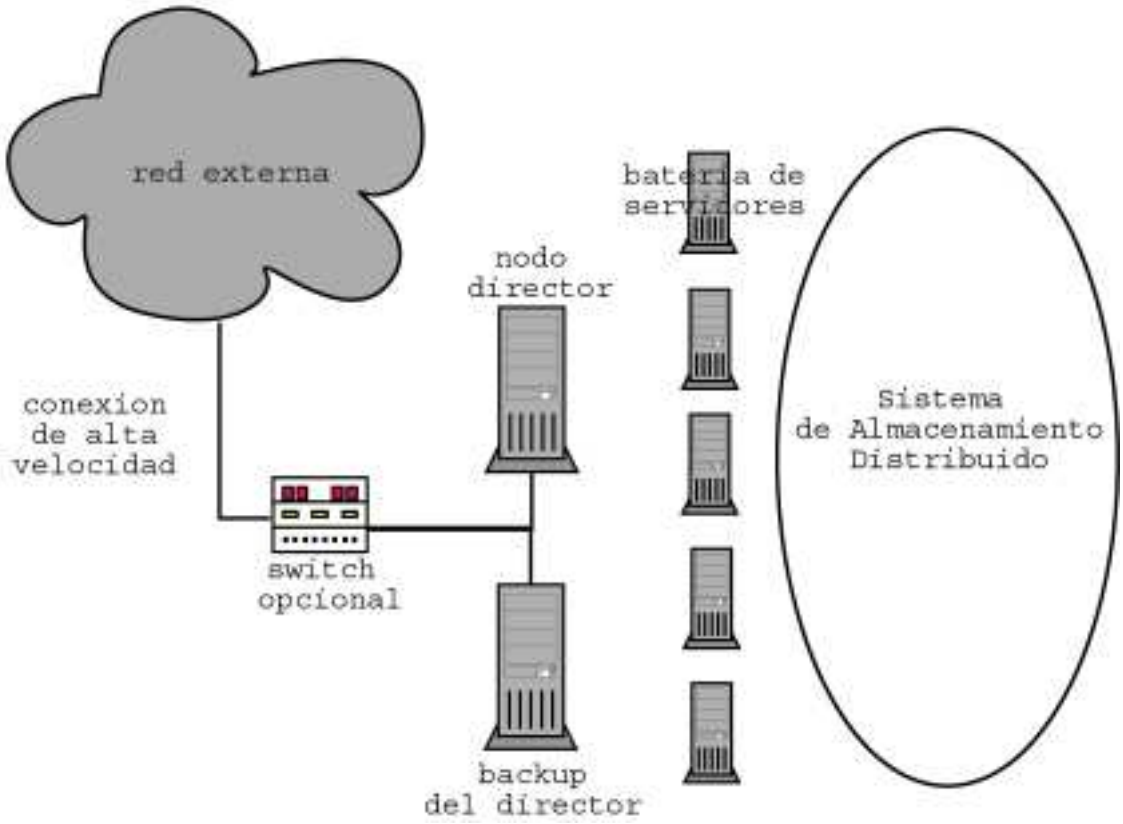



Figura 4.3: Clusters HA. Topología típica de un LVS básico

balanceo de carga de los servicios mediante varios métodos de planificación, además es configurable mediante un programa en zona de usuario que permite pasarle los parámetros al kernel acerca de como debe balancear conexiones. El director se comporta como un conmutador de nivel 4 en la arquitectura OSI, balancea conexiones o datagramas según se le haya exigido en su algoritmo de balanceo.

El efecto de solicitar una petición sobre el Servidor Virtual LVS es el siguiente:

1. el cliente solicita un servicio o conexión a la dirección del Servidor Virtual LVS (llamada VIP) que posee la interfaz pública del nodo director

2. el nodo director se encarga de balancear la conexión según el algoritmo programado, hacia el servidor real dentro de la batería de servidores

3. el servidor contesta al cliente con su respuesta y la envía hacia él.

De esta manera se puede ver que tanto cliente como servidor real trabajan de manera transparente en lo que se refiere al nodo director.

La diferencia con un Reverse Proxy estriba en que LVS no requiere de la vuelta de las peticiones al director, ya que éstas pueden ser enviadas directamente al cliente, lo que evita que el director se convierta en un cuello de botella en el sistema, como ocurre en el caso de los proxys inversos.

LVS puede solucionar muy satisfactoriamente casos de adaptabilidad a requerimientos o escalabilidad, redundancia, alta fiabilidad y mayor crecimiento de los servicios ofrecidos. Por todo esto se puede considerar dentro de los clusters de Alta Fiabilidad (HA).

Realmente LVS no es un cluster propiamente dicho. Un cluster se suele entender en base a una serie de máquinas que actúan de manera conjunta mediante relaciones entre ellas para resolver un problema (generalmente de cómputo). LVS no es un cluster en este sentido puesto que cada servidor es independiente y solamente

está relacionado con los otros mediante un sistema de almacenamiento común[4], los servidores solamente se relacionan con el nodo director para proveer de un servicio.

En cualquier caso y generalizando el concepto de cluster, LVS utiliza varias máquinas para aumentar el rendimiento y la fiabilidad de un servicio, es decir un problema, y como tal se puede considerar como un cluster. La idea de migrar sockets no tiene una solución general inmediata, MOSIX la solucionaba prohibiendo a los procesos servidores de cada máquina que migrasen a otros nodos, impidiendo un balanceo perfecto del sistema. El que un proceso que tiene un socket migre conlleva un problema de difícil resolución. LVS proporciona un sistema bastante adaptable para migrar sockets, por lo que es un sistema ideal y complementario a otros.

El nodo director se comporta como un router al que se le han añadido en el kernel tablas de encaminamiento para reenviar paquetes a los servidores reales para los servicios que se hayan configurado en el cluster LVS.

Existen tres maneras de efectuar el reenvío o encaminamiento en LVS: VS-NAT, VS-DR, VS-TUN.

- VS-NAT hace uso de NAT dinámico para efectuar transacciones entre servidor real y cliente.
- VS-DR o VS-Direct routing hace uso de direcciones virtuales mediante alias en dispositivos Ethernet para reenviar a la dirección Virtual del director (VIP) a cada servidor real.
- VS-TUN hace uso de *ip-tunneling* para reenviar los paquetes a los servidores reales, esto implica que el sistema operativo deba poder manejar la desencapsulación de estos paquetes especiales.

**Métodos de balanceo IP**

En esta sección se describirán las técnicas descritas anteriormente con las que LVS balancea los paquetes TCP/IP o UDP/IP hacia los servidores reales. Estas tres técnicas son bien distintas y permiten configurar LVS de una manera específica para cada solución que se quiera implementar. La elección de una u otra técnica depende principalmente del servicio que se quiera proveer, los conocimientos que se posean de los sistemas, el sistema operativo de los servidores, los recursos económicos que se estén dispuestos a gastar y consideraciones sobre el rendimiento.

**VS-NAT**

Es el caso más sencillo de configurar de todos y el que menor rendimiento tiene respecto a los otros dos.

VS-NAT hace uso de NAT para modificar direcciones, existen tanto la implementación para las versiones de kernel 2.2[5] como para las 2.4[6]. Ambas implementaciones dan soporte SMP para LVS en el nodo director (que es el que tiene el kernel modificado), lo que permite una tasa de manejo de paquetes muy alta para clusters que proveen de mucho servicio.

VS-NAT se compone de un director que corre el kernel parcheado con LVS (ipvs) y de una batería de servidores que pueden correr con cualquier sistema operativo y cualquier tipo de servicio. El nodo director se encarga de recibir las peticiones de los clientes por su VIP mediante el algoritmo de balanceo elegido se reenvían los paquetes a el servidor real de manera que el servidor responde a la petición y los encamina al nodo director, el cual cambia las direcciones de la cabecera de los paquetes IP del servidor, para que el cliente los acepte sin problemas. Como se puede ver, el mecanismo es muy parecido por no decir igual que el de un Proxy inverso, excepto por que el redireccionamiento se hace a nivel de kernel.

Primero el director reenvía sus paquetes mediante el código ipvs, modificando los paquetes que se recibieron del cliente mediante el cambio de la dirección destino hacia los servidores reales y luego vuelve a hacer el cambio inverso mediante NAT dinámico a los paquetes que envían los servidores. VS-NAT tiene el mismo problema que los proxys inversos: el nodo director llega a ser un cuello de botella en cuanto las exigencias por parte de los clientes se hacen muy altas, o el número de servidores internos a la red crece por encima de los 20. Es por esto que este tipo de configuración es la menos utilizada de las tres.

**VS-TUN**

[4] En el caso de que se quiera configurar asíÈn ciertos entornos ni siquiera es necesario este tipo de almacenamiento y basta con un *rsync* de los discos de cada servidor individual.

[5] implementacíon basada en la de *masquerading*, lo que hace que al mostrar todas las entradas mediante *ipchains -M -L* nos muestre las conexiones de LVS,

[6] que ha sido completamente reescrita para adaptarse al nuevo Netfilter, el cual no necesita incluir ninguna regla para poder gestionar bien los paquetes marcados por LVS.

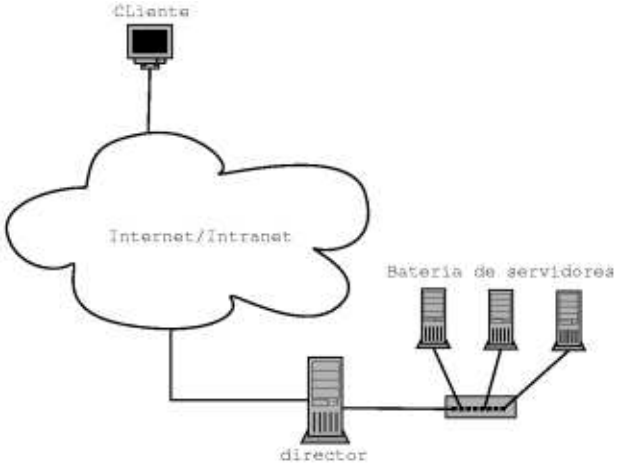



Figura 4.4: Clusters HA. Configuración VS-NAT

Este método es más utilizado que el anterior, se basa en redirigir los paquetes IP del nodo director al destino mediante técnicas de *IP-tunneling*, esto requiere que tanto el nodo director (que debe correr bajo Linux y por tanto puede ser compilado con *IP-tunneling*) como el servidor real puedan encapsular y desencapsular paquetes especiales. Para esto es necesario que la pila IP del sistema operativo lo soporte, y no todos los sistemas operativos lo soportan, en general la mayoría de Unix que existen en el mercado si lo soportan, por lo que en un principio no debe ser un grave inconveniente para la elección de este método como base de LVS.

El funcionamiento mediante este método de balanceo es el siguiente:

- el cliente hace la petición a la VIP del director
- el director elige mediante el algoritmo de balanceo cual será el servidor real que atienda la petición
- el servidor encapsula el paquete (que le llegó por la interfaz asignada a la VIP) en otro paquete IP con destino el servidor real

El servidor real atiende la petición de servicio y la responde, poniendo como dirección de los paquetes IP generados por este la dirección propia por la que llegó el servicio, es decir la VIP. Los envía directamente al cliente, sin tener que pasar los paquetes por el nodo director de nuevo. De esta manera se evita el problema de cuello de botella en el director.

Este mecanismo de redirección permite que los servidores reales puedan encontrar se no en una red local, como sucede en el caso de los dos anteriores, sino dispersos en otras redes a las cuales se pueda acceder desde el director, esto supone el poder distribuir los servicios no solo como método de incrementar el rendimiento, sino como distribución geográfica de los servidores, lo cual puede ser una ventaja para ciertos sistemas, o una desventaja para otros.

La desventaja de esta técnica está en que tanto el director como el servidor tienen que poder crear interfaces de tipo *tunneling*, y como consecuencia de hacer *IP-tunneling* siempre estará implícito un tiempo de procesador ocupado en encapsular o desencapsular los paquetes, que si bien en algunos sistemas puede ser insignificantes, en otros puede ser decisivo para la elección de otro método de balanceo.

**VS-DR**

VS-DR se basa en una tecnología de red local (en un único segmento) y en un cambio de direcciones IP-MAC para proporcionar el método de reenvío de los paquetes.

Al igual que VS-TUN no requiere reenviar los paquetes al nodo director, por lo que no presenta en él un cuello de botella. Es quizá el más utilizado de los tres, por ser el que mayor rendimiento obtiene, pero al igual que el resto, presenta una serie de desventajas en su uso y configuración.

El funcionamiento de VS-DR es similar al de VS-TUN en el sentido de que ambos utilizan la dirección VIP no solamente en el nodo director (donde esta la dirección VIP real a la que acceden los clientes) sino también en los nodos servidores. En este caso, los servidores poseen dos direcciones asociadas al nodo, una es la IP real

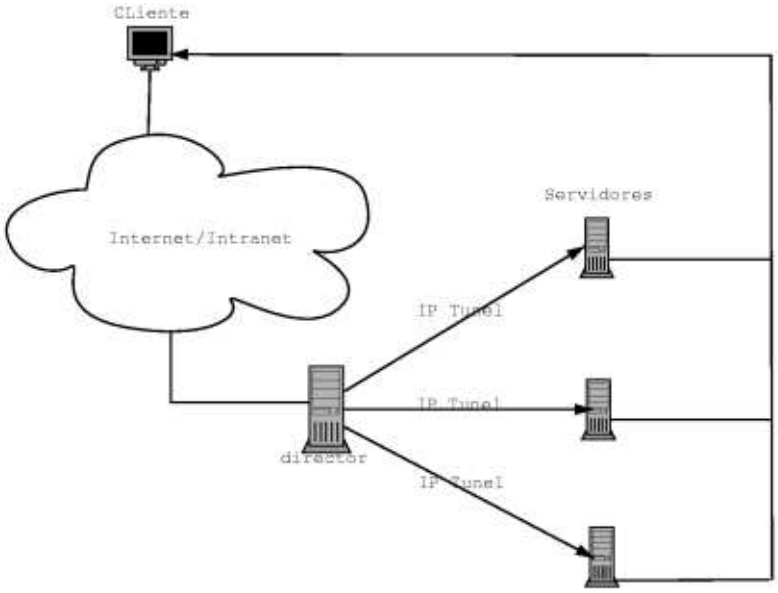



Figura 4.5: Clusters HA. Configuración VS-TUN

asociada a la tarjeta Ethernet, la otra es una dirección loopback especial configurada con la dirección VIP, es conveniente dejar la interfaz loopback que tiene la dirección `127.0.0.1` sin modificar, por lo cual se debe hacer un alias de esta interfaz pero con la dirección conocida como VIP.

De este modo los clientes hacen la petición a la VIP del director, éste ejecuta el algoritmo de elección del servidor, solicitando mediante ARP la dirección del servidor al que pretende enviar para conocer la dirección MAC asociada a esta IP. Una vez que la conoce envía un los paquetes del cliente, sin ser modificados, en una trama Ethernet con destino la dirección del servidor real. Éste recibe la petición y comprueba que pertenezca a alguna de las direcciones que él posee, como hemos configurado la VIP en un interfaz loopback, la petición se efectuará sin problemas.

Este método requiere en ciertas ocasiones que el servidor acepte las peticiones asociadas al interfaz declarado como `lo0:1`, es decir el de *loopback*, en el caso de ser una máquina Linux.

Esto implica generalmente que el servidor pueda ser configurado para ello, por ejemplo en el caso de apache (uno de los servidores más utilizados de HTTP), en el fichero de configuración */etc/httpd.conf* se debe especificar en una linea como

```
Listen <dir_VIP>:<PORT_VIP>
```

para que el servidor acepte las peticiones provenientes de esta interfaz. Esto puede resultar un inconveniente en ciertos servidores.

A pesar de ser el más utilizado por ser el que más alto rendimiento ofrece, está limitado en cuestión de escalabilidad debido a que la red sobre la que funciona está limitada a un único segmento ethernet por motivos de direccionamiento mediante ARP. Por otro lado no se necesita tiempo de encapsulación o desencapsulación de ningún tipo y tampoco ningún factor de redirección hacia el nodo servidor. El encaminamiento de los servidores reales a el cliente se puede hacer mediante otra conexión a red de alta velocidad de manera que el ancho de banda este garantizado.

**Características generales de los técnicas de balanceo**

Una vez vistos los tres mecanismos principales de las técnicas de balanceo se darán algunas consideraciones de carácter general acerca de las mismas.

Casi todas las implementaciones de LVS se suelen hacer con el cluster de servidores colocado en una red de área local, excepto las del tipo VS-TUN. Si disponemos de una conexión con el cliente de alto ancho de banda estaremos utilizando, en el peor de los casos, VS-NAT, y habrá más de 20 servidores reales en la red privada de 10 Mbps. Probablemente la red acabe congestionada con mucha asiduidad, provocando respuestas mucho peores

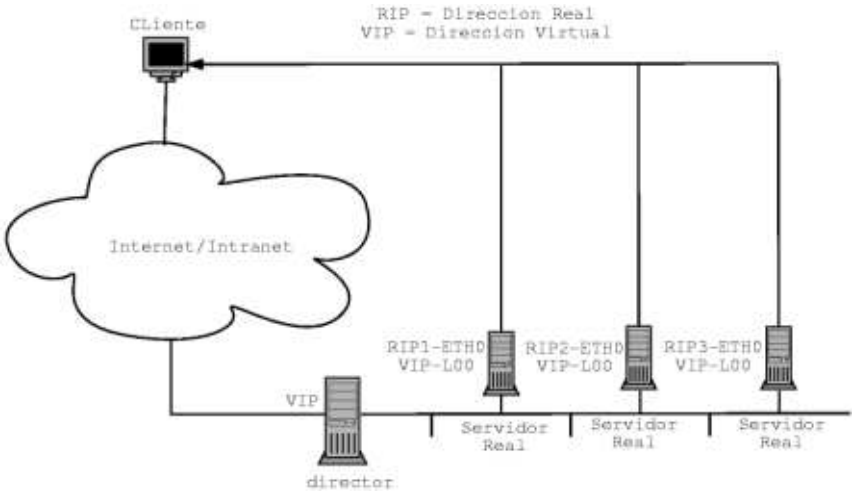



Figura 4.6: Clusters HA. Configuración VS-DR

de las que podría dar un servidor único, más caro.

Por otro lado está el factor de carga de los equipos. Cada servicio proporcionado por el servidor virtual puede tener como servidores reales destino un subconjunto de la batería de servidores. Esto implica que cada nodo debe ser convenientemente administrado y elegido con recursos y características correctas antes de la puesta en funcionamiento del LVS.

En el caso del nodo director sucede lo mismo, éste debe ser conveniente elegido para su cometido, el parche LVS no inhabilita el funcionamiento SMP del kernel de Linux[7] por lo que puede ser elegida una máquina de este tipo para hacer las funciones de nodo director. El funcionamiento de LVS se basa principalmente en *engañar* al cliente acerca de quién le está sirviendo. Así el cliente aceptará todos los paquetes que le vengan con la dirección VIP y determinados números de secuencia y asentimiento (en el caso de los TCP) con lo que solamente hay que elegir entre los diferentes mecanismos para poder llevar a cabo este cambio de direcciones: NAT, *tunneling* o mediante encaminamiento directo.

Otra de las desventajas que conlleva la instalación de un sistema LVS es la formación y el conocimiento con el que deben contar los diseñadores de la red y de cada sistema que intervienen en un sistema LVS. Al estar formado el sistema por un grupo hetereogéneo de elementos, en la mayoría de los casos con una relación de dependencia bastante fuerte, es necesario conocer extensivamente cada uno de los sistemas individuales, para que ninguno de ellos falle o baje su rendimiento. Por ejemplo es necesario saber el como hacer *masquerading* en el nodo director, como evitar ICMP Redirects en el director, como evitar los problemas ARP (se verá más tarde), como hacer auditorías a la red y gestionarla para ver donde tiene el sistema sus cuellos de botella y un largo *etcetera* de problemas potenciales que hacen que la puesta en marcha de uno de estos sistemas en entornos de producción sea más difícil de lo que en un principio pueda parecer.

### Aspectos Técnicos

La instalación de LVS requiere el conocimiento de cada uno de los elementos que requieren para su funcionamiento correcto (que no son pocos). La instalación o puesta en marcha se puede separar en:

- Diseño de la red. Direcciones, topología, servicios y arquitectura del sistema global.
- Preparación y configuración de los nodos directores, puesto que puede haber más de un nodo director haciendo backup.
- Preparación y configuración de los nodos que actuarán como servidores reales.
- Configuración de los elementos de encaminamiento (enrutadores) y seguridad (firewalls[8]).

[7] Si bien está reconocido que el rendimiento del funcionamiento SMP en el so Linux no es de los mejores, aunque se está trabajando en ello obteniendo cada vez mejores resultados.

[8] No debemos olvidar que al utilizar un sistema tan hetereogéneo con tantos elementos, el sistema es más susceptible a debilidades, por lo que será necesario hacer una auditoria de seguridad y un estudio de cuales son los posibles puntos de entrada al sistema para taponarlos.

- Configuración de la parte que proveerá al sistema de alta fiabilidad y redundancia.
- Configuración de un sistema de almacenamiento compartido.

Se podrían añadir otros tantos puntos para cada sistema específico. Pero para un sistema general en producción, los imprescindibles son los ya enunciados.

**Apartado de Red**

El diseño y elección de la topología de red es bastante crítica en la implantación del sistema ya que puede convertirse en uno de los cuellos de botella.

Generalmente los nodos servidores se encuentran en una red privada Ethernet a la que pertenecen tanto la batería de servidores como los nodos que actúen como director en el sistema. Hay que tener en cuenta que por esta red pasaran:

- las peticiones y respuestas de los clientes en el peor de los casos (VS-NAT),
- los paquetes de monitorización,
- paquetes de sistema de manejo y gestión del cluster, *etc*...

Es decir, una carga alta dependiendo de el sistema elegido.

Lo ideal sería utilizar tecnologías rápidas pero baratas como FastEthernet (100 Mbps) y separar en tantas redes como sea necesario para tener los mínimos cuellos de botella posibles y la máxima escalabilidad (en vista al futuro).

Esto probablemente implica separar por un lado la red por la que se efectúan todos los intercambios de petición entre el nodo director y los servidores reales, una red específica para el sistema de almacenamiento elegido y excepto en el caso de VS-NAT una red (o varias dependiendo del ancho de banda requerido para proveer servicio) para la salida de las respuestas de los servidores reales. Aun quedarían los paquetes de monitorización o redes destinadas a openMosix o similares.

Por último se podría utilizar una red más barata entre el nodo director y su nodo de backup, en el caso de utilizar Heartbeat[9].

Esta trama de redes no es necesaria en todos los sistemas, bastaría medir la congestión en la red para poder ver que servicios se deben separar de los otros para evitar las congestiones en cualquiera de los casos.

En lo referente al direccionamiento, generalmente solamente el nodo director debe tener una dirección pública, mientras que el resto de los nodos servidores suelen tener direcciones privadas reservadas. El mecanismo de direccionamiento utilizado en Ethernet provoca un problema que se tratará más tarde. En el caso de los servicios proporcionados. En la mayoría de los casos no interviene ningún factor externo en la configuración salvo casos como el explicado en el apartado anterior en el que se configura un servicio para un determinado interfaz de la máquina servidora. El único problema que plantean los servicios en un cluster del tipo LVS es el problema de la persistencia.

Se entiende por conexiones persistentes dos tipos de conexiones:

- una es del tipo de los servidores HTTP, en los cuales intervienen los tiempos de establecimiento de conexión activa, el tiempo de transferencia real y el tiempo de aproximadamente 15 segundos (aunque es muy variable) hasta que se cierra la conexión con el cliente.

  El manejo de esta persistencia es realmente sencillo.

- otro tipo de persistencia es la debida a dependencias entre varios sockets, como es el caso de servicios como FTP o el paso de HTTP a HTTPS en una transacción o compra vía Internet. Dichas conexiones son más difíciles de analizar, ya que dependiendo del servicio se establecerán factores de dependencia entre unos puertos u otros.

[9] Este caso es estudiado con más profundidad en otro apartado, y será conveniente referirse.

LVS resuelve el problema de la persistencia de manera explícita, es decir, de la manera que el administrador le obligue a hacerlo.

En cualquier caso la configuración, diseño y elección de red dependen de los servicios, situación geográfica, coste y topología. También dependen de la seguridad del sistema. LVS tiene un apartado especial para poder hacer balanceo sobre firewalls, de manera que los firewalls que ejecutan las mismas reglas pueden actuar en paralelo, utilizando de este modo máquinas más baratas para dicha tarea. A parte de este servicio, en un entorno de producción será necesario aislar la red completamente del exterior mediante las reglas adecuadas en los firewalls. La utilización de los nodos *diskless*, proporciona como se verá más tarde, un mecanismo más sólido para la construcción de nodos.

Por último se deben considerar todos aquellos factores relativos a la seguridad en redes que se puedan aplicar en servidores normales, de los que se encuentra amplia bibliografía.

**Consideración respecto al tipo de encaminamiento elegido.**

En lo que se refiere al rendimiento de cada tipo de encaminamiento es obvio que hay que separarlo en dos, por un lado VS-NAT y por otro VS-TUN y VS-DR.

VS-NAT se basa en el NAT tradicional, esto implica que cada paquete ya sea de conexión o de cualquier otro tipo que entra por parte del router o cliente es evaluado de la forma antes descrita y reenviado al servidor real. La cabecera IP es modificada con los valores fuente del nodo director, el servidor procesa la petición y da su respuesta al nodo director, que otra vez hace NAT al paquete colocando como destino el nodo cliente, y se vuelve a enviar al router por defecto del director.

Este mecanismo exige no solamente el balanceo de las conexiones por parte del kernel, sino un continuo cambio de direcciones a nivel IP de todos los paquetes que pasan por el director, así como la evaluación de los mismos, lo que hace que el rendimiento de este método se vea limitado a la capacidad de proceso que posea el nodo director. Por tanto es muy fácil dimensionar mal las capacidades del director o de la red y hacer que la salida a través del director se convierta en un cuello de botella. En el caso de el VS-NAT existe una opción en el kernel para optimizar el reenvió de paquetes para que este sea más rápido, que consiste en no comprobar ni checksums ni nada, solamente reenviar.

En el caso de VS-DR o VS-TUN el rendimiento no cae tanto sobre el director. No obstante éste sigue siendo un punto crítico, pero las soluciones actuales en lo que se refiere a capacidad de proceso nos sobran para gestionar cantidades considerables. El mayor problema de estos dos tipos es el diseño de la red interior, para que esta no se congestione y por supuesto el problema ARP enunciado en el apartado anterior.

En ambos métodos, las respuestas de los servidores reales se envían directamente a los clientes a través de un enrutador que puede ser (o no) diferente del que envío la petición inicial al director. En el caso de VS-DR se exige, por el modo del algoritmo que todos los servidores reales pertenezcan al mismo tramo de una LAN, esta es la forma más común en entornos de producción de configurar LVS, ya que permite el mejor rendimiento y menos carga de la red de los tres.

**El problema ARP**

Otra consideración que afecta no solamente a la red sino a la configuración del sistema en general es el problema ARP. Como se ha visto hasta ahora la técnica de balanceo más utilizada es VS-DR, esta técnica ya ha estado descrita.

Para configurar LVS se puede hacer mediante *tunneling* o mediante DR, en cualquier caso habrá que añadir direcciones virtuales extras a la configuración de los servidores reales, asícomo a la del nodo director. Para esto se deberán compilar los núcleos de las máquinas con la opciones de `ip aliasing` en el apartado *Networking*.

En la figura se muestra el problema ARP.

Cuando un cliente quiere realizar una conexión con el director lanza un paquete ARP (o en su defecto lo manda el router, en cualquier caso el problema es el mismo) para saber cual es la dirección MAC del director. Éste hace el reenvió dependiendo del algoritmo de planificación y al mismo tiempo guarda en su tabla el estado de la conexión que se ha realizado.

El servidor contesta y se reenvía la respuesta hacia el cliente. Pero ¿qué sucede si en lugar de contestar la máquina que hace las veces de director contesta una de las que esta haciendo de servidor? En este caso al lanzar el paquete ARP pidiendo *Who has VIP tell cliente/router?* en lugar de contestar el director contesta una de las máquinas que hace de servidor real. De esta manera la conexión se establece pasando por encima del LVS en una

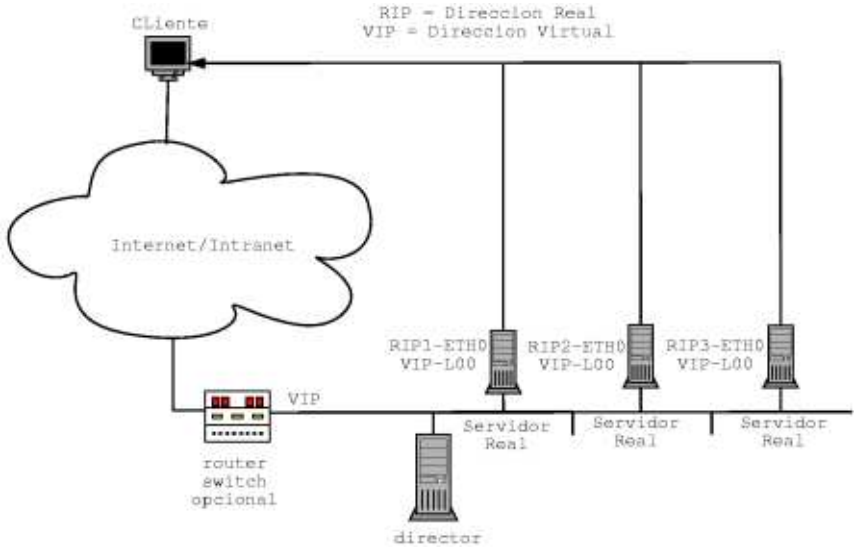



Figura 4.7: Clusters HA. Problema del ARP

especie de bypass, sin tener en cuenta al nodo director, que no guarda los estados ni la conexión.

De hecho hasta aquí no se produce ningún error con respecto a el servicio realizado, ya que el servidor procede a la entrega de la respuesta, el problema esta en el caso de que la tabla ARP del router o cliente se borre[10] y vuelva a pedir ARP *Who has VIP tell cliente/router* concediéndose la MAC de otro servidor o incluso del nodo director, que no tiene ni idea de como lleva la conexión en ese momento ya que el no la estaba sirviendo. Como se puede ver es un problema inherente a la manera en la que hemos configurado nuestra solución que podría ser resuelto poniendo más de una tarjeta de red al director u otras soluciones. En cualquier caso, la red se comporta normalmente de manera aleatoria en la concesión de la MAC.

En cualquier caso la solución a este problema esta en conseguir que el director conteste a los paquetes ARP *Who has VIP tell cliente/router*, que son solicitados por el router o cliente.

Existen varias soluciones, desde parches para el kernel hasta un `ifconfig -arp`, pasando por meter una tarjeta de red más en el director o incluso modificando la tabla ARP del router para que no solicite ARP dinámicamente sino lo especificado.

Por ejemplo en */etc/ethers* o directamente cambiando la tabla ARP de manera que todos los paquetes IP del router para la dirección VIP queden asociados al director. Cualquier solución es válida.

Al mismo tiempo exige un conocimiento bastante profundo del problema para los diseñadores de la red, ya que cada solución aportar unas ventajas y alguna desventaja. Generalmente se suelen utilizar dos opciones: añadir nueva tarjeta al director o adaptar el kernel de los clientes para que ellos no respondan a las peticiones ARP de alguna de sus interfaces, accediendo a */proc/sys/net/ipv4/lo/hidden* o similar respecto al interfaz al que hemos asociado la VIP.

Hay que tener en cuenta que la especificación de *-arp* en el *ifconfig* se deshecha en kernels superiores al 2.0.* y exige el otro tipo de soluciones. Otra solución que se propone es modificar la tabla ARP que se encuentra en el cliente o router que hace el *Who has VIP tell router*, pero esto crea alguna complicación a la hora de tener que controlar la caída del nodo director de *backup* en el caso de que exista (caso muy probable) y la toma de sus funciones, ya que la MAC del nodo director actual cambia y por tanto debe cambiar la tabla MAC-IP configurada en el router.

**Configuración y elementos que componen el nodo o nodos directores**

El nodo director es uno de los puntos más críticos del sistema, por eso debe ser bien elegido y configurado para las tareas que debe hacer. Suele tener algún mecanismo de alta fiabilidad con un servidor replica que toma las funciones del nodo director cuando este cae de manera casi transparente[11] al sistema. La puesta a punto del

[10] La asiduidad de este efecto depende del sistema operativo utilizado, de hecho es configurable, pero suele rondar entre los 2 y 10 segundos.

[11] El casi se debe a que de momento no se provee de ningún mecanismo para pasar las tablas de conexiones establecidas y monitorizaciones.

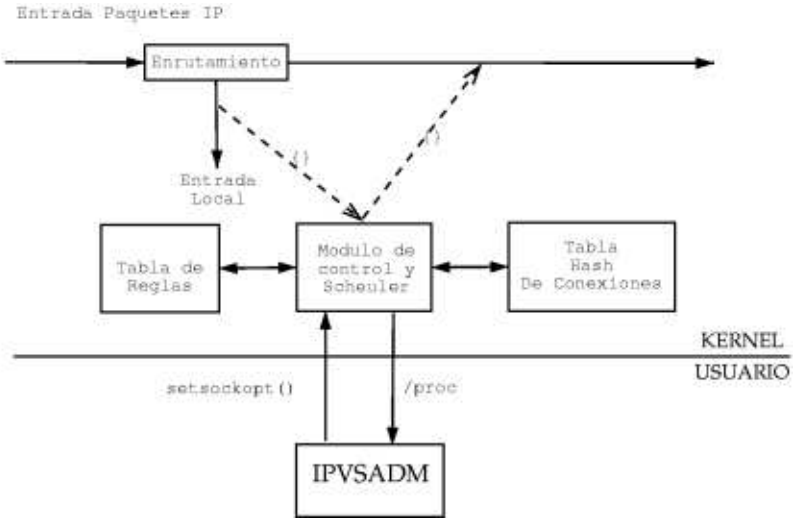



Figura 4.8: Clusters HA. Funcionamiento del kernel LVS

nodo director (dejando a un lado el nodo réplica) esta formada por la configuración de varias de las herramientas de las que explicamos antes.

Lo primero es encontrar el código de lvs, este código se puede encontrar en la pagina oficial LinuxVirtualServer.

El paquete a descargar depende del kernel que se utilice, de esta manera se pueden elegir dos tipos de kernel donde instalarlo: la serie antigua (2.2) y la nueva (2.4). La manera de configurarlos es distinta.

El paquete LVS contiene más programas necesarios para la instalación de LVS y algunas herramientas de ayuda como el script configure del que hablaremos más tarde. El kernel del director debe ser parcheado, con el código de LVS una vez puesto el parche al kernel y compilado, se ejecuta y se configura mediante un programa en zona de usuario llamado *ipvsadm* que permite especificar el comportamiento del director.

Antes de seguir avanzando con la instalación de el kernel es necesario conocer, al menos por encima, el funcionamiento del núcleo que vamos a utilizar. Ver figura.

El núcleo del nodo director será el que se encargue de establecer, mediante un algoritmo de balanceo, quien será el servidor real de cierta conexión. Este algoritmo se encuentra como código del núcleo y es elegido antes de compilar el kernel ya sea como modulo o como código interno al kernel.

El director hace lo siguiente:

- las conexiones que se reciben del cliente (al interfaz que tiene la dirección virtual) son insertadas en el núcleo de decisiones de LVS,
- este núcleo se encarga de hacer la planificación mediante un mecanismo programado (algoritmo de balanceo)
- luego se insertan dichas conexiones en una tabla hash o tabla de dispersión.

Todos los paquetes pertenecientes a conexiones realizadas son comprobados en su tabla hash para ver qué servidor real estaba gestionando la transacción y poder enviar los paquetes a dicho servidor. Cada entrada en la tabla hash ocupa 128 bytes y el número máximo de conexiones que el director puede realizar se puede configurar antes de la compilación del kernel, lo cual implica un estudio previo de las conexiones que se estima que se podría realizar al director para poder adecuar este a las necesidades del sistema, proveerlo de suficiente memoria para que no haya problemas con el número de conexiones[12].

Mediante la elección de las técnicas de balanceo ya vistas el núcleo se encarga de formar los paquetes IP que se enviarán a los nodos servidores.

[12]Las tablas hash de LVS asi como el código esta marcado como codigo de kernel, y no se puede hacer swap con ellas. Si se hiciese se llegaría a una bajada del rendimiento de LVS que haria que no mereciese la pena utilizar el sistema por las latencias de la conexión

| SERVIDORES | A | B | C |
|---|---|---|---|
| PESOS | 4 | 3 | 2 |

Cuadro 4.1: Clusters HA. Disposición de servidores con pesos en LVS

| RESULTADO | A | A | B | A B | C | A | B | C |
|---|---|---|---|---|---|---|---|---|

Cuadro 4.2: Clusters HA. Resultado de la elección

En la zona de usuario se ejecuta el programa **ipvsadm**. Este programa viene dentro del paquete ipvs, debe ser compilado para cada versión específica del parche de kernel ipvs. El programa se encarga mediante la función *setsockopt()* de configurar el modo de funcionamiento del sistema, y mediante el sistema de ficheros */proc* se encarga de leer las configuraciones que este tiene en cada momento.

Existen seis algoritmos para la elección del funcionamiento del nodo director. El algoritmo se selecciona en el kernel antes de la compilación, en el campo IPVS dentro de *Networking options*. Se pueden compilar como módulos o internos al kernel, se deben compilar todos los algoritmos que más tarde se utilizarán a la hora de encaminar a los servidores reales mediante la configuración de **ipvsadm**.

Se pueden distinguir dos familias de algoritmos, la dinámica y la estática. Los algoritmos a su vez se pueden separar en 3 tipos y sus derivados para pesos constantes.

- Round Robin y Round Robin con peso.
- Número de conexiones y número de conexiones con peso.
- Algoritmos basados en número de conexiones y localidad al nodo.
- Algoritmos relativos a cache-proxy (*squid*) y algoritmos relativos a *fwmark* (firewalls).

Los dos últimos juegos de algoritmos no serán enunciados, pero sirven para balancear carga entre *squid* y también para balancear entre varios firewalls. Para más información ver la documentación del kernel parcheado o el HOWTO de LVS.

El algoritmo de Round Robin es el habitual en el mundo de la informática. Es un algoritmo estático y su ampliación a Round Robin con peso solamente implica que a los nodos más potentes se les asigna más cantidad de trabajo que a los menos potentes. En cualquier caso, Round Robin no es un algoritmo óptimo para balancear la carga cuando hay un elevado numero de conexiones.

El algoritmo RR es semejante al utilizado en RR-DNS, pero en el caso de LVS la granularidad es mucho menor ya que se balancea por conexión. Un ejemplo de el algoritmo de Round Robin con peso, podría ser el caso de tener 3 servidores:

Como se puede ver las conexiones son balanceadas de manera alterna siguiendo un algoritmo sencillo de decremento del peso y nodo que lleva más tiempo sin ser utilizado. Esta es una manera muy sencilla de balancear la carga pero generalmente implicará que el balanceo no sea perfecto (en el caso de que el número de conexiones sea muy alto).

El algoritmo por número de conexiones (o *least connections*) y *wheigthed least-connection* responde a un algoritmo dinámico. Se basa en enviar las nuevas conexiones al nodo que tenga menos conexiones abiertas, esto requiere saber en todo momento cuantas conexiones abiertas tiene cada nodo, pero supone una ventaja en el comportamiento del sistema cuando el número de conexiones es elevado.

El algoritmo, que es sensible a los pesos, actúa dependiendo de los pesos que cada nodo tiene asignados en el director y se balancea según el número de conexiones abiertas[13]. Este grupo de algoritmos balancea mejor la carga en lineas generales pero requiere un nodo director más potente para llevar el conteo del número de conexiones de cada nodo.

El algoritmo contador de conexiones con peso sería el siguiente.

[13] Aquí el problema de las conexiones residuales que están en un TIME_WAIT, un nodo puede tener más conexiones que otro pero estar todas en un TIME_WAIT y por lo tanto no consumir recursos, lo que implica que la carga no queda homogeneamente distribuida.

Existe una batería de servidores para un determinado servicio con N servidores, a cada uno se le asigna un peso $W_i$(i=1...N) y número de conexiones activas $C_i$(i=1...N).

El número total de conexiones **S** no es más que la suma de las conexiones activas de cada nodo. De este modo la siguiente conexión para el servicio sera para el nodo **j** que cumpla:

$$(C_j/W_j) = min\{C_i/W_i\}(\text{i= 1..N})$$

El único inconveniente es que el kernel no puede realizar esta operación, ya que no dispone de los números en coma flotante de forma que esta expresión debe ser cambiada a algo como $C_j * W_i > C_i * W_j$ de manera que la comparación pueda ser evaluada por el kernel. Lo más importante a tener en cuenta es que las conexiones con las que se trabaja son conexiones activas. Existen otros dos algoritmos más basados en el algoritmo *least-connection* que mejoran este en algunas situaciones, estos dos algoritmos se basan en la localidad de las conexiones, e intentan que el mismo cliente vaya, en caso de que el balanceo de la carga lo permita, al mismo nodo servidor que fue en otras conexiones realizadas por este cliente.

Una vez vistos los fundamentos teóricos para ver como debe compilarse el kernel puede pasarse a la instalación y compilación del mismo. Respecto a este punto se da por sentado que el administrador sabe aplicar parches al kernel y compilarlos de manera correcta.

Para empezar hay que obtener las fuentes del parche del kernel, el programa *ipvsadm* y el script *configure*.

Habrá que parchear el kernel (las versión de las fuentes del kernel deben coincidir con la versión de LVS), mediante el comando `cat <parche> | patch -p1` o `patch -p1 < <parche>`. Luego se deberán compilar con las opciones descritas anteriormente (para más información mini-Howto-LVS) y preparar el parche para las máquinas que vayan a actuar como director.

El nodo director puede ser una máquina sin discos que cargue su kernel a través de la red mediante TFTP (ver la sección referente a *Nodos sin discos*) para luego componer su directorio raíz mediante NFSroot. La opción de utilizar un diskette (o lilo) para ejecutar el kernel es la más utilizada.

Luego habrá que compilar e instalar las fuentes del programa **ipvsadm** y el sistema estará preparado.

A partir de aquí pueden hacerse dos tipos de configuraciones

- la primera exige un conocimiento del sistema LVS más profundo, ya que todos los parámetros se configuraran de manera manual mediante la orden **ipvsadm**
- la otra es utilizar el script en perl (para lo cual se debe tener instalado perl) para que configure los servidores y el director.

El programa **ipvsadm** servirá para especificar

- servicios y servidores que conoce y gestiona el director
- pesos de cada servidor respecto al servicio ofrecido
- algoritmo de gestión (de *scheduling*)
- añadir servicios y servidores
- quitar servicios
- también existen un par de programas al tipo de ipchains que permiten guardar configuraciones

Para configuraciones iniciales o sistemas complejos el script escrito en perl está preparado para varios sistemas operativos del tipo Unix, por lo que puede evitar más de un dolor de cabeza al configurar nuestro sistema. Este script *configure* obtiene la información de un fichero de texto (las fuentes contienen ejemplos de varios de estos ficheros) y genera otros archivos. Entre estos archivos están el script de inicialización de LVS preparado para ser incluido en la sección de comienzo *rc.d*, que configura desde las IP de las máquinas que se le pasaron hasta la tabla de rutas, pasando por los interfaces que no deben hacer ARP.

| Dirección IP | Significado |
|---|---|
| CIP | Dirección del cliente |
| VIP | Dirección virtual del servicio (en la red del cliente o accesible desde ésta) |
| DIP | Dirección del director en la red privada de servidores reales |
| RIPn | Direcció IP del servidor real |
| DIR-ROUTER | Resto de las direcciones pertenecientes a los routers |

Cuadro 4.3: Clusters HA. Relación de tipos de direcciones IP

También incluye los programas y archivos de configuración necesarios para luego utilizar en el programa de monitorización MON. Es una ayuda cómoda para los que no quieran tener que configurar a mano de manera extensiva su LVS[14].

Para utilizar *configure* simplemente hay que seguir los pasos que se ven en uno de los ficheros de configuración por defecto que viene en la versión que se se haya obtenido, configurar los parámetros del sistema y ejecutar `perl configure <fich_de configuración>`. Al estar el script en perl son necesarios que los módulos necesarios para ejecutarlo. La salida del programa será un script del tipo *rc.d* que puede ejecutarse en los servidores reales y en el director (el script sabrá donde se ejecuta).

Otro factor a tener en cuenta a la hora de configurar servicios mediante *ipvsadm* son las conexiones persistentes. Hay que atender este tipo de conexiones en casos como una sesión HTTPS, en protocolos del tipo multipuerto como FTP (en los cuales los puertos 20 y 21 corresponden al mismo servicio) o sitios comerciales, donde un cliente debe cambiar del puerto 80 de HTTP al 443 para enviar por HTTPS sus datos bancarios.

Este problema requiere de configuraciones específicas en el director para que los servicios sean balanceados al mismo nodo, saltándose el algoritmo de balanceo. El caso de las conexiones persistentes es otro de los casos (junto con el de ARP) que dificulta la configuración de los servicios, pero al mismo tiempo es impensable un sitio en Internet que no vaya a proveer conexiones seguras a través de HTTPS o que simplemente tenga un FTP. En cualquier caso, la orden *ipvsadm* permite configurar este tipo de conexiones persistentes de manera bastante cómoda.

**Instalación y configuración de un caso de estudio**

---

*A continuación se adjunta un caso práctico de instalación de algunos servicios hasta ahora expuestos.*

*Corresponde a una de las experiencia prácticas que Carlos Manzanedo y Jordi Polo realizaron en su proyecto de fin de Ingeniería Informática en el año 2001 en la Universidad de Alcalá de henares (Madrid).*

---

Al realizar la instalación nos hemos encontrado con otros problemas debidos mayormente a la topología y diseño de la red utilizada. En la figura mostramos el esquema de nuestra imaplementación (VS-NAT y VS-DR) para probar LVS con servicios no persistentes como HTTP y telnet.

El nodo director es arrancado sin discos y esto no supone ninguna diferencia. La red interna pertenece a 10.0.0.0/24 y la externa a 192.168.10.0/24, el nodo director pertenece y encamina a las dos redes. Se ha utilizado un *hub* y cable coaxial para la conexión. Para trazar paquetes en la red se utilizaron herramientas GPL como *ethereal* o *tcpdump* (suelen venir en la distribución habitual que utilices) y generaciones de filtros para facilitar su uso.

Configuramos LVS con LVS-NAT, de manera que en un principio no nos tuvimos que preocupar del problema de ARP, que se puede evitar con

```
echo 1 > /proc/sys/net/ipv4/conf/lo0:1/hidden
```

para que no mande respuestas ARP).

El funcionamiento es el siguiente[15]:

[14] Si bien no evita el tener que conocer de manera extensa el sistema.

[15] Para más detalles puede comprobarse mediante *ethereal* y *tcpdump*.

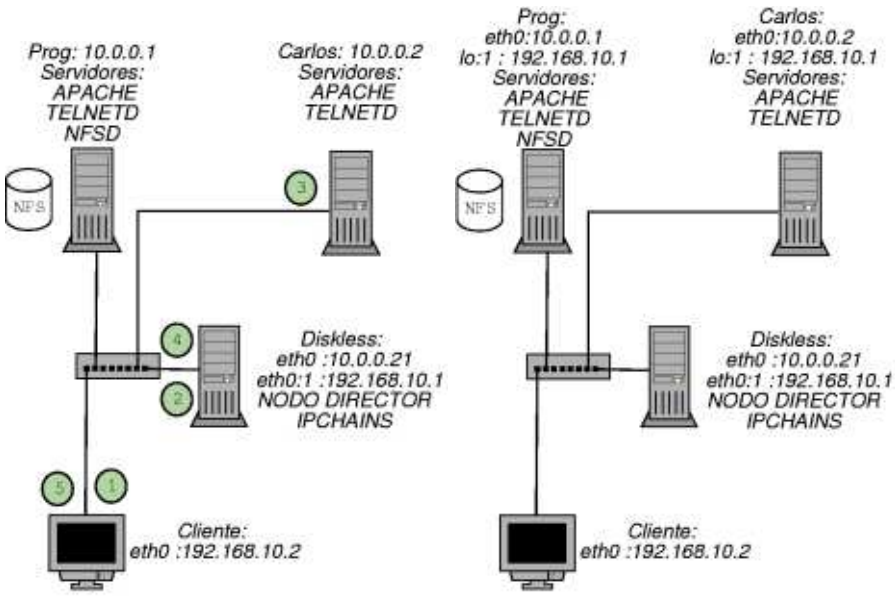



Figura 4.9: Clusters HA. NAT y DR un caso práctico

1. el cliente solicita una conexión con algún servicio de VIP address, es decir ladirección virtual de los servicios),

2. crea algún paquete de nivel cuatro ya sea TCP o UDP que envía al director VIP,

3. éste reenvía el paquete al nodo servidor real que decida el algoritmo de encaminamiento pero cambiándole el dato destino (se cambia por la dirección del servidor real),

4. el paquete o petición llega al destino, es atendido por el servidor de dicho servicio (si existe) y se genera la respuesta que se envía al cliente.

En este punto es donde es necesario activar *forwarding* (router) en el director, y aún más, debe estar configurado NAT dinámico para enmascarar todas las respuestas que salen de los servidores reales cuando salen hacia el cliente.

Quizá la explicación sea algo liosa así que pongo un dibujo y explicación por puntos.

1. Cliente envía petición a VIP del director CIP->VIP

2. Director LVS consulta tabla-HASH para ver si hay conexión, si no la hay la crea según el algoritmo de *scheduling* del servicio.

3. Director envía la petición a el servidor elegido cambiando la dirección fuente CIP->RIP (paquete replica de el anterior CIP->VIP).

4. El servidor-real propio del servicio contesta mandando RIP->CIP a su gateway por defecto, es decir en nodo director DIP.

5. El director actúa como router-NAT y cambia el paquete de RIP->CIP a VIP->CIP y se lo envía al cliente.

En estos pasos no he tenido en cuenta todas las peticiones ARP pertinentes. Estas peticiones pueden parecer que en un principio no tienen ningún efecto, pero uno de los problemas que hemos tenido en la configuración es que la cache de direccionamiento del kernel (así como la de ARP) contenían rutas que estropeaban el direccionamiento. Hemos tenido que limpiar estas caches mediante

```
echo 0 > /proc/sys/net/ipv4/route/flush
```

para *limpiar* las rutas, que ha sido lo que más ha molestado.

Para configurar esta red arrancamos el servidor que baja su kernel por bootp/tftp y configura su directorio raíz con nfsroot, una vez que tenemos todos los ordenadores encendidos.

**Configuración del sistema, relativa a IP's y tablas de rutas**

En esta explicación por puntos de las pautas seguidas para configurar de forma manual el sistema no entraremos en detalles en lo que se refiere a conceptos o configuración de NAT o ipchains, aunque se requiere un conocimiento mínimo de ambos para poder configurar el sistema. De hecho, en este caso de estudio se dará una configuración mínima para que el LVS funcione de manera correcta, pero se necesitara de más reglas añadidas a ipchains o Netfilter para proveer de un sistema seguro.

- cliente (Redmond95) ip 192.168.10.2/24 sin *gateway* ni DNS (uno de los problemas es que requería DNS para que funcionase el cliente de HTTP, a pesar de no utilizar en ningún momento nombres simbólicos. No sabemos exáctamente a qué se puede deber esto.)
- servidores reales
  - nodo1 - ip 10.0.0.1/24 rutas a su propia red y gateway por defecto 10.0.0.21
  - nodo2 - ip 10.0.0.2/24 rutas a su propia red y gateway por defecto 10.0.0.21
- nodo director.
  - `ifconfig eth0 10.0.0.21 netmask 255.255.255.0 up ESTA ES LA DIP`
  - `ifconfig eth0:1 192.168.10.1 netmask 255.255.255.0 up ESTA ES LA VIP`
  - `route add -net 192.168.10.0 netmask 255.255.255.0 dev eth0`
  - `route add -net 10.0.0.0 netmask 255.255.255.0 dev eth0`
- arranque de servicios en nuestro caso /etc/init.d/apache start en los dos servidores reales y configuración de telnet en */etc/inetd.conf*
- configuración del director:
  - primero hay que configurarlo como router para que haga NAT, es algo necesario, y se puede hacer mediante *ipchains* (como en nuestro caso) para los nuevos kernels, deben ser compilados sin compatibilidad hacia atrás con *ipchains*, es decir con *iptables/Netfilter*, y además no hace falta añadir ninguna regla ya que el soporte propio de NAT esta implementado en el kernel. En nuestro caso:
    - `echo 1>/proc/sys/net/ipv4/ip_forward`
    - `ipchains -A forward -p (UDP/TCP) -i eth0:1 -s localnet [PUERTO] -d 0/0 -j MASQ`

Esta cadena es susceptible de muchos cambios según el servicio que se quiera dar, por otro lado hay que recordar que se procurará que no pasaren los mensajes ICMP, entre ambas redes, así que las comprobaciones de destino (por ejemplo de RIP-¿CIP, se deberían hacer mediante traceroute). En un entorno de producción, esta cadena debe ser colocada según los requerimie ntos de seguridad del sistema, se debe permitir NAT, pero no descuidar la seguridad del sistema.

- Por otro lado, generalmente (y este ha sido otro problema a descubrir con ETHEREAL y TCPDUMP) los routers tienen habilitado el servicio ICMP-Redirects , que envían paquetes ICMP para que el cliente reenvíe el mismo mensaje a dicha IP. SE DEBE DESHABILITAR ESTE COMPORTAMIENTO, para poder deshabilitarlo, hacemos:
  - `echo 0 > /proc/sys/net/ipv4/conf/all/send_redirect`
  - `echo 0 > /proc/sys/net/ipv4/conf/eth0/send_redirect`
  - `echo 0 > /proc/sys/net/ipv4/conf/default/send_redirect`

**Configuración del director con ipvsadm**

Con esto queda asegurado el buen funcionamiento NAT del servicio. En este punto es necesario probar el estado de la red de manera que se comporte según se espera, es mejor ver el comportamiento en un caso simple de fallo que en uno complejo, esto se puede probar haciendo un *traceroute* desde los servidores a el cliente y comprobar que se llega a el en dos saltos.

Una vez comprobado esto, podemos pasar a la configuración del nodo director para que actúe de la forma requerida mediante VS-NAT en un primer caso. Con la orden *ipvsadm*. Lo más importante es leer el manual de este programa que es realmente simple de configurar. En nuestro caso, hemos configurado los servicios telnet y www para ambos servidores con distintos algoritmos de balanceo.

```
Luis:~# ./activa
Luis:~# chmod 760 config
Luis:~# ./config
Luis:~# cat activa
#!/bin/bash

echo 1 > /proc/sys/net/ip4/ip_forward
echo 0 > /proc/sys/net/ip4/conf/all/send_redirects
echo 0 > /proc/sys/net/ip4/conf/default/send_redirects
echo 0 > /proc/sys/net/ip4/conf/eth0/send_redirects
ifconfig eth0:1 192.168.10.1 netmask 255.255.255.0 up
ifconfig eth0 10.0.0.21 netmask 255.255.255.0 up
ipchains -A forward -i eth0:1 -p tcp -s 10.0.0.0/24 -d 0/0 -j MASQ

Luis:~# cat config
#!/bin/bash

ipvsadm -A -t 192.168.10.1:23 -s wrr
ipvsadm -a -t 192.168.10.1:23 -r 10.0.0.2:23 -m -w 1
ipvsadm -a -t 192.168.10.1:23 -r 10.0.0.1:23 -m -w 1
ipvsadm -A -t 192.168.10.1:23 -s wlc
ipvsadm -a -t 192.168.10.1:23 -r 10.0.0.2:80 -m -w 3
ipvsadm -a -t 192.168.10.1:23 -r 10.0.0.1:80 -m -w 1

Luis:~# ipvsadm -L -n
IP Virtual Server version 1.0.5 (size=4096)
Prot LocalAddress: Port Scheduler Flags
  -> RemoeAddress:Port               Forward Weight ActiveConn InActConn
 TCP 192.168.10.1:23 wrr
  -> 10.0.0.0.1:23                  Masq    1      0          0
  -> 10.0.0.0.2:23                  Masq    1      0          0
 TCP 192.168.10.1:80 wlc
  -> 10.0.0.0.1:80                  Masq    1      0          0
  -> 10.0.0.0.2:80                  Masq    3      0          0
```

Con estas líneas queda configurado, podemos ver conexiones y estados mediante varios programas como:

- `ipvsadm -L -n`, muestra el estado de LVS
- `ipchains -L -M`, muestra conexiones enmascaradas
- `netstat -c -M`, muestra estado de conexiones enmascaradas de manera continua

Para hacer la prueba en el cliente, con un navegador conectamos con 192.168.10.1 en unas cuantas ventanas. Como el sistema de ficheros no es compartido en nuestro caso, pudimos ver como en algunas ventanas aparecía la página principal de un servidor y en otros casos (menos cantidad de ellos) la del servidor de menos peso.

Con el caso de `telnet` no sucede lo mismo. Para cada nueva conexión alterna entre uno y otro servidor, propio del algoritmo de RR. Para poder hacer estas comprobaciones hay que deshabilitar la cache del navegador de manera que este realice una nueva conexión cada vez que se haga la petición de la página.

Existen herramientas preparadas para medir el rendimiento del un servidor web, como `polygraph`. Existen diversos programas para probar como es el rendimiento de el servicio de los servidores o hacer un montón de conexiones desde un mismo cliente a nuestro servicio de LVS.

El rendimiento mejoraría seguro si en lugar de configurar nuestros servidores con NAT los configuramos con DR, para lo cual casi no tenemos que tocar nada, *solamente* deshabilitar NAT añadir interfaces lo con la dirección VIP y configurar un router entre los servidores reales y el cliente externo. En los servidores reales hacer un:

```
echo 1 > /proc/sys/net/ipv4/conf/lo/hidden
```

Para evitar el problema de ARP y configurar el nodo director de la siguiente manera:

```
ipvsadm -A -t 192.168.10.1:80 -s wlc
```

Crea servicio www asociado a la dirección VIP balanceado mediante weight least-connectios

```
ipvsadm -a -t 192.168.10.1:80 -r 10.0.0.1:80 -g -w 1
```

Añade servidor a www con peso 1

```
ipvsadm -a -t 192.168.10.1:80 -r 10.0.0.2:80 -g -w 3
```

Añade servidor a www con peso 3

```
ipvsadm -A -t 192.168.10.1:23 -s rr
```

```
ipvsadm -a -t 192.168.10.1:23 -r 10.0.0.1:23 -g -w 1
```

```
ipvsadm -a -t 192.168.10.1:23 -r 10.0.0.2:23 -g -w 1
```

El parámetro `-g` en la configuración de los clientes implica que la técnica de balanceo sera a traves de gateway, es decir mediante Direct Routing o encaminamiento directo. Solamente con cambiar el modo de encaminamiento en los servidores, que pasa a ser vía gateway (propio de DR), queda todo preparado. Por otro lado en la tabla de rutas de los servidores tiene que estar configurado el encaminador por defecto, y en caso de haber firewall éste debe dejar pasar los paquetes que vayan de VIP->CIP generados por la interfaz virtual de los servidores reales.

Como últimas conclusiones al apartado de configuración de LVS hemos de añadir que en las pruebas realizadas en nuestro caso, mediante la topología y diseño visto arriba sobre un solo tramo Ethernet y con MOSIX instalado y funcionando en los servidores reales (es decir CARLOS y PROG) y la máquina diskless accediendo mediante NFS a todos sus archivos, el rendimiento de la red es pésimo. Se producen colisiones contínuamente, lo que provoca errores y retransmisiones continuas.

Esta continua congestión, debida a que la red sea lenta (10 Mbps), deberá tenerse en cuenta seriamente el diseño de nuestra red al completo para que esta no se convierta en el cuello de botella del sistema.

Los programas sniffers *ethereal* y *tcpdump* nos han ayudado a detectar los problemas generados por ARP así como el problema de inconsistencias de la cache o el envío de ICMP_REDIRECTS, que hacían que las primeras pruebas fallasen continuamente. De esta manera, mediante filtros, se traceaba la red con fines de ver cual era el cambio de comportamiento respecto al explicado teóricamente en los apartados anteriores, y adecuarlo a las exigencias del método elegido en LVS.

#### ¿Y la alta disponibilidad?

Llegados a este punto, tenemos nuestro servidor LVS funcionando, pero ¿qué sucede en el caso de que uno de los servidores o directores falle? ¿cómo se comporta el sistema?

De la manera que lo tenemos configurado hasta este punto, un fallo en cualquier a de los nodos sería fatídico en el sistema. En el caso de que el fallo estuviese en uno de los nodos servidores, el nodo director intentaría

reenviar los paquetes del cliente al servidor, de manera que obtendría fallo después de un tiempo, al estar el nodo o servicio caído.

En el caso del servidor el problema sería aún mayor, ya que produciría la pérdida total del servicio. La intención de cualquier sitio en Internet no es sólo proveer a sus usuarios de servicio durante algún tiempo, el servicio debe estar funcionando en lo que se denomina en el argot técnico-empresarial 24x7, es decir 24 horas al dia 7 días a la semana. Es aquí donde el proyecto LVS deja de manos de otras soluciones el proveer de alta fiabilidad al sistema. En un principio se recomienda utilizar ciertas herramientas con las que se han configurado varios sistemas como pueden ser:

- Piranha
- LVS-GUI + Heartbeat + Ldirectord
- MON + Heartbeat

Pero sirve cualquier otra opción que nos provea de algún método de mantener los servicios proporcionados en alto ante el mayor número de eventos catastróficos.

Nosotros hemos decidido utilizar MON y heartbeat para nuestro sistema, como ejemplo de configuración para cluster de alta fiabilidad HA, en este apartado explicaremos cómo hemos configurado MON y qué es este programa.

Para monitorizar los servidores reales, configuramos MON de la manera que se explicará en el nodo o nodos servidores, de manera que se éste continuamente monitorizando servidores o servicios, para controlar en todo momento el estado del cluster. Para asegurar la continuidad de un nodo servidor en el sistema se puede utilizar también MON, pero es mejor utilizar Heartbeat para hacer la replica de este servidor, y en caso de estar en un entorno de producción utilizar conexiones Ethernet y serial para asegurarnos de que la comunicación entre los directores es continua.

Para controlar la caída de los servidores reales podemos pensar en un principio como solución de aproximación, un programa que haga pings cada cierto intervalo a los servidores y compruebe si éstos están en la red. Este programa lanzaría la ejecución de la orden *ipvsadm* con los parámetros adecuados para quitar o introducir al servidor en la tabla del nodo director. Otra opción es utilizar el protocolo y sistema de gestión de redes SNMP para comprobar en cada momento mediante peticiones snmpget, sobre la MIB de los agentes (en este caso los servidores) si estos proveen de los recursos necesarios a la red, por ejemplo espacio de disco duro, un proceso servidor, o incluso la carga y funcionamiento de las interfaces. Este sistema es en un principio el más adecuado para la gestión de nuestro cluster, pero tiene ciertas desventajas que evitan que sea utilizado. Entre estas desventajas se pueden contar:

1. **Inseguridad en la que esta actualmente SNMP.**

   Hasta que no exista implementación en Linux de SNMPv3, ya que el mecanismo de autenticación es algo obsoleto y se necesita de un alto nivel de seguridad para poder correr SNMP en una red privada sin riesgos.

2. **Complejidad y carga.**

   SNMP permitiría una gestión a niveles muy complejos. Mediante scripts podríamos comprobar a intervalos regulares el estado de los parámetros de la MIB sobre cada servidor, lo que implicaría un modelo de monitorización complejo y adaptable en el sentido de que podríamos requerir mayor cantidad de información que en cualesquiera de los otros métodos, pero implicaría mucha más carga en la red.

   Probablemente la solución está en añadir una nueva red para el apartado de monitorización.

3. **Mayor conocimiento y estudio del sistema.**

   Lo que supone encarecer el tiempo de creación de este y abaratar luego su mantenimiento.

Existen más puntos a tener en cuenta a la hora de elegir SNMP como método de monitorización. En cualquier caso, el uso de traps de SNMP nos permitiría de manera cómoda establecer mecanismos suficientes como para que el sistema estuviese funcionand o de manera correcta sin ningún problema. Hasta el momento a nadie se ha puesto a diseñar una MIB especial para LVS en los que se pueda manejar los nodos directores de manera remota o incluso la monitorización de los nodos servidores, este sistema en el caso de ser implantado, sería de suma

utilidad a la hora de realizar la gestión del cluster de una manera cómoda y sencilla desde otras estaciones y con programas como el HP-OPENVIEW o similares.

El método de utilizar scripts para monitorizar y actuar sobre el sistema es el punto opuesto a la opción SNMP. Tiene como desventajas, que es más lento de desarrollar y cada nueva acción de monitorización puede tener consecuencias colaterales en el sistema, por lo que en muchos casos es necesario realizar un estudio de cómo será la monitorización y comprobar que todos los componentes sean ortogonales entre sí. Como ventaja de este sistema es que es tan adaptable como lo queramos hacer, lo podemos programar en C, C++, Perl o shell scripts si nos apetece, y tan adecuado a nuestro sistema como lo necesitemos, esto es una ventaja y un inconveniente a su vez, ya que será difícil reutilizar el código de un sistema a otro, teniendo por tanto un ciclo de implementación en varios sistemas mucho más largo.

**MON**

El método que utilizaremos en este caso, es el que provee un paquete que incluye varios programas llamado MON. Este paquete está hecho prácticamente en perl, por lo que deberemos bajar no sólo el paquete MON sino todos los módulos de perl que se requieran para correr el programa.

La solución, MON, es una solución intermedia entre la mastodóntica SNMP y la casera SCRIPT. Es configurable, requiere poco tiempo de puesta a punto, y además también aporta tanta adaptabilidad a nuestras exigencias como nosotros queramos. Este paquete fue escrito por un administrador de sistemas de Transmeta llamado Jim Trocki, la compañía donde trabaja Linus Torvalds, y está prácticamente incluido en la totalidad de las distribuciones que conozco.

MON es un agente de tipo general para monitorizar tanto servicios como servidores y lanzar alertas que pueden ser fácilmente implementadas en C, perl o cualquier otro lenguaje. MON lee de un fichero de configuración su forma de actuación, dicho fichero tiene una forma similar en funcionamiento a SNMP, es una manera muy adaptable de monitorizar cualquier sistema.

En dicho fichero de configuración podemos encontrar las siguientes secciones:

- La primera es la sección *hostgroups*, que aporta los grupos a los que se realizara la monitorización.
- La segunda es la sección de vistas, que se compone de una o más vistas sobre los grupos de trabajo que contiene las especificaciones de la monitorización y la actuación que se hará acorde con cada evento. En las zonas de vistas se pueden separar en:
  - El servicio a monitorizar, pueden contenerse uno o más servicios en cada vista, de manera que se especifica en ellos.
  - El intervalo de resolución en el que se realizaran los sondeos de las monitoriza ciones.
  - Las traps, interrupciones o señales que se utilizarán.
  - El programa monitor que se utilizará para cada servicio.
  - La dependencia entre servicios y el comportamiento en caso de dependencias
  - El campo periodo que especifica entre qué fechas se realizará la monitorización y de qué manera se procederá en caso de ciertos eventos. Este campo posee a su vez otros campos como pueden ser
    - Periodo real
    - Programa de alerta o control de eventos
    - Varios parámetros para el funcionamiento y control de las alertas

Una vez explicada la estructura del fichero de configuración, pasaremos a explicar cuál es el método de funcionamiento de MON.

MON como ya hemos dicho se compone de un agente planificador que solamente se encarga de interpretar el fichero de configuración y realizar periódicamente y en las horas señaladas las llamadas oportunas a los programas de monitorizaci ón y alerta sobre los nodos perteneciente a un conjunto hostgroup. De esta manera, el programa monitor puede ser implementado por un programador, de manera que este es independiente de MON, mientras que acepte los parámetros que MON le envía y las variables de entorno que adquiere al ser hijo de este.

El programa monitor es llamado periódicamente para cada nodo perteneciente a un grupo de host o host particulares pertenecientes a dicha vista, hace su monitorización específica y da un código en la llamada *exit()*.

Este código debe ser interpretado por la sección de alertas, de manera que se proceda a la ejecución necesaria para el manejo del evento en el caso de que el código de salida del monitor sea el que decide de que manera se controla el evento.

De esta manera, lo único que debe crear el programador son los programas monitor y alerta, donde el programa monitor se encargará de monitorizar los eventos de los que depende nuestro sistema, ya sean estos adquisición en una tarjeta de adquisición de datos como control de un puerto ttyS<X> como el control de acceso a un servidor de un determinado servicio. Además programado mediante el lenguaje que resulte más cómodo al programador, lo que facilita la tarea.

Por otro lado el programa de alerta debe atender a dos tipos de parámetros y variables de entorno pasados por MON. El primer tipo son los códigos de retorno del programa monitor que deberá entender a la perfección para poder ejecutar el manejo del evento que detectase el monitor. El segundo son los parámetros que proporciona MON para el control de caída o puesta a punto de un servicio.

El paquete MON lleva incorporado unos cuantos programas monitores y alertas que se guardan dentro de */usr/lib/mon/*, estos programas por defecto suelen bastar para llevar a cabo la monitorización que nosotros necesitamos. Entre estos programas se encuentran los siguientes monitorizadores:

`ftp.monitor` monitor de FTP, comprueba el estado para poder acceder a un servidor FTPD mediante la cuenta que le proporcionemos como parámetro en el fichero de configuración.

`http.monitor` monitor HTTP, igual que el anterior pero para www.

`telnet.monitor` idem que los anteriores.

`smtp.monitor` idem para este otro protocolo.

`dialin.monitor` entradas de llamadas en el modem.

`snmp.monitor` compatible con SNMP, para comprobar el acceso a un agente.

Y muchos otros monitores que permiten monitorizar en general servicios de red mediante programas conocidos como ping, o peticiones a determinada pagina del servidor o a determinado recurso. De esta manera podemos ver el carácter adaptable que posee este tipo de configuración.

Además de los monitores se encuentran los scripts o programas de alerta que funcionan de la misma manera que los monitores, son llamados por MON con determinados parámetros de manera que se actúa acorde con el evento que generó la alerta, subsanando algún problema, guardando en los log que lo provocó, solucionando en la manera de lo posible dicho problema e informando mediante algún método al administrador del sistema, acerca del evento que produjo la llamada. Existen más programas externos que controlan el funcionamiento de MON como son, *monshow* y *mon-cgi*, estos permiten ver y controlar desde línea de órdenes o desde un navegador mediante el acceso a una WEB el estado y funcionamiento de MON. En nuestro caso no los hemos probado, ya que la solución que buscábamos no dependía ni requería de este CGI.

Una vez explicado el funcionamiento de MON, debemos explicar qué servicios debemos monitorizar, de qué manera los podemos monitorizar y que medidas debemos tomar ante determinados eventos. Nuestra intención es explicar este apartado utilizando como ejemplo el caso de estudio aportado arriba, de manera que se pueda comprobar como es el funcionamiento de MON y qué factores son importantes a la hora de configurar un sistema de monitorización de este tipo.

**Factores que intervienen en la configuración de MON**

Para empezar debemos hacer un estudio de ciertos factores que nos pueden interesar y que pueden afectar a nuestro sistema como son.

1. Servicios que poseemos en nuestro cluster.
2. Máquinas dentro del cluster. Es necesario para controlar la resolución de la monitorización, y que la carga de esta no sea demasiado alta.
3. Comportamiento ante caídas.

El apartado de servicios es necesario conocerlo a fondo. Es muy sencillo decir que se necesita un servicio de HTTP, otro de FTP y otro de squid, pero saber los temporizadores de cierre de conexión de dichos servicios no es tan fácil, en cambio seguramente estemos interesados en poder disponer de una monitorización sobre estos servícios.

Pongamos el ejemplo de un servicio de telnet sobre el que se activa algún tipo de envoltura como TCP wrapper, o kerberos. Este telnet tardará algo más en terminar la configuración de la sesión telnet y por tanto el tiempo de establecimiento sera algo más alto. De esta manera cuando un script que intente comprobar el estado de un servidor *telnetd* y éste tarde más de lo que esperaba (aunque sea tan solamente un segundo), se mandará la alerta apropiada quitando el servicio del LVS, aunque dicho servicio funcione y por tanto se dispondrá de una máquina menos dentro cluster con la que hacer el balanceo. También puede ser necesaria hacer una previsión de cuándo se deben realizar más monitorizaciones sobre un servicio, o las horas a las que no son necesarias hacerlas. Hay que recordar que MON es un programa que corre en el nodo director, y al ser este un punto crítico del sistema es conveniente mantenerlo lo más descargado posible para que se dedique a lo que realmente debe hacer, balancear conexiones y proporcionar servicios.

El apartado de las máquinas que contiene el cluster es necesario para establecer la tasa o intervalo de monitorización. Este intervalo afectará en dos aspectos. El primero es la carga del nodo director donde se ejecuta MON, al estar continuamente lanzando programas monitores y de alerta, el nodo consume recursos como son memoria y CPU, por lo que puede ser un factor importante a tener en cuenta en caso de haber instalado un nodo director inadecuado en requerimientos para dicha labor.

Por otro lado está la carga sobre la red sobre la que tanto estamos hablando a lo largo de todo el proyecto. Mantener la red lo menos congestionada posible es cuestión de cuidar este tipo de apartados y de tener un diseño de red previo adaptado a las necesidades que se preveen para nuestro sistema.

Por último hemos de definir cual será el comportamiento ante caídas. Generalmente en nuestro caso, el comportamiento suele ser algo como, quitar la entrada del nodo director DIRECCIÓN-SERVICIO-PESO, para que dicho nodo no sea tenido en cuenta en las siguientes decisiones del nodo director.

Avisar al administrador mediante cualquier método o recurso, enviando mensajes al móvil o mails a la cuenta del administrador o dando pitidos incesantes para que alguien trate de solucionar el problema lo antes posible. Intentar solucionar el problema desde el nodo director suele requerir de sistemas expertos que evalúen las probabilidades de la causa que provocó la caída e intente solucionarse. Registrar los eventos que sucedieron de manera que se sepa cuales fueron las causas de caída del servicio en cualquier momento.

Se pueden monitorizar en un LVS cualquier tipo de servicios o recursos, desde la accesibilidad a un nodo con fping.monitor hasta el espacio que queda libre en el sistema de almacenamiento distribuido NFS. De manera general, se suele monitorizar como mínimo el acceso a los servidores y el acceso a los servicios.

La utilización del acceso a los servicios es más conveniente que el acceso a caída de un servidor, pero implica más carga de red, por lo que depende, qué método será elegido monitor del sistema. Los programas monitores de un servicio, suelen ser programas que hacen una petición al servicio de carácter general (como pedir la información de versión o página principal de un servidor HTTP), esto conlleva mucha más carga en la red y en el sistema general que un fping destinado a un grupo de hosts, pero por el contrario, si imaginamos una situación muy usual como es la de tener una máquina con varios servicios activados no es difícil entender que puede caer un servicio, y seguir funcionando el otro, y por lo tanto fping dará una monitorización errónea respecto a lo que en un principio requeríamos para el sistema.

### Conclusiones

El servicio ofrecido por la conjunción de LVS, MON y Heartbeat pueden llegar a ser tan potente como otras aplicaciones o configuraciones propietarias que existen en el mercado a un precio mucho mayor. Uno de los problemas que no hemos comentado que tiene LVS de momento es que al caer el nodo director y retomar su trabajo el nodo de backup mediante heartbeat como veremos en el apartado de heartbeat, el contenido de la tabla hash, así como las conexiones y toda la información del nodo director se pierden, esto produce en los clientes que tenían una conexión en curso, la pérdida de dicha conexión.

En un principio y dependiendo de qué servicios puede ser más o menos drástico. La gente del proyecto LVS esta trabajando en mejorar este comportamiento, y se espera que dada la facilidad y adaptabilidad con la que pretende dotar Alan Robertson el proyecto heartbeat, cubrir este problema sea más simple [16].

[16] Si bien un nodo que ha caido, no puede enviar los últimos ack que recibió de una conexión, con lo cual, y pese a los intentos, el problema no es de resolución fácil, y ha sido aplazado por la gente de LVS desde hace un tiempo.

## 4.3. CLUSTERS HP

*The problems that exist in the world today*
*cannot be solved by the level of thinking that created them.*
*Albert Einstein*

### 4.3.1. Conceptos importantes: migración y balanceo)

Los clusters HP están dedicados a dar el mayor rendimiento computacional posible y existen multitud de formas de implementarlos.

Ha llevado años implementarlos, por tanto a lo largo de la historia ha habido todo tipo de ideas para intentar hacerlos lo más eficientes posible. En este documento se estudian unas cuantas de estas implementaciones (PVM, MPI, Beowulf) y se ahondará en una de ellas: openMosix.

La primera división en las implementaciones puede ser la división entre las

- soluciones que funcionan a nivel de aplicación y
- soluciones que funcionan a nivel de kernel.

Las que funcionan a nivel de aplicación suelen tomar forma de librería. Se tienen que realizar los programas para que aprovechen esta librería por lo tanto cualquier programa ya existente para que pueda ser usado en un cluster y mejore su rendimiento tiene que ser reescrito al menos parcialmente.

Esto tiene bastantes inconvenientes: muchas de las aplicaciones que necesitan grandes tiempos de cómputo se realizaron hace décadas en lenguaje Fortran. Este lenguaje aparte de estar relegado hoy en día a aplicaciones matemáticas o físicas, puede llegar a ser bastante difícil de comprender; por lo tanto es bastante difícil migrarlo al nuevo entorno distribuido.

Por otro lado una de las ventajas que tienen los clusters HP con respecto a los supercomputadores es que son bastante más económicos. Pero si el dinero que se ahorra en el hardware hay que invertirlo en cambiar los programas esta solución no aporta beneficios que justifiquen tal migración de equipos. Además hay que tener en cuenta que la mayor parte de las instituciones o instalaciones domésticas no tienen dinero para invertir en ese software, pero que sí disponen de ordenadores en una red (universidades por ejemplo) .

La segunda opción es que el software que se encarga del HP se encuentre en el kernel del sistema operativo. En este caso no se necesitan cambiar las aplicaciones de usuario, sino que éstas usan las llamadas estándar del kernel por lo tanto el kernel internamente es el que se encarga de distribuir el trabajo de forma transparente a dicha aplicación. Esto tiene la ventaja de que no hace falta hacer un desembolso en cambiar las aplicaciones que lo necesitan y que cualquier aplicación puede ser distribuida. Por supuesto un factor que siempre habrá que tener en cuenta es la propia programación de la aplicación.

Por otro lado esta aproximación también tiene varios inconvenientes: el kernel se vuelve mucho más complejo y es más propenso a fallos. También hay que tener en cuenta que estas soluciones son específicas de un kernel, por lo que si las aplicaciones no están pensadas para ese sistema operativo habría que portarlas. Si los sistemas operativos tienen las mismas llamadas al sistema, siguiendo un estándar POSIX, no habría grandes problemas. Otros sistemas operativos propietarios que no cumplen estos estándares no pueden disponer de estas ventajas.

Una forma de conseguir HP es migrando procesos, dividiendo las aplicaciones grandes en procesos y ejecutando cada proceso en un nodo distinto. Lo que se quiere conseguir es el máximo uso de los recursos en todo momento, especialmente los procesadores. Para conseguirlo hay dos aproximaciones:

- **Asignar estáticamente cada proceso a un nodo en particular.**

  En esta aproximación es importantísima la política de localización. Se elige estáticamente el nodo donde el proceso vivirá toda su vida. Por tanto es muy importante elegir correctamente estos nodos. Muchas veces esta solución necesita un administrador que decida dónde debe ir cada proceso. El caso más simple es tener todos los nodos con la misma potencia de cálculo y dividir el único programa al que se quiera dedicar los recursos en un número de procesos igual al número de nodos de los que se disponen. Asíse asignaría cada proceso a uno de los nodos. Hay que tener en cuenta que esta configuración es lejana a la normal y que ya que un fallo en el algoritmo de elección de nodo puede infrautilizar mucho de los recursos la configuración

| Tema | Problemas que genera |
|---|---|
| Sin requisa de procesos | Mayor latencia para procesos de alta prioridad |
| Con requisa de procesos | Sobrecarga, implementación compleja |
| Balanceo estático | Mal balanceo de carga |
| Balanceo dinámico | Sobrecarga, implementación compelja |
| Nivel de aplicación | Sobrecarga, falta de información |
| Nivel de kernel | Dificultad de implementación |
| Nodos dedicados | Infrautilizción de recursos |
| Nodos comparten espacio | Se encesita una plítica eficiente de localización |
| Nodos comparten tiempo | Sobrecarga cambio de contexto |
| Scheduling independiente | Pérdida de rendimiento |
| Scheduling de grupo | Implementación compleja |
| Con carga externa, quedarse | Pérdida de rendimiento en trabajos locales |
| Ante carga externa, mgrar | Sobrecarga de migración, límites de migración |

Cuadro 4.4: Clusters HP. Aspectos de implementación

manual es normal en estos algoritmos. Esto no es tan malo como pueda parecer a primera vista porque es también muy corriente en estos casos que una vez hecha la configuración inicial, los procesos estén ejecutándose durante años si es necesario.

- **Asignar dinámicamente procesos a los nodos.**

  Los procesos una vez iniciados en un nodo pueden migrar a otro nodo dinámicamente. En estos casos aunque es importante la política de localización para minimizar el gasto de recursos, también es importantísima la política de migración. Por supuesto también se pueden ubicar los procesos manualmente, con la ventaja de que se pueden ubicar en cualquier momento durante la vida del proceso. Si la política de migración es correcta y los procesos tienen una vida larga y se ha dividido correctamente la aplicación, debería haber al comienzo de la ejecución de los procesos un periodo de reordenación de los procesos, con varias migraciones, una vez el sistema llegara a una condición estable, no deberían producirse apenas migraciones hasta que los procesos finalizaran. Igual que ocurre en el caso anterior, esta configuración es lejana a la habitual, pero al contrario del caso anterior, aquíno es necesaria la configuración manual (si el algoritmo de migración es suficientemente bueno). Cuando se desbalancea el sistema éste se encarga de que se vuelva a balancear, de tal forma de que se aprovechen los recursos al máximo.

Sobre el balanceo ya se ha hablado en la sección anterior, como se puede comprender el correcto balanceo es muy complejo, se puede balancear con respecto a una variable (por ejemplo el procesador), pero hacerlo con respecto a todo el sistema en conjunto es algo demasiado complejo y el tiempo de cómputo para hacer las elecciones correctas sería muy grande. Por tanto estos algoritmos intentan tener un fundamento matemático fuerte que intenta minimizar el tiempo de cómputo. Muchos de estos sistemas usan fundamentos estadísticos, como por ejemplo openMosix.

Como ya se verá con más detalle, openMosix intenta maximizar el uso de todos los recursos. Intentar solamente balancear respecto al procesador puede dar lugar a decisiones bastante malas, porque se pueden enviar muchos procesos que no hagan mucho uso del procesador a uno de los nodos, si estos procesos están haciendo entrada/salida y son procesos grandes, es muy posible que el nodo empiece a hacer *trashing* pues se quedará sin memoria, con lo que los procesos no podrán ejecutar su función.

En el cuadro se aprecian las posibles formas que existen para clusters. Se muestra este cuadro para que se pueda comprender la cantidad de decisiones que se tienen que hacer cuando se implementa un cluster de este tipo y asíse comprenderá mejor por qué hay tantas implementaciones y tan dispares. Ya hemos explicado el balanceo estático frente a balanceo dinámico y el nivel de aplicación frente al nivel de sistema que serán conceptos que necesitemos en las próximas secciones, ahora vamos a explicar las demás características.

*Requisa* se refiere a poder parar el proceso y coger sus recursos (básicamente los registros del procesador y memoria). La requisa de los procesos puede existir o no. Con requisa no queremos decir hacer requisa de los procesos para migrarlos después, sino simplemente poder hacer requisa de un proceso en cualquier momento. Cuando un sistema es multitarea normalmente se implementa algún sistema de requisa para poder parar procesos

y hacer que otros procesos tomen el procesador para dar la sensación al usuario de que todos los procesos se están ejecutando concurrentemente.

Si no se implementa requisa, un proceso de baja prioridad puede tomar el procesador y otro proceso de alta prioridad no podrá tomarlo hasta que el proceso de baja prioridad lo ceda a otros procesos. Este esquema puede ser injusto con las prioridades y un error en un programa, puede llegar a dejar sin funcionar la máquina pues nunca devolvería el control, pero tampoco haría ningún trabajo útil. Además para sistemas que necesitan tiempo real, simplemente es inaceptable que procesos de baja prioridad estén dejando a los procesos de tiempo real sin tiempo de procesador y quizás con esta latencia extra se esté haciendo que el sistema no pueda cumplir sus operaciones en tiempo real, haciendo que el sistema sea inútil. Hoy en día la requisa se implementa al menos a un nivel elemental en casi todos los sistemas que necesiten hacer funcionar más de un proceso (por no decir todos). Algunos sistemas lo que hacen es no esperar a que cumpla un temporizador y realizar la requisa sino a esperar que el proceso haga alguna llamada al sistema para aprovechar, tomar el procesador y cederlo a otro proceso. Esta aproximación sigue teniendo el problema de que si un proceso maligno o mal programado no hace llamadas a sistema porque haya quedado en un bucle, nunca se ejecutará nada en ese ambiente.

Si se implementa la requisa hay que tener en cuenta que la implementación más simple que se puede dar (que es la que usan buena parte de los sistemas) necesita un temporizador que marque cuando se acabó el tiempo del proceso y requisar ese proceso para asignar a otro proceso. Esto impone una sobre carga pues hay que tratar una interrupción, actualizar unas variables para saber cuanto tiempo lleva un proceso trabajando.

Hay una implementación más compleja que trata de que siempre que haya un proceso de una prioridad mayor al que se está ejecutando se quite el procesador al proceso y se dé el procesador al proceso con mayor prioridad. Estos suelen ser sistemas en tiempo real que también (ya que se ponen) pueden tener otras exigencias como unos tiempos mínimos de latencia para ciertos procesos. Para conseguir esto, el kernel no solamente tiene que requisar procesos de baja prioridad en favor de los procesos de tiempo real sino que tiene que ser capaz de requisar su propio código. Esto suele significar que casi cualquier porción del código del kernel se puede ejecutar entre dos instrucciones de este mismo código. Esto presenta muchísimos problemas a la hora de programar, hay que tener mucho más cuidado con evitar condiciones de carrera dentro del propio kernel que antes por ser código no requisable no se podían dar. Por tanto implementar requisa, puede hacer que un sistema sea tiempo real pero complica tremendamente el núcleo del sistema.

Las siguientes tres líneas en el cuadro tratan sobre los recursos del cluster, estos son los nodos. Existen tres modos en los que se puede dedicar los nodos del cluster, estos modos son:

- **Modo dedicado.**

  En este modo que es el más simple de todos, solamente un trabajo está siendo ejecutado en el cluster en un tiempo dado, y como mucho un proceso de este trabajo que se está ejecutando es asignado a un nodo en cualquier momento en el que se siga ejecutando el trabajo. Este trabajo no liberará el cluster hasta que acabe completamente aunque solamente quede un proceso ejecutándose en un único nodo. Todos los recursos se dedican a este trabajo, como se puede comprender fácilmente esta forma de uso de un cluster puede llevar a una pérdida importante de potencia sobre todo si no todos los nodos acaban el trabajo al mismo tiempo.

- **Modo de división en el espacio.**

  En este modo, varios trabajos pueden estar ejecutándose en particiones disjuntas del cluster que no son más que grupos de nodos. Otra vez como mucho un proceso puede estar asignado a un cluster en un momento dado. Las particiones están dedicadas a un trabajo, la interconexión y el sistema de entrada/salida puede estar compartidos por todos los trabajos, consiguiendo un mejor aprovechamiento de los recursos. Los grupos de nodos son estáticos y los programas necesitan un número específico de nodos para poder ejecutarse, esto lleva a dos conclusiones:

  1. Puede existir trabajos para los que no haya una división lo suficientemente grande por lo que non podrían ser ejecutados
  2. Puede tener un trabajo que solamente aproveche alguno de los nodos desperdiciando una división de gran cantidad de nodos. Para evitar este segundo punto se tienen que usar técnicas para elegir inteligentemente los nodos donde se ejecuten los trabajos, intentando minimizar el número de nodos ociosos.

También puede ocurrir que un trabajo muy largo tome los recursos del cluster evitando que otros trabajos más rápidos acaben, esto consigue aumentar la latencia.

- **Modo de división en el tiempo.**

  En cada nodo pueden estar ejecutándose varios procesos a la vez por lo que se solucionan los problemas anteriores. Este es el modo más usado normalmente puesto que no tiene tantas restricciones como el otro y se puede intentar hacer un equilibrado de carga eligiendo correctamente los procesos.

Los dos siguientes puntos de la tabla tratan sobre *scheduling*, esta planifiación solo se da cuando el modo que se ha elegido es el modo de división en el tiempo.

Hay dos tipos de *scheduling* en cuanto a clusters se refiere:

- ***Scheduling* independiente.**

  Es el caso más sencillo y más implementado, se usa un sistema operativo en cada nodo del cluster para hacer *scheduling* de los distintos procesos en un nodo tradicional, esto también es llamado *scheduling* local. Sin embargo, el rendimiento de los trabajos paralelos que esté llevando a cabo el cluster puede verse degradado en gran medida.

  Cuando uno de los procesos del trabajo paralelo quiera hacer cualquier tipo de interacción con otro proceso por ejemplo sincronizarse con él, este proceso puede que no esté ejecutándose en esos momentos y puede que aún se tarde un tiempo (dependiente normalmente de su prioridad) hasta que se le ejecute por otro cuanto de tiempo. Esto quiere decir que el primer proceso tendrá que esperar y cuando el segundo proceso esté listo para interactuar quizás el primer proceso esté en swap y tenga que esperar a ser elegido otra vez para funcionar.

- ***Scheduling* de grupo.**

  En este tipo se hace *scheduling* sobre todos los procesos del trabajo a la vez. Cuando uno de los procesos está activo, todos los procesos están activos. Estudios[17] han demostrado que este tipo de *scheduling* puede aumentar el rendimiento en uno o dos puntos de magnitud. Los nodos del cluster no están perféctamente sincronizados. De hecho, la mayoría de los clusters son sistemas asíncronos, que no usan el mismo reloj.

  Cuando decimos, a todos los procesos se le hace *scheduling* a la vez, no quiere decir que sea exáctamene a la vez. Según ese mismo estudio, según aumenta la diferencia entre que se elige para ejecutarse el primer proceso y el último, se pierda rendimiento (se tarda más en acabar el trabajo). Para conseguir buenos rendimientos se tiene que o bien, permitir a los procesos funcionar por mucho tiempo de forma continuada o bien que la diferencia entre que un proceso se ejecuta y el ultimo se ejecuta es muy pequeña.

  Pero como se puede comprender, hacer *scheduling* en un cluster grande, siendo el *scheduling* una operación crítica y que tiene que estar optimizada al máximo es una operación bastante compleja de lograr, además se necesita la información de los nodos para poder tomar buenas decisiones, lo que acaba necesitando redes rápidas.

Las dos últimas filas tratan de que deben hacer los procesos cuando se encuentran que en su nodo local hay otros procesos que provienen de otros nodos. Estos pueden venir por alguna política de migración o porque se esté ejecutando el *scheduler* de grupo del que hemos hablado en el punto anterior. Los trabajos locales podrían tener prioridad sobre trabajos externos, por ejemplo los procesos de usuario interactivos donde no queremos que se degrade el rendimiento deben mantener mayor prioridad. Hay dos formas de tratar esta situación:

- El trabajo externo migra: si el proceso migra, existe el coste de migración pero el proceso puede ir a un nodo donde se ejecute de forma más eficiente.

- El trabajo externo se mantiene en el cluster: si el proceso se mantiene en el nodo donde se encuentra se evita la sobrecarga que conlleva la migración, para no afectar a los procesos de ese nodo se les da muy poca prioridad o por ejemplo se hace un grupo de procesos especial que son los extenso que disponen de menos CPU. El problema es que seguramente se ralenticen los procesos tanto locales como los externos, sobre todo si es un proceso que necesita frecuentar sincronizaciones, comunicación y acceso a Entrada/Salida.

Con una buenas decisiones en este apartado se puede solucionar los problemas expuestos.

[17] Estudios realizados en Berkeley

### 4.3.2. PVM y MPI

**Paso de mensajes**

Tanto PVM como MPI se basan en el concepto de paso de mensajes. Los mensajes son pasados entre los procesos para conseguir que se ejecuten de manera colaborativa y de forma sincronizada. Se ha elegido mensajes pues se puede implementar de forma más o menos efectiva en un cluster, los mensajes se pueden enviar en forma de paquete IP y el ordenador destino desempaqueta el mensaje y decide a que proceso va dirigido. Una vez hecho esto, envía la información al proceso en cuestión. MPI en particular se necesita conocer de forma básica los mecanismos de paso de mensajes. Hay tres mecanismos básicos de paso de mensajes:

- **Paso de mensajes síncrono.**

  Cuando un proceso P ejecuta un envío síncrono a un proceso Q, tiene que esperar hasta que el proceso Q ejecuta el correspondiente recibo de información síncrono. Ambos procesos no volverán del envío o el recibo hasta que el mensaje está a la vez enviado y recibido.

  Cuando el enviar y recibir acaban, el mensaje original puede ser inmediatamente sobrescrito por el nuevo mensaje de vuelta y éste puede ser inmediatamente leído por el proceso que envió originariamente el mensaje. No se necesita un buffer extra en el mismo buffer donde se encuentra el mensaje de ida, se escribe el mensaje de vuelta.

- **Enviar/Recibir bloqueante.**

  Un envío bloqueante es ejecutado cuando un proceso lo alcanza sin esperar el recibo correspondiente. Esta llamada bloquea hasta que el mensaje es efectivamente enviado, lo que significa que el mensaje (el buffer donde se encuentra el mensaje) puede ser reescrito sin problemas. Cuando la operación de enviar ha acabado no es necesario que se haya ejecutado una operación de recibir. Sólo sabemos que el mensaje fue enviado, puede haber sido recibido o puede estar en un buffer del nodo que lo envía, o en un buffer de algún lugar de la red de comunicaciones o puede que esté en el buffer del nodo receptor.

  Un recibo bloqueante es ejecutado cuando un proceso lo alcanza, sin esperar a su correspondiente envío. Sin embargo no puede acabar sin recibir un mensaje. Quizás el sistema esté proveyendo un buffer temporal para los mensajes.

- **Envío/recibo no bloqueante.**

  Un envío no bloqueante es ejecutado cuando un proceso lo alcanza, sin esperar al recibo. Puede acabar inmediatamente tras notificar al sistema que debe enviar el mensaje. Los datos del mensaje no están necesariamente fuera del buffer del mensaje, por lo que es posible incurrir en error si se sobreescriben los datos.

  Un recibo no bloqueante es ejecutado cuando un proceso lo alcanza, sin esperar el envío. Puede volver inmediatamente tras notificar al sistema que hay un mensaje que se debe recibir. El mensaje puede que no haya llegado aún, puede estar todavía en transito o puede no haber sido enviado aún.

**PVM**

PVM es un conjunto de herramientas y librerías que emulan un entorno de propósito general compuesto de nodos interconectados de distintas arquitecturas. El objetivo es conseguir que ese conjunto de nodos pueda ser usado de forma colaborativa para el procesamiento paralelo[18].

El modelo en el que se basa PVM es dividir las aplicaciones en distintas tareas (igual que ocurre con openMosix). Son los procesos los que se dividen por las máquinas para aprovechar todos los recursos. Cada tarea es responsable de una parte de la carga que conlleva esa aplicacion. PVM soporta tanto paralelismo en datos, como funcional o una mezcla de ambos.

PVM permite que las tareas se comuniquen y sincronicen con las demás tareas de la máquina virtual, enviando y recibiendo mensajes, muchas tareas de una aplicación pueden cooperar para resolver un problema en paralelo. Cada tarea puede enviar un mensaje a cualquiera de las otras tareas, sin límite de tamaño ni de número de mensajes.

[18] Descripción extraída directamente de la descripción oficial del proyecto.

El sistema PVM se compone de dos partes. La primera es un demonio, llamado ***pvmd*** que residen en todas los nodos que forman parte de la máquina virtual. Cuando un usuario quiere ejecutar una aplicación PVM, primero crea una máquina virtual para arrancar PVM. Entonces se puede ejecutar la aplicación PVM en cualquiera de los nodos. Muchos usuarios pueden configurar varias máquinas virtuales aunque se mezclen unas con las otras y se pueden ejecutar varias aplicaciones PVM simultáneamente. Cada demonio es responsable de todas las aplicaciones que se ejecutan en su nodo.

Asíel control está totalmente distribuido excepto por un demonio maestro, que es el primero que se ejecuto a mano por el usuario, los demás nodos fueron iniciados por el maestro y son esclavos. En todo momento siempre hay un *pvmd* maestro. Por tanto la máquina virtual mínima es de un miembro, el maestro.

La segunda parte del sistema es la **librería de PVM**. Contiene un repertorio de primitivas que son necesarias para la cooperación entre los procesos o threads de una aplicación. Esta librería contiene rutinas para inicialización y terminación de tareas, envío y recepción de mensajes, coordinar y sincronizar tareas, broadcast, modificar la máquina virtual.

Cuando un usuario define un conjunto de nodos, PVM abstrae toda la complejidad que tenga el sistema y toda esa complejidad se ve como un gran computador de memoria distribuida llamada máquina virtual. Esta máquina virtual es creada por el usuario cuando se comienza la operación. Es un conjunto de nodos elegidos por el usuario. En cualquier momento durante la operación puede elegir nuevos nodos para la máquina virtual. Esto puede ser de gran ayuda para mejorar la tolerancia a fallos pues se tiene unos cuantos nodos de reserva (PVM no tiene migración) para si alguno de los nodos fallara. O si se ve que un conjunto de nodos de una determinada red están fallando se pueden habilitar nodos de otra red para solucionarlo. Para conseguir abstraer toda la complejidad de las diferentes configuraciones, soporta la heterogeneidad de un sistema a tres niveles:

- Aplicaciones: las subtareas pueden estar hechas para aprovechar las arquitectura s sobre la que funcionan. Por tanto como se puede elegir en que conjunto de nodos se ejecutarán unas tareas específicas, podemos hacer nuestras aplicaciones con la arquitectura al máximo por lo que se puede optimizar y hacer que funcionen aplicaciones hechas para arquitecturas específicas con PVM.

- Máquinas: nodos con distintos formatos de datos están soportados, incluyendo arquitecturas secuenciales, vectoriales, SMP. Abstrae little endian y big endian.

- Redes: la máquina virtual puede ser interconectada gracias a distintas tecnologías de red. Para PVM, bajo él existe una red punto a punto, no fiable y no secuencial. Esto abstrae cualquier tecnología de red. Utiliza UDP y implementa toda la confiabilidad y todas las demás operaciones como broadcast en la propia librería PVM.

Tiene un conjunto de interfaces que está basado en la observación de las necesidades de la mayoría de las aplicaciones, que están escritas en C y Fortran. Los enlaces para C y C++ para la librería PVM están implementados como funciones, siguiendo las reglas usadas por la mayoría de los sistemas que usan C, incluyendo los sistemas operativos tipo UNIX. Los enlaces para Fortran están implementados como subrutinas más que funciones.

Todas las tareas están identificadas con un único identificador de tarea TID (Task IDentifier). Los mensajes son enviados y recibidos por TIDs. Son únicos en toda la máquina virtual y están determinados por el *pvmd* local y no se pueden elegir por el usuario. Varias funciones devuelven estos TIDs (*pvm_mytid(), pvm_parent(), etc.*)para permitir que las aplicaciones de los usuarios conozcan datos de las otras tareas. Existen grupos nombrados por los usuarios, que son agrupaciones lógicas de tareas. Cuando una tarea se une al grupo, a ésta se le asigna un único número dentro de ese grupo. Estos números empiezan en 0 y hasta el número de tareas que disponga el grupo. Cualquier tarea puede unirse o dejar cualquier grupo en cualquier momento sin tener que informar a ninguna otra tarea del grupo. Los grupos se pueden superponer y las tareas pueden enviar mensajes multicast a grupos de los que no son miembro.

Cuando una tarea se quiere comunicar con otra ocurren una serie de cosas, los datos que la tarea ha enviado con una operación *send*, son transferidos a su demonio local quien decodifica el nodo de destino y transfiere los datos al demonio destino. Este demonio decodifica la tarea destino y le entrega los datos. Este protocolo necesita 3 transferencias de datos de las cuales solamente una es sobre la red. También se puede elegir una política de encaminado directo (dependiente de los recursos disponibles). En esta política tras la primera comunicación entre dos tareas los datos sobre el camino a seguir por los datos son guardados en una caché local. Las siguientes llamadas son hechas directamente gracias a esta información. De esta manera las transferencias se reduce a una transferencia sobre la red. Para comunicar entre sé los demonios *pvmd* se usa UDP pues es mucho más sencillo,

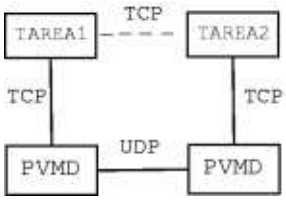



Figura 4.10: Clusters HP. Comunicaciones en PVM

sólo consume un descriptor de fichero, y con un simple socket UDP se puede comunicar a todos los demás demonios. Además es muy sencillo colocar temporizadores sobre UDP para detectar fallos de nodo, *pvmd* o red. La comunicación entre las tareas y los *pvmd* es mediante TCP puesto que se necesita tener la seguridad de que los datos llegarán. En el caso de que sólo se haga una trasferencia ésta es TCP por lo que hay que establecer la conexión primero por lo que realmente tampoco es tan beneficioso. En la siguiente figura se puede observar como los distintos métodos de comunicación de PVM.

Cada nodo tiene una estructura llamada `host table`. Esta tabla tiene una entrada (`host descriptor`) por cada nodo de la máquina virtual. El descriptor del nodo mantiene la información de la configuración del host, las colas de paquetes y los buffer de mensajes. Inicialmente la tabla sólo tiene la entrada del nodo maestro. Cuando un nuevo esclavo es incluido a la máquina virtual, la tabla del nodo maestro es actualizado para añadir al nuevo esclavo. Entonces esta nueva información es enviada por broadcast a todos los nodos que pertenezcan a la máquina virtual. De esta manera se actualizan todas las tablas y se mantienen consistentes.

Las aplicaciones pueden ver el hardware como una colección de elementos de proceso virtuales sin atributos o pueden intentar explotar las capacidades de máquinas específicas, intentando posicionar ciertas tareas en los nodos más apropiados para ejecutarlas.

Hemos querido dar un rápido repaso a PVM para poder decir qué es lo que no nos gusta de su aproximación y porque pensamos que openMosix es superior. Sabemos que la explicación que hemos dado está lejos de mostrar todo el universo de PVM pero pensamos que puede dar una idea de cómo funciona.

PVM no tiene requisa de procesos dinámico, esto quiere decir que una vez que un proceso empieza en una determinada máquina seguirá en ella hasta que se muera. Esto tiene graves inconvenientes como explicamos en las características de asignar estáticamente un proceso a un nodo en concreto. Hay que tener en cuenta que las cargas suelen variar y que, a no ser que todos los procesos que se estén ejecutando sean muy homogéneos entre sí, se está descompensando el cluster. Por lo tanto tenemos unos nodos más cargados que otros y seguramente unos nodos terminen su ejecución antes que otros, con lo que se podrían tener nodos muy cargados mientras otros nodos están libres. Esto lleva a una pérdida de rendimiento general.

Otro problema de PVM es que está implementado a nivel de usuario, esto no es malo de por sí pero teniendo en cuenta el tipo de operaciones que lleva, sílo es puesto que son operaciones de bastante bajo nivel como puedan ser paso de mensajes entre aplicaciones y la capa sobre UDP. Esto añade complejidad y latencia a las comunicaciones que se tienen que producir sobre las comunicaciones del kernel. Por lo que es una capa de software extra que carga bastante.

Se necesita un conocimiento amplio del sistema, tanto los programadores como los administradores tienen que conocer el sistema para sacar el máximo rendimiento de él. No existe un programa que se ejecute de forma ideal en cualquier arquitectura ni configuración de cluster. Por lo tanto para paralelizar correcta y eficazmente se necesita que los programadores y administradores conozcan a fondo el sistema.

El paralelismo es explícito, esto quiere decir que se programa de forma especial para poder usar las características especiales de PVM. Los programas deben ser reescritos. Si a esto se unimos que, como se necesita que los desarrolladores estén bien formados por lo explicado en el punto anterior y que conozcan además PVM, se puede decir que migrar una aplicación a un sistema PVM no es nada económico.

### MPI

MPI es una especificación estándar para una librería de funciones de paso de mensajes. MPI fue desarrollado por el *MPI Forum*, un consorcio de vendedores de ordenadores paralelos, escritores de librerías y especialistas en aplicaciones.

Consigue portabilidad proveyendo una librería de paso de mensajes estándar independiente de la plataforma y de dominio público. La especificación de esta librería está en una forma independiente del lenguaje y proporciona funciones para ser usadas con C y Fortran. Abstrae los sistemas operativos y el hardware. Hay implementaciones MPI en casi todas las máquinas y sistemas operativos. Esto significa que un programa paralelo escrito en C o Fortran usando MPI para el paso de mensajes, puede funcionar sin cambios en una gran variedad de hardware y sistemas operativos. Por estas razones MPI ha ganado gran aceptación dentro el mundillo de la computación paralela.

MPI tiene que ser implementado sobre un entorno que se preocupe de el manejo de los procesos y la E/S por ejemplo, puesto que MPI sólo se ocupa de la capa de comunicación por paso de mensajes. Necesita un ambiente de programación paralelo nativo.

Todos los procesos son creados cuando se carga el programa paralelo y están vivos hasta que el programa termina. Hay un grupo de procesos por defecto que consiste en todos esos procesos, identificado por MPI_COMM_WORLD.

Los procesos MPI son procesos como se han considerado tradicionalmente, del tipo pesados, cada proceso tiene su propio espacio de direcciones, por lo que otros procesos no pueden acceder directamente al las variables del espacio de direcciones de otro proceso. La intercomunicación de procesos se hace vía paso de mensajes.

Las desventajas de MPI son las mismas que se han citado en PVM, realmente son desventajas del modelo de paso de mensajes y de la implementación en espacio de usuario. Además aunque es un estándar y debería tener un API estándar, cada una de las implementaciones varía, no en las llamadas sino en el número de llamadas implementadas (MPI tiene unas 200 llamadas). Esto hace que en la práctica los diseñadores del sistema y los programadores tengan que conocer el sistema particular de MPI para sacar el máximo rendimiento. Además como sólo especifica el método de paso de mensajes, el resto del entorno puede ser totalmente distinto en cada implementación con lo que otra vez se impide esa portabilidad que teóricamente tiene.

Existen implementaciones fuera del estándar que son tolerantes a fallos, no son versiones demasiado populares porque causan mucha sobrecarga.

### 4.3.3. Beowulf

El proyecto Beowulf fue iniciado por Donald Becker (también famoso por crear numerosos drivers para tarjetas de red en Linux) en 1994 para la NASA. Este proyecto se basa en usar PVM y MPI, añadiendo algún programa más que se usan para monitorizar, realizar benchmarks y facilitar el manejo del cluster.

Entre las posibilidades que integra este proyecto se encuentra la posibilidad de que algunos equipos no necesiten discos duros, por eso se consideran que no son un cluster de estaciones de trabajo, sino que dicen que pueden introducir nodos heterogéneos. Esta posibilidad la da otro programa y Beowulf lo añade a su distribución.

Beowulf puede verse como un empaquetado de PVM/MPI junto con más software para facilitar el día a día del cluster pero no aporta realmente nada nuevo con respecto a tecnología.

### 4.3.4. openMosix

OpenMosix es un software para conseguir clustering en GNU/Linux, migrando los procesos de forma dinámica con requisa. Consiste en unos algoritmos de compartición de recursos adaptativos a nivel de kernel, que están enfocados a conseguir alto rendimiento, escalabilidad con baja sobrecarga y un cluster fácil de utilizar. La idea es que los procesos colaboren de forma que parezca que están en un mismo nodo.

Los algoritmos de openMosix son dinámicos lo que contrasta y es una fuerte ventaja frente a los algoritmos estáticos de PVM/MPI, responden a las variaciones en el uso de los recursos entre los nodos migrando procesos de un nodo a otro, con requisa y de forma transparente para el proceso, para balancear la carga y para evitar falta de memoria en un nodo.

Los fuentes de openMosix han sido desarrollados 7 veces para distintas versiones de Unix y BSD, nosotros en este proyecto siempre hablaremos de la séptima implementación que es la que se está llevando a cabo para Linux.

OpenMosix, al contrario que PVM/MPI, no necesita una adaptación de la aplicación ni siquiera que el usuario sepa nada sobre el cluster. Como se ha visto, para tomar ventaja con PVM/MPI hay que programar con sus librerías, por tanto hay que rehacer todo el código que haya (para aprovechar el cluster).

En la sección de PVM ya se han explicado las desventajas que tenía esta aproximación. Por otro lado openMosix puede balancear una única aplicación si esta está dividida en procesos lo que ocurre en gran número de aplicaciones hoy en día. Y también puede balancear las aplicaciones entre sí, lo que balancea openMosix son

procesos, es la mínima unidad de balanceo. Cuando un nodo está muy cargado por sus procesos y otro no, se migran procesos del primer nodo al segundo. Con lo que openMosix se puede usar con todo el software actual si bien la división en procesos ayuda al balanceo gran cantidad del software de gran carga ya dispone de esta división.

El usuario en PVM/MPI tiene que crear la máquina virtual decidiendo qué nodos del cluster usar para correr sus aplicaciones cada vez que las arranca y se debe conocer bastante bien la topología y características del cluster en general. Sin embargo en openMosix una vez que el administrador del sistema que es quien realmente conoce el sistema, lo ha instalado, cada usuario puede ejecutar sus aplicaciones y seguramente no descubra que se está balanceando la carga, simplemente verá que sus aplicaciones acabaron en un tiempo record.

PVM/MPI usa una adaptación inicial fija de los procesos a unos ciertos nodos, a veces considerando la carga pero ignorando la disponibilidad de otros recursos como puedan ser la memoria libre y la sobrecarga en dispositivos E/S.

En la práctica el problema de alojar recursos es mucho más complejo de lo que parece a primera vista y de como lo consideran estos proyectos, puesto que hay muchas clases de recursos (CPU, memoria, E/S, intercomunicación de procesos, etc.) donde cada tipo es usado de una forma distinta e impredecible. Si hay usuarios en el sistema, existe aún más complejidad y dificultad de prever que va a ocurrir, por lo que ya que alojar los procesos de forma estática es tan complejo que seguramente lleve a que se desperdicien recursos, lo mejor es una asignación dinámica de estos recursos.

Además estos paquetes funcionan a nivel de usuario, como si fueran aplicaciones corrientes, lo que les hacen incapaces de responder a las fluctuaciones de la carga o de otros recursos o de adaptar la carga entre los distintos nodos que participan en el cluster. En cambio openMosix funciona a nivel de kernel por tanto puede conseguir toda la información que necesite para decidir cómo está de cargado un sistema y qué pasos se deben seguir para aumentar el rendimiento, además puede realizar más funciones que cualquier aplicación a nivel de usuario, por ejemplo puede migrar procesos, lo que necesita una modificación de las estructuras del kernel.

### 4.3.5. TOP 500

Este *ranking* indica cuáles son los 500 computadores más potentes del mundo. Se incluyen MPPs, constelaciones, clusters y máquinas vectoriales. Vamos a destacar algunos de los resultados del Top de supercomputadores en diferentes épocas.

A fecha de junio de 2001 la lista demostraba claramente como estaban avanzando los supercomputadores, algunos datos curiosos fueron:

- El número 1 del top era `ASCI White` de IBM que llega a 7,2 TeraFlops/s.
- 12 de los sistemas tenían más de 1 TFlop/s, el más pequeño del*top ten* alcanzaba 1.18TFlop/s.
- El rendimiento total era de 108.8 TFlop/s, comparado con 88.8 TFlop/s del año anterior.
- El número 500 pasó de tener 55.1 TFlop/s a 67.8 TFlop/s.

Esta lista hace una división de clusters entre clusters tradicionales y constelaciones. De los cluster que cuyos nodos no son SMP podía verse que los dos primeros estaban en los puestos 30 y 31 y eran IBM.

## 4.4. REQUERIMIENTOS Y PLANTEAMIENTOS

*There are people who don't like capitalism, and people who don't like PCs.*
*But there's no-one who likes the PC who doesn't like Microsoft.*
*Bill Gates*

### 4.4.1. Requerimientos hardware

Para la instalación básica de un cluster necesitaremos al menos dos computadoras conectadas en red. Podremos conectarlas mediante un cable cruzado entre las respectivas tarjetas de red, con un *hub* o con un *switch*.

Evidentemente cuanto más rápida sea la conexión entre máquinas, más eficaz será nuestro sistema global.

Actualmente *Fast Ethernet* es un estándar, permitiendo múltiples puertos en una máquina. *Gigabit Ethernet* es más cara y no es recomendable probar con ella sin antes haberse asegurado un correcto funcionamiento con *Fast Ethernet* y comprobar que realmente se necesita este extra en la velocidad de transferencia de datos entre nodos.

Siempre podremos hacer funcionar varias tarjetas *Fast* en cada nodo para asignarles luego la misma dirección (IP) y de esta forma poder obtener múltiples en la velocidad.

El resto del hardware necesario dependerá de las decisiones que se hayan hecho con el sistema de ficheros (en red o no), la instalación de monitores gráficos en todos o solo algunos nodos, etc.

Algunas disposiciones hardware especiales pueden encontrarse en la sección *Nodos sin discos* del capítulo *Tutoriales para casos especiales*.

### 4.4.2. Lineas básicas en la configuración del hardware

Para poder configurar un *gran* cluster (refiriéndose al número de nodos) hay que pensar en ciertos aspectos, como por ejemplo dónde situar las máquinas. Tenerlas en medio de una oficina puede resultar incómodo en muchos aspectos. La mejor opción sería raquearlas.

El acondicionamiento de la sala donde deba situarse el cluster también es importante para evitar sobrecalentamientos y demás incomodidades a la hora de trabajar con él.

En todo caso hay que asegurarse de poder tener siempre un fácil acceso a los nodos.

### 4.4.3. Planteamientos del cluster

Para configurar tu cluster openMosix en un *pool* de nodos, o conjunto de estaciones de trabajo, tendremos diferentes opciones, cada una con sus ventajas e inconvenientes.

En una **single-pool** todos los servidores y estaciones de trabajo son utilizadas como un cluster único: cada máquina forma parte del cluster y puede migrar procesos hacia cada uno de los otros nodos existentes.

Esta configuración hace que tu propia máquina forme parte del *pool*.

En un entorno llamado **server-pool** los servidores son parte del cluster mientras que las estaciones de trabajo no lo son. Si quisiéramos ejecutar aplicaciones en el cluster necesitaremos entrar en él de forma específica. De este modo las estaciones de trabajo permanecerán libres de procesos remotos que les pudieran llegar.

Existe una tercera alternativa llamada **adaptive-pool**, donde los servidores son compartidos mientras que las estaciones de trabajo podrán entrar y salir del cluster. Podemos imaginar que las estaciones deban ser usadas durante un cierto intervalo de tiempo diario, y que fuera de este horario puedan ser aprovechadas para las tareas del cluster.

# Capítulo 5

# Clustering con openMosix

## 5.1. ¿QUÉ ES REALMENTE OPENMOSIX?

*There are two ways of constructing a software design;*
*one way is to make it so simple that there are obviously no deficiencies,*
*and the other way is to make it so complicated that there are no obvious deficiencies.*
*The first method is far more difficult.*

*C. A. R. Hoare*

### 5.1.1. Una muy breve introducción al clustering

La mayor parte del tiempo tu computadora permanece ociosa. Si lanzas un programa de monitorización del sistema como `xload` o `top`, verás probablemente que la lectura de la carga de tu procesador permanece generalmente por debajo del 10 %.

Si tienes al alcance varias computadoras los resultados serán los mismos ya que no podrás interactuar con más de una de ellas al mismo tiempo. Desafortunadamente cuando realmente necesites potencia computacional (como por ejemplo para comprimir un fichero *Ogg Vorbis*, o para una gran compilación) no podrás disponer de la potencia conjunta que te proporcionarían todas ellas como un todo.

La idea que se esconde en el trasfondo del *clustering* es precisamente poder contar con todos los recursos que puedan brindarte el conjunto de computadoras de que puedas disponer para poder aprovechar aquellos que permanecen sin usar, basicamente en otras computadoras.

La unidad básica de un cluster es una computadora simple, también denominada **nodo**. Los clusters pueden crecer en tamaño (o mejor dicho, pueden *escalar*) añadiendo más máquinas.

Un cluster como un todo puede ser más potente que la más veloz de las máquinas con las que cuenta, factor que estará ligado irremediablemente a la velocidad de conexión con la que hemos construido las comunicaciones entre nodos.

Además, el sistema operativo del cluster puede hacer un mejor uso del hardware disponible en respuesta al cambio de condiciones. Esto produce un reto a un cluster heterogéneo (compuesto por máquinas de diferente arquitectura) tal como iremos viendo paso a paso.

#### HPC, Fail-over y Load-balancing

Básicamente existen tres tipos de clusters: *Fail-over*, *Load-balancing* y *HIGH Performance Computing*.

Los clusters **Fail-over** consisten en dos o más computadoras conectadas en red con una conexión *heartbeat* separada entre ellas. La conexión heartbeat entre las computadoras es usualmente utilizada para monitorear cuál de todos los servicios está en uso, así como la sustitución de una máquina por otra cuando uno de sus servicios haya caído.

El concepto en los **Load-balancing** se basa en que cuando haya una petición entrante al servidor web, el cluster verifica cuál de las máquinas disponibles posee mayores recursos libres, para luego asignarle el trabajo pertinente.

Actualmente un cluster *load-balancing* es también *fail-over* con el extra del balanceo de la carga y a menudo con mayor número de nodos.

La última variación en el clustering son los **High Performance Computing**.

Estas máquinas han estado configuradas especialmente para centros de datos que requieren una potencia de computación extrema.

Los clusters **Beowulf** han sido diseñados específicamente para estas tareas de tipo masivo, teniendo en contrapartida otras limitaciones que no lo hacen tan accesible para el usuario como un openMosix.

#### Mainframes y supercomputadoras vs. clusters

Tradicionalmente los *mainframes* y las supercomputadoras han estado construidas solamente por unos fabricantes muy concretos y para un colectivo elitista que necesitaba gran potencia de cálculo, como pueden ser empresas o universidades.

Pero muchos colectivos no pueden afrontar el costo económico que supone adquirir una máquina de estas características, y aquí es donde toma la máxima importancia la idea de poder disponer de esa potencia de cálculo, pero a un precio muy inferior.

El concepto de cluster nació cuando los pioneros de la supercomputación intentaban difundir diferentes procesos entre varias computadoras, para luego poder recoger los resultados que dichos procesos debían producir. Con un hardware más barato y fácil de conseguir se pudo perfilar que podrían conseguirse resultados muy parecidos a los obtenidos con aquellas máquinas mucho más costosas, como se ha venido probando desde entonces.

#### Modelos de clusters NUMA, PVM y MPI

Hay diferentes formas de hacer procesamiento paralelo, entre las más conocidas y usadas podemos destacar NUMA, PVM y MPI.

Las máquinas de tipo NUMA (Non-Uniform Memory Access) tienen acceso compartido a la memoria donde pueden ejecutar su código de programa. En el kernel de Linux hay ya implementado NUMA, que hace variar el número de accesos a las diferentes regiones de memoria.

PVM / MPI son herramientas que han estado ampliamente utilizadas y son muy conocidas por la gente que entiende de supercomputación basada en GNU/Linux.

MPI es el estándar abierto de bibliotecas de paso de mensajes. MPICH es una de las implementaciones más usadas de MPI, tras MPICH se puede encontrar LAM, otra implementación basada en MPI también con bibliotecas de código abierto.

PVM (Parallel Virtual Machine) es un primo de MPI que también es ampliamente usado para funcionar en entornos Beowulf.

PVM habita en el espacio de usuario y tiene la ventaja que no hacen falta modificaciones en el kernel de Linux, basicamente cada usuario con derechos suficientes puede ejecutar PVM.

### 5.1.2. Una aproximación histoórica

#### Desarrollo histórico

Algunos rumores hablaban que MOSIX venía de Moshe Unix. Inicialmente Mosix empezó siendo una aplicación para BSD/OS 3.0, como podemos leer en este email:

```
Announcing MO6 for BSD/OS 3.0
Oren Laadan (orenl@cs.huji.ac.il)
Tue, 9 Sep 1997 19:50:12 +0300 (IDT)

Hi:

We are pleased to announce the availability of MO6 Version 3.0
Release 1.04 (beta-4) - compatible with BSD/OS 3.0, patch level
K300-001 through M300-029.

MO6 is a 6 processor version of the MOSIX multicomputer enhancements
of BSD/OS for a PC Cluster. If you have 2 to 6 PC's connected by a
LAN, you can experience truly multi-computing environment by using
the MO6 enhancements.

The MO6 Distribution
--------------------
MO6 is available either in "source" or "binary" distribution. It is
installed as a patch to BSD/OS, using an interactive installation
script.

MO6 is available at http://www.cnds.jhu.edu/mirrors/mosix/
or at our site: http://www.cs.huji.ac.il/mosix/
```

```
Main highlights of the current release:
---------------------------------------
- Memory ushering (depletion prevention) by process migration.
- Improved installation procedure.
- Enhanced migration control.
- Improved administration tools.
- More user utilities.
- More documentation and new man pages.
- Dynamic configurations.

Please send feedback and comments to mosix@cs.huji.ac.il.
-------------------
```

La plataforma GNU/Linux para el desarrollo de posteriores versiones fue elegida en la séptima reencarnación, en 1999.

A principios de 1999 Mosix M06 fue lanzado para el kernel de Linux 2.2.1.

Entre finales de 2001 e inicios de 2002 nacía openMosix, la versión de código abierto, de forma separada.

**openMosix != MOSIX**

openMosix en principio tenía que ser una ampliación a lo que años atrás ya se podía encontrar en `www.mosix.org`, respetando todo el trabajo llevado a cabo por el Prof. Barak y su equipo.

Moshe Bar estuvo ligado al proyecto Mosix, en la Universidad Hebrea de Jerusalem, durante bastantes años. Era el co-administrador del proyecto y el principal administrador de los asuntos comerciales de *Mosix company*.

Tras algunas diferencias de opinión sobre el futuro comercial de Mosix, Moshe Bar empezó un nuevo proyecto de *clustering* alzando la empresa *Qlusters, Inc.* en la que el profesor A. Barak[1] decidió no participar ya que no quería poner Mosix bajo licencia GPL.

Como había una significativa base de usuarios clientes de la tecnologia Mosix (unas 1000 instalaciones a lo ancho del planeta) Moshe Bar decidió continuar el desarrollo de Mosix pero bajo otro nombre, openMosix, totalmente bajo licencia GPL2.

openMosix es un parche (*patch*) para el kernel de linux que proporciona compatibilidad completa con el estardard de Linux para plataformas IA32. Actualmente se está trabajando para portarlo a IA64.

El algoritmo interno de balanceo de carga migra, transparentemente para el usuario, los procesos entre los nodos del cluster. La principal ventaja es una mejor compartición de recursos entre nodos, así como un mejor aprovechamiento de los mismos.

El cluster escoge por sí mismo la utilización óptima de los recursos que son necesarios en cada momento, y de forma automática.

Esta característica de migración transparente hace que el cluster funcione a todos los efectos como un gran sistema SMP (*Symmetric Multi Processing*) con varios procesadores disponibles. Su estabilidad ha sido ampliamente probada aunque todavía se está trabajando en diversas líneas para aumentar su eficiencia.

openMosix está respaldado y siendo desarrollado por personas muy competentes y respetadas en el mundo del *open source*, trabajando juntas en todo el mundo.

El punto fuerte de este proyecto es que intenta crear un estándar en el entorno del clustering para todo tipo de aplicaciones HPC.

openMosix tiene una página web[2] con un árbol CVS[3] y un par de listas de correo para los desarrolladores y para los usuarios.

**openMosix en acción: un ejemplo**

Los clusters openMosix pueden adoptar varias formas. Para demostrarlo intentad imaginar que compartes el piso de estudiante con un chico adinerado que estudia ciencias de la computación. Imaginad también que tenéis las computadoras conectadas en red para formar un cluster openMosix.

[1] `http://www.cs.huji.ac.il/~amnon/`
[2] `http://www.openmosix.org`
[3] `http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/openmosix/`

Asume también que te encuentras convirtiendo ficheros de música desde tus CDs de audio a Ogg Vorbis para tu uso privado, cosa que resulta ser legal en tu país.

Tu compañero de habitación se encuentra trabajando en un proyecto de C++ que según dice podrá traer la paz mundial, pero en este justo momento está en el servicio cantando cosas ininteligibles, y evidentemente su computadora está a la espera de ser intervenida de nuevo.

Resulta que cuando inicias un programa de compresión, como puede ser `bladeenc` para convertir un preludio de Bach desde el fichero *.wav* al *.ogg*, las rutinas de openMosix en tu máquina comparan la carga de procesos en tu máquina y en la de tu compañero y deciden que es mejor migrar las tareas de compresión ya que el otro nodo es más potente debido a que el chico disponía de más medios económicos para poderse permitir una computadora más potente, a la vez que en ese momento permanece ociosa ya que no se encuentra frente a ella.

Así pues lo que normalemte en tu pentium233 tardaría varios minutos te das cuenta que ha terminado en pocos segundos.

Lo que ha ocurrido es que gran parte de la tarea ha sido ejecutada en el AMD AthlonXP de tu compañero, de forma transparente a ti.

Minutos después te encuentras escribiendo y tu compañero de habitación vuelve del servicio. Éste reanuda sus pruebas de compilación utilizando `pmake`, una versión del `make` optimizada para arquitecturas paralelas. Te das cuenta que openMosix está migrando hacia tu máquina algunos subprocesos con el fin de equilibrar la carga.

Esta configuración se llama *single-pool*: todas las computadoras están dispuestas como un único cluster. La ventaja o desventaja de esta disposición es que tu computadora es parte del *pool*: tus procesos serán ejecutados, al menos en parte, en otras computadoras, pudiendo atentar contra tu privacidad de datos. Evidentemente las tareas de los demás también podrán ser ejecutadas en la tuya.

## 5.2. CARACTERISTICAS DE OPENMOSIX

*We all know Linux is great...*
*it does infinite loops in 5 seconds.*

*Linus Torvalds*

### 5.2.1. Pros de openMosix

- No se requieren paquetes extra
- No son necesarias modificaciones en el código

### 5.2.2. Contras de openMosix

- Es dependiente del kernel
- No migra todos los procesos siempre, tiene limitaciones de funcionamiento
- Problemas con memoria compartida

Además los procesos con múltiples *threads* no ganan demasiada eficiencia
Tampoco se obtendrá mucha mejora cuando se ejecute un solo proceso, como por ejemplo el navegador.

### 5.2.3. Subsistemas de openMosix

Actualmente podemos dividir los parches de openMosix dentro del kernel en cuatro grandes subsistemas, veámoslos.

#### Mosix File System (MFS)

El primer y mayor subsistema (en cuanto a líneas de código) es MFS que te permite un acceso a sistemas de ficheros (FS) remotos (i.e. de cualquier otro nodo) si está localmente montado.

El sistema de ficheros de tu nodo y de los demás podrán ser montados en el directorio */mfs* y de esta forma se podrá, por ejemplo, acceder al directorio */home* del nodo 3 dentro del directorio */mfs/3/home* desde cualquier nodo del cluster.

#### Migración de procesos

Con openMosix se puede lanzar un proceso en una computadora y ver si se ejecuta en otra, en el seno del cluster.

Cada proceso tiene su único nodo raíz (UHN, *unique home node*) que se corresponde con el que lo ha generado.

El concepto de migración significa que un proceso se divide en dos partes: la parte del usuario y la del sistema.

La parte, o área, de usuario será movida al nodo remoto mientras el área de sistema espera en el raíz.

openMosix se encargará de establecer la comunicación entre estos 2 procesos.

#### Direct File System Access (DFSA)

openMosix proporciona MFS con la opción DFSA que permite acceso a todos los sistemas de ficheros, tanto locales como remotos. Para más información diríjase a la sección de *Sistema de ficheros* de las FAQs (preguntas más frecuentes) del presente documento.

#### Memory ushering

Este subsistema se encarga de migrar las tareas que superan la memoria disponible en el nodo en el que se ejecutan. Las tareas que superan dicho límite se migran forzosamente a un nodo destino de entre los nodos del cluster que tengan suficiente memoria como para ejecutar el proceso sin necesidad de hacer *swap* a disco, ahorrando así la gran pérdida de rendimiento que esto supone. El subsistema de *memory ushering* es un subsistema independiente del subsistema de equilibrado de carga, y por ello se le considera por separado.

### 5.2.4. El algoritmo de migración

De entre las propiedades compartidas entre Mosix y openMosix podemos destacar el mecanismo de migración, en el que puede migrar-se cualquiera proceso a cualquier nodo del cluster de forma completamente transparente al proceso migrado. La migración también puede ser automática: el algoritmo que lo implementa tiene una complejidad computacional del orden de O(n), siendo *n* el número de nodos del cluster.

Para implementarlo openMosix utiliza el modelo *fork-and-forget*[4], desarrollado en un principio dentro de Mosix para máquinas PDP11/45 empleadas en las fuerzas aéreas norteamericanas. La idea de este modelo es que la distribución de tareas en el cluster la determina openMosix de forma dinámica, conforme se van creando tareas. Cuando un nodo está demasiado cargado, y las tareas que se están ejecutando puedan migrar a cualquier otro nodo del cluster. Así desde que se ejecuta una tarea hasta que ésta muere, podrá migrar de un nodo a otro, sin que el proceso sufra mayores cambios.

Podríamos pensar que el comportamiento de un clsuter openMosix es como una máquina NUMA, aunque estos clusters son mucho más baratos.

#### El nodo raíz

Cada proceso ejecutado en el cluster tiene un único nodo raíz, como se ha visto. El nodo raíz es el nodo en el cual se lanza originalmente el proceso y donde éste empieza a ejecutarse.

Desde el punto de vista del espacio de procesos de las máquinas del cluster, cada proceso (con su correspondiente PID) parece ejecutarse en su nodo raíz. El nodo de ejecución puede ser el nodo raíz u otro diferente, hecho que da lugar a que el proceso no use un PID del nodo de ejecución, sino que el proceso migrado se ejecutará en éste como una hebra del kernel. La interacción con un proceso, por ejemplo enviarle señales desde cualquier otro proceso migrado, se puede realizar exclusivamente desde el nodo raíz.

El usuario que ejecuta un proceso en el cluster ha accedido al cluster desde el nodo raíz del proceso (puesto que ha logado en él). El propietario del proceso en cuestión tendrá control en todo momento del mismo como si se ejecutara localmente.

Por otra parte la migración y el retorno al nodo raíz de un proceso se puede realizar tanto desde el nodo raíz como desde el nodo dónde se ejecuta el proceso. Esta tarea la puede llevar a término el administrador de cualquiera de los dos sistemas.

#### El mecanismo de migrado

**La migración de procesos en openMosix es completamente transparente**. Esto significa que al proceso migrado no se le avisa de que ya no se ejecuta en su nodo de origen. Es más, este proceso migrado seguirá ejecutándose como si siguiera en el nodo origen: si escribiera o leyera al disco, lo haría en el nodo origen, hecho que supone leer o grabar remotamente en este nodo.

#### ¿Cuándo podrá migrar un proceso?

Desgraciadamente, no todos los procesos pueden migrar en cualquiera circunstancia. El mecanismo de migración de procesos puede operar sobre cualquier tarea de un nodo sobre el que se cumplen algunas condiciones predeterminadas. Éstas son:

- el proceso no puede ejecutarse en modo de emulación VM86

[4] hace fererencia a que el sistema cuando reconoce un subproceso se encarga de ejecutarlo en otro nodo, en paralelo, sin ningun efecto ni notificación al propietario del mismo

- el proceso no puede ejecutar instrucciones en ensamblador propias de la máquina donde se lanza y que no tiene la máquina destino (en un cluster heterogéneo)
- el proceso no puede mapear memoria de un dispositivo a la RAM, ni acceder directamente a los registros de un dispositivo
- el proceso no puede usar segmentos de memoria compartida

Cumpliendo todas estas condiciones el proceso puede migrar y ejecutarse migrado. No obstante, como podemos sospechar, openMosix no adivina nada. openMosix no sabe a priori si alguno de los procesos que pueden migrar tendrán algunos de estos problemas.

Por esto en un principio openMosix migra todos los procesos que puedan hacerlo si por el momento cumplen todas las condiciones, y en caso de que algún proceso deje de cumplirlas, lo devuelve de nuevo a su nodo raíz para que se ejecute en él mientras no pueda migrar de nuevo.

Todo esto significa que mientras el proceso esté en modo de emulación VM86, mapee memoria de un dispositivo RAM, acceda a un registro o tenga reservado/bloqueado un puntero a un segmento de memoria compartida, el proceso se ejecutará en el nodo raíz, y cuando acabe la condición que lo bloquea volverá a migrar.

Con el uso de instrucciones asociadas a procesadores no compatibles entre ellos, openMosix tiene un comportamiento diferente: solo permitirá migrar a los procesadores que tengan la misma arquitectura.

**La comunicación entre las dos áreas**

Un aspecto importante en el que podemos tener interés es en cómo se realiza la comunicación entre el área de usuario y el área de kernel.

En algún momento, el proceso migrado puede necesitar hacer alguna llamada al sistema. Esta llamada se captura y se evalúa

- si puede ser ejecutada al nodo al que la tarea ha migrado, o
- si necesita ser lanzada en el nodo raíz del proceso migrado

Si la llamada puede ser lanzada al nodo dónde la tarea migrada se ejecuta, los accesos al kernel se hacen de forma local, es decir, que se atiende en el nodo dónde la tarea se ejecuta sin ninguna carga adicional a la red.

Por desgracia, las llamadas más comunes son las que se han de ejecutar forzosamente al nodo raíz, puesto que *hablan* con el hardware. Es el caso, por ejemplo, de una lectura o una escritura a disco. En este caso el subsistema de openMosix del nodo dónde se ejecuta la tarea contacta con el subsistema de openMosix del nodo raíz. Para enviarle la petición, así como todos los parámetros y los datos del nodo raíz que necesitará procesar.

El nodo raíz procesará la llamada y enviará de vuelta al nodo dónde se está ejecutando realmente el proceso migrado:

- el valor del éxito/fracaso de la llamada
- aquello que necesite saber para actualizar sus segmentos de datos, de pila y de *heap*
- el estado en el que estaría si se estuviera ejecutando el proceso al nodo raíz

Esta comunicación también puede ser generada por el nodo raíz. Es el caso, por ejemplo, del envío de una señal. El subsistema de openMosix del nodo raíz contacta con el subsistema de openMosix del nodo dónde el proceso migrado se ejecuta, y el avisa que ha ocurrido un evento asíncono. El subsistema de openMosix del nodo dónde el proceso migrado se ejecuta parará el proceso migrado y el nodo raíz podrá empezar a atender el código del área del kernel que correspondería a la seal asíncrona.

Finalmente, una vez realizada toda el operativa necesaria de la área del kernel, el subsistema de openMosix del nodo raíz del proceso envía al nodo donde está ejecutándose realmente el proceso migrado el aviso detallado de la llamada, y todo aquello que el proceso necesita saber (anteriormente enumerado) cuando recibió la señal, y el proceso migrado finalmente recuperará el control.

Por todo esto el proceso migrado es como sí estuviera al nodo raíz y hubiera recibido la señal de éste. Tenemos un escenario muy simple donde el proceso se suspende esperando un recurso. Recordemos que la suspensión esperando un recurso se produce únicamente en área de kernel. Cuando se pide una página de disco o se espera un paquete de red se resuelto como en el primero caso comentado, es decir, como un llamada al kernel.

Este mecanismo de comunicación entre áreas es el que nos asegura que

- la migración sea completamente transparente tanto para el proceso que migra como para los procesos que cohabiten con el nodo raíz
- que el proceso no necesite ser reescrito para poder migrar, ni sea necesario conocer la topologia del cluster para escribir una aplicación paralela

No obstante, en el caso de llamadas al kernel que tengan que ser enviadas forzosamente al nodo raíz, tendremos una sobrecarga adicional a la red debida a la transmisión constante de las llamadas al kernel y la recepción de sus valores de vuelta.

Destacamos especialmente esta sobrecarga en el acceso a sockets y el acceso a disco duro, que son las dos operaciones más importantes que se habrán de ejecutar en el nodo raíz y suponen una sobrecarga al proceso de comunicación entre la área de usuario migrada y la área de kernel del proceso migrado.

## 5.3. INSTALACIÓN DE UN CLUSTER OPENMOSIX

*A good traveler has no fixed plans,*
*and is not intent on arriving.*

*Lao Tzu*

Puede construirse físicamente un cluster openMosix de la misma forma que se construye un cluster Beowulf. Por otro lado, desde el punto de vista del programario, la construcción de un cluster openMosix es distinta.

La construcción de un cluster openMosix se compone de varios pasos.

1. el primero es compilar e instalar un kernel con soporte openMosix;
2. el segundo, compilar e instalar las herramientas de área de usuario.
3. El tercer paso es configurar los demonios del sistema para que cada máquina del cluster siga el comportamiento que esperamos,
4. y el cuarto paso es crear el fichero del mapa del cluster. Este cuarto paso sólo es necesario si no usamos la herramienta de autodetección de nodos.

### 5.3.1. Instalación del kernel de openMosix

El primer paso para instalar el kernel de openMosix es descargar el tarball. No es conveniente usar la versión del kernel del CVS, ya que suele no ser muy estable. Puede descargarse el tarball del parche del kernel de openMosix de la dirección en SourceForge[5].

En las fechas en las que se escribe este capítulo, la versión del kernel openMosix para el kernel de Linux 2.4.20 va por su versión 2.

Una vez descargado el parche del kernel openMosix habrá que descomprimirlo:

```
gunzip openMosix-2.4.20-2.gz
```

Después de descargar el parche openMosix, habrá que descargar el kernel de Linux. Un posible sitio para descargarlo es http://www.kernel.org .

Hay que descargar el kernel correspondiente a la versión del parche que hemos descargado. Esto quiere decir que un parche 2.4.X-Y valdrá para el kernel 2.4.X. Por ejemplo, el parche 2.4.20-2 sólo puede ser aplicado sobre el kernel 2.4.20.

Una vez descargado el kernel de Linux lo descomprimimos:

```
tar -zxf linux-2.4.20.tar.gz
```

Movemos el directorio donde hemos descomprimido el kernel a *linux-openmosix*:

```
mv linux-2.4.20 linux-openmosix
```

y aplicamos el parche de openMosix:

```
patch -p0 < openMosix-2.4.20-2
```

Entramos en el directorio del kernel de Linux:

```
cd linux-openmosix
```

Y lanzamos el menú de configuración del kernel:

```
make menuconfig
```

para la configuración a través de menús con *ncurses*, o:

[5] http://sourceforge.net/project/showfiles.php?group_id=46729

```
make xconfig
```

para la configuración a través de X-Window.

**Opciones del kernel de openMosix**

El siguiente paso para configurar el kernel de openMosix es entrar en la opción `openMosix` -la primera opción del menú principal de la pantalla de configuración del kernel-. Allí encontraremos un menú con todas las opciones propias de openMosix. Estas opciones son:

**openMosix process migration support**: Esta opción permite activar el soporte a la migración de procesos en openMosix. Esto incluye tanto la migración forzada por el administrador como la migración transparente automática de procesos, el algoritmo de equilibrado de carga y el *Memory Ushering*. Si no activamos esta opción, no tenemos openMosix. Por ello, si estamos leyendo este documento, nos interesa tener esta opción activada.

**Support clusters with a complex network topology**: Las máquinas que pertenecen al cluster openMosix pueden pertenecer a la misma red IP, estando conectadas por un hub o por un switch. En este caso, en openMosix consideramos que la topología de la red es simple, lo que permite realizar algunas modificaciones en los algoritmos de calculo de la función de coste del algoritmo de equilibrado de carga que hacen muchísimo más eficiente su cálculo. También mejora la eficiencia del algoritmo de colecta automática de información del cluster. Si tenemos todas las máquinas del cluster conectadas a través de hubs o switchs -es decir, que un paquete openMosix nunca necesitará pasar por un router- podemos aumentar sensiblemente el rendimiento de nuestro cluster desactivando esta opción.

**Maximum network-topology complexity to support (2-10)**: Si activamos la opción anterior, aparecerá esta opción. En esta opción se nos pregunta cuantos niveles de complejidad hay entre las dos máquinas más lejanas del cluster, entendiendo por niveles de complejidad el número de routers intermedios más uno. Si ponemos un número más alto de la cuenta, no tendremos todo el rendimiento que podríamos tener en nuestro cluster. Si ponemos un número más bajo de la cuenta, no podrán verse entre sí las máquinas que tengan más routers intermedios que los indicados en este parámetro menos uno.

**Stricter security on openMosix ports**: Esta opción permite un chequeo adicional sobre los paquetes recibidos en el puerto de openMosix, y unas comprobaciones adicionales del remitente. Aunque esto suponga una pequeña pérdida de rendimiento, permite evitar que mediante el envio de paquetes quebrantados se pueda colgar un nodo del cluster. De hecho, hasta hace poco tiempo se podía colgar un nodo del antiguo proyecto Mosix sólo haciéndole un escaneo de puertos UDP. Salvo que tengamos mucha seguridad en lo que estamos haciendo, debemos activar esta opción de compilación.

**Level of process-identity disclosure (0-3)**: Este parámetro indica la información que va a tener el nodo de ejecución real de la tarea sobre el proceso remoto que está ejecutando. Aquí debemos destacar que esta información siempre estará disponible en el nodo raíz -en el nodo en el que se originó la tarea-; limitamos la información sólo en el nodo en el que se ejecuta la tarea si este es distinto del nodo raíz. Este es un parámetro de compromiso: valores más bajos aseguran mayor privacidad, a cambio de complicar la administración. Valores más altos hacen más cómoda la administración del cluster y su uso, pero en algunos escenarios pueden violar la política de privacidad del departamento o de la empresa.

Un **0** significa que el nodo remoto que ejecuta el proceso migrado no tiene ninguna información relativa al proceso migrado que se ejecuta en dicho nodo. Este modo paranoico hace la administración del cluster realmente complicada, y no hay ninguna razón objetiva para recomendarlo.

Un **1** significa que el nodo remoto que ejecuta el proceso migrado tiene como única información el PID del proceso. Este es un modo paranoico, pero que permite al menos al administrador del cluster saber con un poco de más comodidad qué es lo que está pasando en caso de problemas. Es un nodo útil cuando usamos máquinas no dedicadas que estén en los despachos de los usuarios del cluster, y no queremos protestas entre los usuarios del cluster sobre quién está haciendo más uso del cluster.

Un **2** significa que el nodo remoto que ejecuta el proceso migrado conoce PID , el usuario propietario y el grupo propietario del proceso. Este es un modo útil en clusters dedicados y no dedicados cuando sabemos que no va a haber discusiones entre los usuarios porque alguien use los recursos del cluster más de la cuenta. Es una

buena opción si tenemos nodos no dedicados en despachos de usuarios donde cualquier usuario no tiene cuentas en las máquinas de los otros, para asegurar una cantidad razonable de privacidad.

Un **3** significa que en el nodo remoto que ejecuta el proceso migrado se tiene exactamente la misma información de los procesos migrados que de los procesos locales. Esto significa que para la información de los procesos el sistema se comporta realmente como un sistema SSI. Este modo es recomendable en los escenarios donde todos los usuarios tienen cuentas en todas las máquinas del cluster, con lo que mantener la privacidad del espacio de procesos ya es de por si imposible, y bloquear esta información solo complica el uso y la administración del cluster. Este es el escenario más habitual de uso de un cluster, por lo que en caso de dudas es mejor que usemos este nivel de privacidad. De cualquier forma, cualquier proceso puede variar su nivel particular de privacidad grabando desde el propio proceso su nuevo nivel de privacidad en el archivo */proc/self/disclosure*.

**Create the kernel with a *-openmosix* extension**: Si activamos esta opción, el nombre simbólico del kernel llevará la extensión *-openmosix*. Esto es importante a la hora de buscar y enlazar los módulos. Debemos activar esta opción si, teniendo la misma versión de kernel base y de kernel para openMosix, queremos que los módulos del kernel openMosix y los módulos del kernel antiguo no sean compartidos. En nuestro caso significa que si activamos esta opción podemos tener un kernel 2.4.20 en el sistema y un kernel openMosix 2.4.20-2 al mismo tiempo, y que cada uno tendrá sus módulos por separado y guardados en directorios distintos. En principio, es una buena idea activar esta opción para evitar problemas de efectos laterales con un kernel ya existente en el sistema.

**openMosix File-System**: Si activamos esta opción podremos usar el sistema de ficheros oMFS, por lo que debemos activar esta opción sólo si vamos a usar el oMFS.

**Poll/Select exceptions on pipes**: Esta opción es interesante, aunque separa a openMosix de una semántica SSI pura. En Unix, un proceso que escriba en un pipe en principio no es interrumpido si otro proceso abre el mismo pipe para leer o ya lo tenía abierto y lo cierra. Activando esta opción nos separamos de Posix: un proceso escritor en un pipe puede recibir una excepción cuando otro proceso abra un pipe para leer dicho pipe, y puede recibir también una excepción si el pipe se queda sin lectores.

Activamos el lanzamiento de la excepción de lectura del pipe con la llamada al kernel *ioctl(pipefd, TCSBRK, 1)*, y activamos la señal de que el último lector ha cerrado el pipe con *ioctl(pipefd, TCSBRK, 2)*. Por último, podemos tener una estimación aproximada de la cantidad de información que los procesos lectores han pedido leer de un pipe en particular con la llamada al sistema *ioctl(pipefd, TIOCGWINSZ, 0)*. Esta llamada no da un valor exacto, y puede equivocarse -pensemos que nos da apenas una estimación a la baja-. Por lo tanto, en caso de equivocación de la llamada, suele ser porque el proceso lector lee más de lo estimado. Aunque activemos esta opción, por defecto, el envío de estas excepciones está desactivado para todos los procesos, y cada proceso que quiera usar estas excepciones debe activar su posibilidad con `ioctl`. En principio no activamos esta opción, salvo que queramos usarla para nuestros propios programas.

**Disable OOM Killer (NEW)**: Las últimas versiones del kernel de Linux incluyen una característica bastante discutida: el *OOM Killer*. Esta opción nos permite inhabilitar el OOM Killer, y evitar los problemas que este suele causar. En caso de duda, es recomendable habilitar esta opción -es decir, inhabilitar el *OOM Killer*-.

Por último, no debemos olvidar que todos los nodos del cluster deben tener el mismo tamaño máximo de memoria, o si no las tareas no migrarán. Para ello, entramos en la opción *Processor type and features*, y en la opción *High Memory Support* asignamos el mismo valor a todos los nodos del cluster.

Después de esto, nuestro kernel estará listo para poder usar openMosix. Ahora seleccionamos las opciones adicionales del kernel que coincidan con nuestro hardware y el uso que le queramos dar a nuestro Linux, grabamos la configuración y hacemos:

```
make dep
```

lo que calcula las dependencias entre partes del kernel -qué se compila y qué no se compila, entre otras cosas-. Después limpiamos el kernel de restos de compilaciones anteriores, que pudieran tener una configuración distinta:

```
make clean
```

Compilamos el kernel:

```
make bzImage
```

Compilamos los módulos:

```
make modules
```

Instalamos los módulos:

```
make modules_install
```

y ahora copiamos el nuevo kernel en el directorio */boot*:

```
cp arch/i386/boot/bzImage /boot/kernelopenMosix
```

por último, creamos una entrada en *lilo.conf* para el nuevo kernel; por ejemplo:

```
image=/boot/kernelopenMosix
        label=om
        root=/dev/hda1
        initrd=/boot/initrd.img
        append=" devfs=mount"
        read-only
```

donde */dev/hda1* es el dispositivo de bloques donde encontramos el directorio raíz de Linux; en nuestro sistema puede cambiar. Compilamos la tabla del LILO con el comando:

```
lilo
```

y listo. Ya tenemos un kernel listo para poder usarlo en un nodo openMosix.

**Errores más frecuentes al compilar e instalar el kernel**

Los errores más frecuentes a la hora de configurar el kernel de openMosix son:

**Las aplicaciones no migran de todos los nodos a todos los nodos:** La causa de este error suele ser que hemos olvidado poner en todas las máquinas el mismo tamaño máximo de memoria.

**El parche no se aplica correctamente:** Tenemos que usar un *vanilla kernel*, es decir, un kernel tal y como Linus Torvalds lo distribuye. Particularmente, los kernels que vienen con las distribuciones de Linux no valen ya que están demasiado parcheados.

**No existe el directorio*/proc/hpc*:** O no hemos arrancado con el kernel openMosix, o se nos ha olvidado activar la primera opción de *openMosix process migration support* al compilar el kernel.

**Uso la ultimísima versión del gcc, y el kernel de openMosix no compila:** No es un problema de openMosix, es un problema del kernel de Linux. Cualquier versión de gcc que compile el kernel de Linux compila el kernel de openMosix; y casi todas las distribuciones de Linux modernas traen un paquete con una versión del backend de gcc para compilar el kernel de Linux.

Además de estos problemas, tendremos los problemas habituales de instalar un kernel nuevo: olvidarnos de ejecutar el comando *lilo*, olvidarnos de instalar los módulos... de cualquier forma, si sabemos recompilar el kernel e instalar un kernel nuevo no tendremos ningún problema con el kernel de openMosix.

### 5.3.2. Instalación de las herramientas de área de usuario

Para configurar e instalar correctamente las herramientas de área de usuario no es necesario recompilar el kernel de openMosix; pero es fundamental tener descargado el kernel de openMosix, ya que necesitaremos sus cabeceras para compilar correctamente las herramientas de área de usuario.

El primer paso para instalar las herramientas de área de usuario del proyecto openMosix es descargarlas. Podemos descargar las herramientas de área de usuario de http://www.orcero.org/irbis/openmosix .

Podemos descargar también las herramientas de área de usuario de Sourceforge.

Descargamos la última versión -en la fecha en la que se escribe este documento, la última versión de las herramientas de área de usuario son las 0.2.4-. La descomprimimos con:

```
tar -zxf openMosixUserland-0.2.4.tgz
```

entramos en el directorio creado:

```
cd openMosixUserland-0.2.4
```

y, antes de cualquier otro paso, leemos el archivo de instalación:

```
cat Installation | more
```

para la 0.2.4, la información que tenemos en el es la misma que veremos en este capítulo; pero versiones posteriores de las herramientas de área de usuario pueden tener opciones de configuración distintas.

### Configurando las herramientas de área de usuario

El paso siguiente para configurar las herramientas de área de usuario será editar el archivo *configuration* con nuestro editor de textos favorito. Hay una opción que debemos modificar obligatoriamente si queremos recompilar openMosix, y otras opciones que no debemos tocar salvo que estemos muy seguros de lo que queremos hacer.

Para una instalación estándar de openMosix en los directorios estándares apenas debemos modificar el valor de la variable *OPENMOSIX*. Esta variable debe contener el camino completo absoluto -no el camino relativo- del kernel de openMosix. Por ejemplo, */usr/src/openmosix* es un camino válido, y */home/irbis/openMosix/linux-openmosix* también lo es. *../../linux-openmosix* no es un camino válido, ya que es un camino relativo, y debe ser un camino absoluto.

En casi todos los casos, ahora solo tenemos que hacer:

```
make all
```

y los *Makefile* que acompañan a openMosix se encargarán del resto: compilarán e instalarán las herramientas de área de usuario de openMosix y sus páginas del manual. A pesar de que el código nuevo tiene muy pocos warnings y no tiene errores -al menos, que se sepa-, el código antiguo heredado de Mosix esta programado de una forma muy poco ortodoxa -por decirlo de una forma educada-, por lo que hay activadas todas las opciones de warning para buscar los errores existentes en el código y eliminarlos.

Una vez que la ejecución del *make all* esté terminada, las herramientas de área de usuario estarán instaladas y bien configuradas.

### Otras opciones de configuración

La opción que hemos visto es la única que deberemos modificar manualmente en condiciones normales. Pero hay otras opciones que pueden ser interesantes para algunos usuarios con necesidades muy específicas. Otras opciones del archivo *configuration* son:

**MONNAME:** nombre del programa monitor de openMosix. El programa monitor del proyecto Mosix se llamaba `mon`, lo que era un problema ya que compartía nombre con una herramienta muy común en Linux de monitoramiento del sistema. La gente de Debian lo solucionó cambiando el nombre de la aplicación para Debian de `mon` a `mmon`; pero en openMosix la aplicación la llamamos `mosmon`. En principio, es recomendable dejarlo en `mosmon`, salvo que por razones de compatibilidad inversa con algun script de Mosix queramos llamar a esta aplicación `mon` o `mmon`.

**CC:** Nombre del compilador de C. Casi siempre es `gcc`.

**INSTALLBASEDIR:** Ruta completa absoluta donde está el directorio raíz del sistema de ficheros donde queremos instalar openMosix. Casi siempre es / .

**INSTALLEXTRADIR:** Ruta completa absoluta donde está el directorio donde están los directorios con las aplicaciones del sistema y su documentación donde queremos instalar openMosix. Casi siempre es */usr* .

**INSTALLMANDIR:** Ruta completa absoluta donde está el directorio donde están las páginas del manual donde queremos instalar openMosix. Casi siempre es */usr/man* .

**CFLAGS:** Opciones de compilación en C para las herramientas de área de usuario. Las opciones *-i./*, *-i/usr/include* y *-i$(OPENMOSIX)/include* son obligatorias; el resto la pondremos según nuestro interés particular -las opciones por defecto que encontramos en el archivo de configuración son válidas, y debemos mantenerlas salvo que haya una buena razón para no hacerlo-.

**Errores frecuentes en la compilación e instalación de las herramientas**

Hay un conjunto de errores en la instalación de las herramientas de área de usuario que debemos conocer bien, ya que son objeto de frecuentes preguntas en la lista de correos. Estos errores son:

**Las herramientas de área de usuario no compilan:** ¿Hemos puesto correctamente el valor de la variable `OPENMOSIX` en el archivo *configuration*? ¿Esta variable realmente apunta al kernel de openMosix? ¿Es realmente el kernel de openMosix, o un kernel normal? Recordemos que la variable `OPENMOSIX` debe ser forzosamente un camino absoluto. Por último, ¿tenemos instalado un compilador de C?

**Error *This is not Mosix!*** Este error se da cuando usamos las herramientas de área de usuario del proyecto Mosix sobre un kernel normal, o sobre un kernel openMosix. También se da si usamos el `setpe` de una versión de las herramientas de área de usuario anterior a la 0.2.4 sobre un kernel que no tiene soporte openMosix. Si es este último caso, es muy recomendable usar una versión de herramientas de área de usuario 0.2.4 o posterior; la 0.2.4 es conocida por ser realmente estable, y no hay ninguna razón objetiva para usar una versión anterior. En cualquier otro caso, recordemos que las herramientas de área de usuario del proyecto Mosix no son libres; y que, de cualquier forma, no funcionan en openMosix.

**Error *This is not openMosix!*** Este error se da cuando usamos las herramientas de área de usuario de openMosix en un equipo que no está usando un kernel con soporte openMosix. O estamos usando las herramientas de área de usuario de openMosix sobre un kernel Mosix, o no hemos arrancado con el kernel nuevo, o el kernel nuevo no tiene soporte a migración de procesos openMosix.

### 5.3.3. Configurando la topología del cluster

Una vez que ya tenemos instalado nuestro kernel de openMosix y nuestras utilidades de área de usuario de openMosix, el siguiente paso que daremos será configurar la topología del cluster.

Para configurar la topología del cluster openMosix tenemos dos procedimientos distintos.

1. El primero es el procedimiento tradicional, consistente en un archivo por nodo donde se especifican las IPs de todos los nodos del cluster.

2. El segundo procedimiento consiste en emplear el demonio de autodetección de nodos.

Cada procedimiento tiene sus ventajas y sus inconvenientes.

La configuración tradicional manual tiene como inconveniente que es más laboriosa en clusters grandes: cada vez que queramos introducir un nodo nuevo en el cluster, debemos tocar el archivo de configuración de todos los nodos de dicho cluster para actualizar la configuración. Este método nos permite, por otro lado, tener topologías arbitrariamente complejas y grandes; también es el método más eficiente.

El segundo método para configurar los nodos de un cluster openMosix es usar el demonio de detección automática de nodos, el *omdiscd*. Este método es el más cómodo, ya que el cluster casi se configura solo. Por otro lado, tiene dos pegas:

1. la primera, que todas las máquinas del cluster deben estar en el mismo segmento físico. Esto impide que el demonio de autodetección pueda ser usado en redes muy complejas.

2. La segunda, que todos los nodos cada cierto tiempo deben mandar un paquete de broadcast a todos los otros nodos de la red para escanear los nodos openMosix de la red.

Para pocos nodos, estos paquetes no afectan al rendimiento; pero en caso de clusters de tamaño medio o grande, la pérdida de rendimiento puede ser crítica.

Realmente el demonio de autodetección de nodos no detecta nodos openMosix, sino que detecta otros demonios de autodetección de nodos. Esto significa en la práctica que el demonio de autodetección de nodos no es capaz de detectar nodos openMosix configurados mediante el método manual. Tampoco debemos mezclar los dos tipos de configuración en el cluster, ya que esto nos dará bastantes dolores de cabeza en la configuración de la topología.

#### Configuración automática de topología

La forma automática de configurar la topología de un cluster openMosix es mediante un demonio recientemente Incorporado en las herramientas de área de usuario, que permite la detección automática de nodos del cluster.

El nombre del demonio es *omdiscd*. Para ejecutarlo, si hemos instalado correctamente las herramientas de área de usuario, basta con hacer:

```
omdiscd
```

y este programa creará automáticamente una lista con las máquinas existentes en la red que tienen un demonio de autodetección de nodos válido y funcionando correctamente, e informará al kernel openMosix de estas máquinas para que las tenga en cuenta.

Podemos consultar la lista generada por el demonio de autodetección de nodos con el comando:

```
showmap
```

este comando muestra una lista de los nodos que han sido dados de alta en la lista de nodos local al nodo que ejecuta el demonio *omdiscd*. Cuidado, ya que el hecho de que un nodo sea reconocido por un segundo no implica el caso recíproco: alguno de los nodos de la lista pueden no habernos reconocido aún como nodo válido del cluster.

Podemos informar a openMosix sobre por cual de los interfaces de red queremos mandar el paquete de broadcast. Esto es especialmente interesante en el caso particular de que el nodo donde lanzaremos el demonio de autodetección de nodos no tenga una ruta por defecto definida, caso en el que *omdiscd* parece fallar para algunos usuarios; aunque hay otros escenarios donde también es necesario controlar manualmente por que interfaz de red se manda el paquete de broadcast. Para forzar a *omdiscd* a mandar la información por un interfaz en particular tenemos que usar la opción `-i`. Por ejemplo, llamamos al demonio de autodetección de nodos en este caso con el comando:

```
omdiscd -i eth0,eth2
```

lo que significa que mandaremos el paquete de broadcast por `eth0` y por `eth2`, y solamente por estos dos interfaces de red.

Otro caso particular que es interesante conocer es el de algunas tarjetas PCMCIA que por un error en el driver del kernel no sean capaces de recibir correctamente los paquetes de broadcast -existen algunas así en el mercado-. La única solución que podemos tener en la actualidad es poner el interfaz de dicha tarjeta con un mal driver en modo promiscuo, con lo que la tarjeta leerá y analizará todos los paquetes, incluidos los de broadcast; y el así el kernel podrá acceder a dichos paquetes de broadcast. No es un problema del código de openMosix, sino de los drivers de algunas tarjetas; pero el demonio de autodetección de nodos lo sufre directamente. Ponemos la tarjeta con un driver que tenga problemas con los paquetes de broadcast en modo promiscuo con:

```
ifconfig eth0 promisc
```

suponiendo que `eth0` es nuestro interfaz de red. En otro caso, sustituiremos `eth0` por nuestro interfaz de red. En caso de que el ordenador tenga varios interfaces de red, usamos esta instrucción con todos aquellos interfaces por los que esperamos recibir paquetes de openMosix -puede ser más de un interfaz- y que tengan este problema. Sólo root puede poner en modo promiscuo un interfaz.

Para verificar con comodidad la autodetección viendo que hace el demonio de autodetección de nodos, podemos lanzarla en primer plano con la opción `-n`, con la sintaxis:

```
omdiscd -n
```

así se lanzará en primer plano, y veremos en todo momento lo que está pasando con el demonio.

Otro modificador que es interesante conocer en algunos escenarios es `-m`. Lleva como parámetro un único número entero, que será el TTL de los paquetes de broadcast que enviemos a otros demonios de autodetección de nodos.

Se aconseja pues que usemos la herramienta de detección automática de nodos sólo para clusters de prueba. A pesar de el gran interés que han mostrado Moshe Bar y Martin Hoy en ella, al hacerle pruebas algunas veces no ha detectado todos los nodos. Es una herramienta aún experimental, y esto es algo que no debemos olvidar.

Finalmente debemos destacar que este demonio debe ser lanzado en todos los nodos del cluster, o en ninguno de ellos. Mezclar los dos mecanismos de configuración de la topología de un cluster en el mismo cluster no es una buena idea.

### Configuración manual de topología

El sistema de autodetección tiene muchos problemas que lo hacen inconveniente para determinadas aplicaciones. No funciona si hay algo entre dos nodos que bloquee los paquetes de broadcast, puede sobrecargar la red si tenemos muchos nodos en el cluster, supone un demonio más que debe correr en todos los nodos del cluster, lo que complica la administración; y, quizás lo más importante, aún es software beta y no siempre detecta todos los nodos. Por todo ello, muchas veces un fichero compartido de configuración, común a todos los nodos, es la solución más simple a nuestro problema.

En openMosix llamamos a nuestro fichero */etc/openmosix.map*. El script de arranque de openMosix -que estudiaremos más adelante- lee este archivo, y lo utiliza para informar al kernel de openMosix sobre cuales son los nodos del cluster.

El fichero */etc/openmosix.map* contiene una lista con los rangos de direcciones IP que pertenecen al cluster. Además de indicar que rangos de direcciones IP pertenecen al cluster, nos permite asignar un número de nodo único a cada IP de la red. Este número será empleado internamente por el kernel de openMosix y por las herramientas de usuario; también lo emplearemos como identificador en comandos como `migrate` para referenciar de forma unívocamente cada nodo. Tanto el sistema de ficheros */proc/hpc* como las herramientas de área de usuario usan estos números identificativos de nodo, en lugar de la IP, ya que un nodo del cluster openMosix puede tener más de una IP, y solo uno de estos números.

Cada linea del fichero */etc/openmosix.map* corresponde a un rango de direcciones correlativas que pertenecen al cluster. La sintaxis de una linea es:

```
numeronodo IP tamañorango
```

donde `numeronodo` es el primer número de la primera IP del rango, `IP` es la primera IP del rango, y `tamañorango` es el tamaño del rango.

Por ejemplo, en el archivo */etc/openmosix.map* con el contenido:

```
1 10.1.1.100 16
17 10.1.1.200 8
```

estamos diciendo que nuestro cluster openMosix tiene 24 nodos. En la primera linea decimos que los 16 nodos primeros, que comenzamos a numerar por el número 1, comienzan desde la IP 10.1.1.100; y continúan con 10.1.1.101, 10.1.1.102... así hasta 10.1.1.115.

En la segunda linea decimos que, comenzando por el nodo número 17, tenemos 8 nodos más; comenzando por la IP 10.1.1.200, 10.1.1.201...hasta la IP 10.1.1.207.

Podemos también Incluir comentarios en este fichero. A partir del carácter #, todo lo que siga en la misma linea del carácter # es un comentario. Por ejemplo, podemos escribir:

```
# redes 10.1.1
 1 10.1.1.100 16 # Los 16 nodos del laboratorio
17 10.1.1.200 8 # El core cluster
```

Este archivo es exactamente igual para openMosix que el archivo anterior.

La sintaxis de la declaración de la topología del cluster en el archivo */etc/openmosix.map* necesita una explicación adicional cuando tenemos alguna máquina con más de una dirección IP. En este archivo deben aparecer todas las IPs que puedan enviar y recibir paquetes openMosix, y sólo ellas. Esto significa que no pondremos en este archivo las IPs que no se usen para enviar y recibir los mensajes; pero también supone un problema cuando una misma máquina puede enviar y recibir mensajes de openMosix por varias IPs distintas. No podemos poner varias entradas como las que hemos visto, ya que el identificador de nodo es único para cada nodo, independientemente del número de IPs que tenga un nodo.

Por todo esto, para solucionar el problema de que un nodo tenga varias direcciones IPs válidas en uso en el cluster openMosix, tenemos la palabra clave ALIAS, que usamos para indicar que la definición de la dirección IP asignada a un número identificador de nodo está replicada porque dicho identificador tiene más de una IP válida.

Lo vamos a ver más claro con un ejemplo. Supongamos, por ejemplo, que un nodo particular en el cluster descrito en el ejemplo anterior -el nodo 8- comparte simultáneamente una dirección de las 10.1.1.x y otra de las 10.1.2.x. Este segmento 10.1.2.x tiene a su vez más nodos openMosix con los que tenemos que comunicarnos. Tendríamos el fichero:

```
# redes 10.1.1
 1 10.1.1.100 16 # Los 16 nodos del laboratorio
17 10.1.1.200 8 # El core cluster

# redes 10.1.2
18 10.1.2.100 7
8 10.1.2.107 ALIAS # Nodo de interconexion con la red 10.1.1
25 10.1.2.108 100
```

es decir, estamos definiendo la IP del nodo 8 dos veces. La primera vez en la línea:

```
 1 10.1.1.100 16 # Los 16 nodos del laboratorio
```

y la segunda vez en la linea:

```
8 10.1.2.107 ALIAS # Nodo de interconexion con la red 10.1.1
```

en la primera línea el nodo 8 está definido dentro del rango entre el nodo 1 y el nodo 16. En la Segunda línea vemos como decimos con ALIAS que el nodo 8, además de la IP ya definida, tiene una IP adicional: la IP 10.1.2.107.

Hay que destacar un concepto clave al usar ALIAS: tenemos que separar forzosamente la IP de la entrada ALIAS del rango donde esta ha sido definida. Por ejemplo, en el escenario anteriormente comentado es erróneo hacer:

```
# redes 10.1.1
 1 10.1.1.100 16 # Los 16 nodos del laboratorio
17 10.1.1.200 8 # El core cluster
# redes 10.1.2
18 10.1.2.100 108
8 10.1.2.107 ALIAS # Nodo de interconexion con la red 10.1.1
```

Esto no funciona, ya que definimos un identificador para la IP 10.1.2.107 dos veces. Por ello, tenemos que resolver este escenario como en el ejemplo completo anteriormente comentado.

Por último, debemos recordar que todos los nodos del cluster openMosix deben tener el mismo archivo */etc/openmosix.map*, y de que la dirección local 127.0.0.1, por lo tanto, jamás debe aparecer en el archivo */etc/openmosix.map*, ya que los ficheros deben ser iguales; y, en caso de incluir la dirección IP local 127.0.0.1, cada archivo asignaría al mismo número distinto una máquina distinta.

#### El script de inicialización

Podemos activar el archivo anteriormente citado con el comando `setpe` -más tarde veremos cómo hacerlo-; pero en openMosix tenemos un mejor procedimiento para configurar la topología del cluster: su script de inicialización.

En el directorio de instalación de las herramientas de área de usuario tenemos un subdirectorio llamado *scripts*. En este directorio tenemos un script que se llama *openmosix*, que es el script encargado de configurar nuestro nodo cuando este entre en el runlevel que determinemos, suponiendo scripts de configuración *la System V* -que son los que usan casi todas las distribuciones modernas de Linux-. Para activarlo, entramos en el directorio *scripts* y hacemos:

```
cp openmosix /etc/rc.d/init.d
```

ahora tenemos que decidir en qué runlevels queremos lanzar openMosix. Si queremos lanzar openMosix en el runlevel 3, hacemos:

```
ln -s /etc/rc.d/init.d/openmosix /etc/rc.d/rc3.d/S99openmosix
```

y si queremos lanzarlo en el runlevel 5 hacemos:

```
ln -s /etc/rc.d/init.d/openmosix /etc/rc.d/rc5.d/S99openmosix
```

Si queremos lanzar openMosix al arrancar una máquina, debemos determinar cual es el runlevel de arranque del nodo. Para saber cual es el runlevel de arranque de nuestra máquina, hacemos:

```
cat /etc/inittab | grep :initdefault:
```

o

```
cat /etc/inittab | grep id:
```

saldrá algo como:

```
id:5:initdefault:
```

en este caso, el runlevel de arranque será el 5, y será el runlevel donde tendremos que activar el script de openMosix como hemos visto anteriormente. Por otro lado, si sale:

```
id:3:initdefault:
```

el runlevel de arranque será el 3, y usaremos el comando anteriormente estudiado para lanzar el script de inicialización en el runlevel 3.

La próxima vez que la máquina arranque, openMosix estará correctamente configurado. De cualquier forma, podemos en cualquier momento reiniciar la configuración de openMosix sin reiniciar la máquina haciendo:

```
/etc/rc.d/init.d/openmosix restart
```

**Migrando tareas**

En principio, los procesos migrarán solos según el algoritmo de equilibrado automático de carga. Sin embargo, podemos tener interés en recomendar una migración, o en forzar que un proceso vuelva a su nodo raíz. Vamos a repasar las utilidades de área de usuario; y comenzaremos con la utilidad que permite controlar las migraciones: la utilidad `migrate`.

Esta utilidad nos permite solicitar la migración de un proceso determinado a un nodo determinado. Su sintaxis es:

```
migrate PID numnodo
```

donde `PID` es el PID del proceso, y `numnodo` el número del nodo al que queremos que el proceso migre. Si queremos forzar que el proceso migre a su nodo raíz, hacemos:

```
migrate PID home
```

por otro lado, si queremos que el proceso migre a un nodo indeterminado que el kernel de openMosix debe decidir según el algoritmo de equilibrado automático de carga, hacemos:

```
migrate PID balance
```

sólo puede solicitar la migración de una tarea root, el usuario propietario de la tarea y el usuario efectivo de la tarea. La migración es solo solicitada; y la tarea puede no migrar si hay alguna razón de fuerza mayor que impida la migración. Particularmente, la única migración de cualquier proceso que tendremos completa seguridad de que se realizará es la migración al nodo raíz con el parámetro *home*. Las otras pueden no ocurrir.

### 5.3.4. Las herramientas de área de usuario

**Monitorizando el cluster**

La herramienta usada para monitorizar un cluster openMosix es `mosmon`. Esta herramienta nos permite ver un gráfico de barras con la carga asociada a cada nodo del cluster. Esta información podríamos obtenerla haciendo un `top` en cada uno de los nodos, pero para clusters de más de ocho nodos la solución del `top` es inviable, y la solución de `mosmon` es especialmente buena.

`mosmon` es una utilidad especialmente interesante por varios puntos adicionales, que no todos sus usuarios conocen, y que lo hacen indispensable para cualquier administrador de sistemas: podemos ver las barras de forma horizontal y vertical, podemos listar todos los nodos definidos en el cluster, Estén o no activos, podemos ver el número de nodos activos y además, en caso de que el número de nodos sea mayor del que se puede ver en una pantalla, podemos con el cursor derecho y el cursor izquierdo movernos un nodo a la derecha o un nodo a la izquierda, con lo que podremos ver grandes cantidades de nodos en una pantalla cualquiera. Todo esto hace a `mosmon` una herramienta realmente imprescindible para el administrador del cluster.

De entre las opciones disponibles que podemos usar al llamar a `mosmon`, las más importantes son:

**-d:** incluye en el gráfico todos los nodos, incluso aquellos que están desactivados. Esta opción es muy útil, ya que así el administrador ve cuando entran en el cluster estos nodos desactivados.

**-t:** lista un el número de nodos activos del cluster. Esta opción es especialmente útil en clusters grandes o muy grandes, para hacernos una idea de cuantos nodos activo realmente tenemos -recordemos que en clusters realmente grandes, todos los días falla el hardware de algún nodo-.

Una vez que estamos dentro del programa `mosmon`, con la pulsación de algunas teclas podemos conseguir funciones extra. De entre las teclas activas de `mosmon`, destacamos:

**d:** lista también aquellos nodos que no están activos. Hace lo mismo que arrancar `mosmon` con la opción `-d`.

**D:** lista sólo aquellos nodos que están activos. Por ello, hace lo mismo que arrancar `mosmon` sin la opción `-d`.

**h:** muestra la ayuda de `mosmon`.

**l:** permite visualizar la carga de cada nodo. Este es el modo de visualización con el que arranca `mosmon`, por lo que esta opción se usa para volver a la visualización de carga después de haber cambiado la vista con `m`, `r`, `s`, `t` o `u`.

**m:** permite visualizar la memoria lógica usada por los procesos -lo que corresponde a suma de las memoria que los procesos creen que realmente usan- y la memoria total por cada máquina, en lugar de la carga. La barra corresponde con la memoria lógica ocupada, mientras que los +, sumados a la barra, corresponden a la memoria total. Los + corresponden a la suma entre la memoria libre y la memoria usada por cosas distintas de los procesos -kernel, Reservada por dispositivos hardware-. Puede ser menor O mayor que la memoria libre real, ya que por un lado varios procesos pueden compartir segmentos, lo que hace que la memoria física usada sea menor que la memoria lógica usada; por otro lado, el kernel y los dispositivos hacen que no toda la memoria no usada por los procesos esté realmente libre. Todas las cantidades se muestran en megabytes.

**q:** sale del programa `mosmon`.

**r:** permite visualizar la memoria física usada, la memoria libre y la memoria total por cada máquina, en lugar de la carga. La barra corresponde con la memoria física usada en un nodo, mientras que los + corresponden a la memoria libre. Por ello los +, sumados a la barra, corresponden a la memoria total. Todas las cantidades se muestran en megabytes.

**s:** permite ver las velocidades de los procesadores y el número de procesadores por cada máquina en lugar de la carga.

**t:** lista un el número de nodos activos del cluster, si no está visible, o desactiva la visión si se están viendo el número de nodos activos del cluster. Está relacionado con la opción de arranque `-t`.

**u:** permite ver el grado de utilización del procesador de cada nodo. En el caso de que el cuello de botella de un nodo esté en su procesador, este valor estará al 100 %. Hay que destacar que un nodo puede estar saturado por muchas cosas, tales como acceso a disco o a swap, y no llegar dicho nodo al 100 % de la utilización neta del procesador. Un valor por debajo del 100 %, por lo tanto, significa que un procesador está infrautilizado, por lo que podría aceptar migraciones de entrada -aunque puede tener migraciones de salida del nodo de procesos que usen mucha memoria o mucho disco-.

Además de todo esto, la tecla *Enter* nos permite redibujar la pantalla, `p` hace lo mismo que el cursor izquierdo -mover la vista de nodos a la izquierda-, y `n` nos permite hacer lo mismo que el cursor derecho -mover la vista de nodos a la derecha-.

#### Configurando los nodos en openMosix

La herramienta para configurar los nodos de un cluster openMosix es `setpe`. Esta herramienta es llamada por los scripts de inicialización y parada de openMosix, así como por numerosos scripts de openMosix y herramientas auxiliares. A pesar de que habitualmente no la llamaremos directamente, es interesante su estudio.

`setpe` se encarga de determinar la configuración de nodos del cluster. Su parámetro principal es el archivo donde estará especificado el mapa de nodos. `setpe` habitualmente es llamado de tres formas; la primera es con el modificador `-f nombrefichero`, que tomará como archivo de configuración `nombrefichero`. La segunda forma es pasando como parámetro `-`, en cuyo caso leerá el archivo de configuración de la entrada estándar. Esto es útil para hacer pruebas de configuración, o para hacer pipes de `setpe` con otras aplicaciones. Por último, puede ser llamado con un único parámetro, `-off`, para sacar el nodo del cluster.

De entre los parámetros de `setpe` destacamos:

**-w:** carga la configuración del fichero indicado con los parámetros especificados en dicho fichero sobre el kernel, si es posible hacerlo sin necesidad de reinicializar la parte de openMosix del kernel. Esto significa que sólo actualiza la configuración si el nodo no ejecuta ningún proceso de otro nodo, ni ningún proceso de el nodo

local se ejecuta en un nodo remoto. En cualquiera de estos dos casos, `-w` dará un error y no actualizará la configuración.

**-W:** carga la configuración del fichero indicado con los parámetros especificados en dicho fichero sobre el kernel, cueste lo que cueste. Esto puede significar expulsar procesos y mandarlos a sus nodos de vuelta, asícomo traerse al nodo local todos los procesos remotos lanzados localmente. Internamente, con un `-W`, realmente `setpe` hace primero un `-off` -vemos más adelante cómo funciona -off-, y después hace un `-w`.

**-c:** realiza toda la operativa que realizaría `-w`, solo que no graba el resultado en el kernel. Esta opción es útil para verificar una configuración sin instalarla en la máquina.

Cualquiera de estas tres opciones puede ser acompañada por la opción `-g numero`. Si la utilizamos, `septe` informará al kernel que el número máximo de gateways que se interponen entre el nodo local y el más remoto de los nodos es, a lo sumo, `número`. Si no indicamos esta opción, simplemente no modificamos el parámetro del kernel de número máximo de gateways, quedando como estaba. El número máximo de gateways intermedio que podemos especificar es `2`.

Además de estas opciones, tenemos otras opciones interesantes de `setpe`. Estas son:

**-r:** lee la configuración del nodo actual, y la vuelca en la salida estándar, o en un archivo determinado con la opción `-f nombrearchivo`. Esta opción es muy útil para ver errores en la configuración en clusters muy grandes, en los que no hemos hecho un seguimiento de la configuración de todos y cada uno de sus nodos y hemos empleado cualquier mecanismo automatizado de configuración.

**-off:** esta opción desactiva openMosix en el nodo actual, es decir, bloquea las migraciones de salida de procesos locales, bloquea las migraciones de entrada de procesos remotos, manda las tareas que se ejecutan en el nodo actual de otros nodos de vuelta a sus nodos de origen, llama a los procesos remotos originados en el nodo local para que vuelvan, borra la tabla de nodos del nodo, y, por último, inhabilita openMosix en el nodo local.

#### Controlando los nodos con `mosctl`

Del mismo modo que `setpe` nos permite ver la configuración de un nodo openMosix y configurarlo, `mosctl` nos permite editar el comportamiento de un nodo ya configurado, y verlo.

De entre las opciones del comando `mosctl`, destacamos:

**block:** en el nodo local, bloquea la entrada de los procesos generados en otro nodo.

**noblock**: deshace los efectos de `block`.

**mfs:** activa el soporte a MFS en openMosix.

**nomfs:** inhabilita el soporte a MFS en openMosix.

**lstay:** bloquea la migración automática hacia fuera de los procesos generados localmente.

**nolstay:** deshace los efectos de `lstay`.

**stay:** bloquea la migración automática hacia fuera de cualquier proceso.

**nostay:** deshace los efectos de `stay`.

**quiet:** el nodo local no informará a los otros nodos de su status.

**noquiet:** deshace los efectos de `quiet`.

Como vemos, todas estas opciones tienen un modificador que habilita alguna propiedad, y otro modificador que la inhabilita. Hay otros modificadores de un solo comando para mosctl, que son:

**bring:** trae de vuelta a todos los procesos que se han generado localmente pero que se ejecutan en un nodo

remoto. Además, internamente realiza primero la misma operación que `lstay`, y deja el estado en `lstay`. No retorna hasta que no han vuelto todos los procesos remotos generados localmente.

**expel:** manda de vuelta a sus nodos de origen a todos los nodos que se ejecutan en el nodo local pero fueron generados en un nodo remoto. Además, internamente realiza primero la misma operación que `block`, y deja el estado en `block`. No retorna hasta que no han salido todos los procesos remotos del nodo local.

Por ejemplo, para apagar ordenadamente un nodo en un cluster openMosix sin perder ningún proceso, debemos hacer:

```
mosctl expel
```

```
mosctl bring
```

Después de estos comandos, el nodo no aceptará ni que migren al nodo local procesos generados en un nodo externo, ni que ningún nodo local migre a otro nodo. Además, no se ejecutará ningún proceso remoto en el nodo local ni ningún proceso local en el nodo remoto. Es decir, nuestra máquina está desligada del cluster openMosix, por lo que si se apaga la máquina de un tirón de cable no afectará al resto del cluster.

Existen, además de estos, un conjunto de modificadores que tienen un parámetro adicional: el identificador de un nodo dentro del cluster. Estos modificadores permiten obtener información sobre cualquier nodo del cluster. Estos modificadores son:

**getload nodo:** siendo `nodo` un nodo válido en el cluster, este modificador nos devuelve la carga que actualmente tiene dicho nodo. Este parámetro de carga no corresponde al parámetro de carga del kernel al que estamos acostumbrados, sino al parámetro de carga que calcula openMosix y usa openMosix.

**getspeed nodo:** da la velocidad relativa de dicho nodo. Esta velocidad es relativa, y se supone que un Pentium-III a 1GHz es un procesador de 1000 unidades de velocidad.

**getmem nodo:** da la memoria lógica libre y la memoria lógica total de un nodo particular del cluster.

**getfree nodo:** da la memoria física libre y la memoria física total de un nodo particular del cluster. Como estudiamos anteriormente al hablar de `mosmon`, la memoria física libre y la memoria lógica libre pueden no coincidir.

**getutil nodo:** siendo `nodo` un nodo válido en el cluster, nos da el grado de uso de dicho nodo en el cluster. Discutimos el parámetro de grado de uso de un nodo al hablar del comando `mosmon`.

**isup nodo:** nos indica si el nodo indicado está activo o no.

**getstatus nodo:** nos dará el estado del nodo indicado. En este estado incluye también información sobre si el nodo permite migraciones de entrada, si permite migraciones de salida, si bloquea todas las migraciones, si está activo, y si está propagando información sobre su carga.

Por ultimo, tenemos un modificador que nos permite descubrir la IP y el identificador de nodo en openMosix asociado a un nodo en particular. Es:

```
mosctl whois dirección
```

y podemos tener como parámetro `dirección` el identificador de nodo, en cuyo caso este comando nos devolverá la IP, o la IP, en cuyo caso nos devolverá el identificador de nodo, o el nombre de la máquina, en cuyo caso nos devolverá el identificador de nodo. Este comando también nos indica si la máquina indicada no pertenece al cluster openMosix.

Si llamamos a `mosctl whois` sin ningún parámetro adicional, este comando nos devolverá el identificador del nodo local.

### Forzando migraciones

Para forzar una migración en un cluster openMosix, debemos usar el comando `migrate`.

El comando `migrate` toma dos parámetros: el primero es el PID del proceso que queremos hacer migrar, y el segundo parámetro es donde queremos que migre. Este segundo parámetro debe ser el identificador válido de un nodo en el cluster.

Existen, sin embargo, dos parámetros que podemos colocar en lugar del identificador válido de un nodo del cluster. Estos dos modificadores modelan dos casos especiales, y son:

**home:** fuerza a migrar a un proceso al nodo donde fue generado.

**balance:** fuerza a migrar a un proceso al nodo donde la migración suponga minimizar el desperdicio de recursos dentro del cluster openMosix. Es una forma de indicar que se evalúe el algoritmo de migración automática de carga de openMosix, pero dando preferencia a la migración del proceso del que hemos indicado el PID.

A la hora de lanzar esta migración, en caso de que el proceso sea un proceso lanzado en la máquina donde ejecutamos el comando `migrate`, debemos ser el administrador de la máquina, el usuario propietario del proceso, el usuario efectivo del proceso, miembro del grupo propietario del proceso o miembro del grupo efectivo del proceso.

Por otro lado, el administrador del sistema de un nodo cualquiera del cluster siempre puede lanzar este comando sobre cualquier proceso que se ejecute en dicho nodo, independientemente de que se haya generado en el nodo local o en un nodo remoto.

En principio, el proceso puede no migrar aunque le lancemos la orden `migrate`. En caso de que no migre, algunas veces recibiremos un mensaje de error avisando que el comando no funcionó, pero unas pocas veces no migrará, y no recibiremos dicho mensaje. Particularmente esto se da cuando forzamos una migración posible pero pésima: el proceso será mandado de vuelta al nodo local incluso antes de que salga, porque el algoritmo de optimización de carga considerará inaceptable la migración.

La única migración que realmente podemos forzar siempre es la de vuelta a casa, siempre que el nodo de origen no acepte salidas de su nodos con `mosctl lstay` y no bloqueemos la entrada en el nodo de destino con `mosctl block`.

### Recomendando nodos de ejecución

Cuando lanzamos un proceso, podemos indicar como se va a comportar frente a la migración, o donde preferimos que se ejecute. Para ello, contamos con el comando `mosrun`. Este comando no se suele llamar directamente, sino a través de un conjunto de scripts que facilitan su uso. Con este comando podemos transmitir a openMosix la información sobre qué hace el proceso, información que será fundamental para que openMosix minimice el desperdicio de recursos del cluster al mínimo. También podemos indicar un conjunto de nodos entre los cuales estará el nodo donde migrará el proceso después de lanzado, si esta migración es posible. En un sistema que no sea openMosix, `mosrun` lanza el proceso en la máquina local de forma correcta.

El comando `mosrun` siempre tiene la misma estructura de llamada:

```
mosrun donde migracion tipo comando argumentos
```

Donde los parámetros son:
`donde`: nodo al que el proceso va a migrar inmediatamente después de ser lanzado, si esto es posible.
`migracion`: si se bloquea o no el proceso en el nodo de destino.
`tipo`: tipo de proceso que se lanzará.
`comando`: nombre del proceso que se va a lanzar.
`argumentos`: argumentos del proceso que se va a lanzar

El modificador `donde` puede ser:
Un nodo de destino, al que el proceso migrará inmediatamente después de ser lanzado, si esto es posible.

**-h**, en cuyo caso será el nodo local.

**-jlista**: en este caso, inmediatamente después de lanzar el proceso, lo migrará a un nodo escogido aleatoriamente dentro de la lista de rangos `lista`. Esta lista es una lista de nodos y rangos, donde los rangos de nodos se determinan separando el menor y el mayor de rango por un guión. Por ejemplo, si indicamos el parámetro `-j1,4-6,8,19-21`, inmediatamente después de lanzar el proceso, de poder migrar el proceso, este migraría a un nodo aleatorio entre los nodos: 1,4,5,6,8,19,20 y 21.

El valor de la opción de `migracion` puede ser:

**-l**: el algoritmo de equilibrado automático de carga puede forzar una migración del proceso después de haber migrado dicho proceso al nodo de destino.

**-L**: una vez migrado al nodo de destino, el proceso se quedará en él y no podrá migrar de forma automática.

**-k**: el proceso heredará la propiedad de migrabilidad de su padre.

El valor de `tipo` es un dato muy importante que sirve de ayuda al algoritmo de migración automática de carga, y este puede ser:

**-c**: en un nodo de memoria infinita, el proceso tiene como cuello de botella la velocidad del procesador.

**-i**: en un nodo de memoria infinita, el proceso tiene como cuello de botella el acceso a disco.

Además de los modificadores anteriormente citados, con `mosrun` también podemos informar a openMosix sobre la forma en que openMosix debe mantener las estadísticas de uso de los recursos del sistema del proceso, datos fundamentales para que el algoritmo de equilibrado automático de carga tome decisiones correctas. Estos modificadores son:

**-f:** mantiene las estadísticas de uso de los recursos del sistema del proceso durante poco tiempo. Esto hace que las *predicciones* de openMosix sobre el comportamiento de un proceso sean mejores ante procesos que tienen durante su evolución comportamientos similares durante largos periodos de tiempo.

**-s:** mantiene las estadísticas de uso de los recursos del sistema del proceso a largo plazo. Esto hace que las *predicciones* de openMosix sobre el comportamiento de un proceso sean mejores ante procesos que cambian constantemente de comportamiento.

**-n:** mantiene las estadísticas de uso de los recursos del sistema del proceso desde el principio del programa hasta su finalización. Esto hace que las *predicciones* de openMosix sobre el comportamiento de un proceso sean mejores en procesos que están constantemente cambiando su comportamiento, y no podemos confiar en lo que hacían hace poco.

Hay también un conjunto de shell scripts que ayudan a no enfrentarse contra las complejidades de `mosrun` al lanzar una tarea en el uso diario del cluster, y que nos permiten realizar las tareas más frecuentes de `mosrun` de forma cómoda. Estas utilidades tienen siempre la misma sintaxis, que es:

```
utilidad proceso argumentos
```

Donde `utilidad` es el nombre del shell script, `proceso` el proceso que vamos a lanzar, y `argumentos` los argumentos del proceso que lanzaremos. Las utilidades que disponemos son:

**cpujob**: ejecuta un proceso, indicando a openMosix que si la memoria del nodo fuera infinita su cuello de botella sería el procesador.

**iojob**: ejecuta un proceso, indicando a openMosix que si la memoria del nodo fuera infinita su cuello de botella será el acceso a disco.

**nomig**: ejecuta un comando en el nodo local de forma que este no podrá migrar de forma automática.

**nunhome**: ejecuta un comando de forma que preferencialmente no migrará.

**omrunon**: ejecuta un proceso, e inmediatamente después lo migra, si es posible, al nodo especificado. La sintaxis de llamada es la de lanzar un proceso directamente desde línea de comando. Útil para lanzar un proceso desde línea de comandos recomendando un nodo de ejecución.

**omsh**: ejecuta un proceso, e inmediatamente después lo migra, si es posible, al nodo especificado. La sintaxis

de llamada es la de `sh` como cuando lanzamos el proceso con `sh -c`, lo que lo hace especialmente útil para sustituir a sh en shell scripts.

**fastdecay**: ejecuta un proceso, indicando a openMosix que mantenga las estadísticas de uso de los recursos del sistema del proceso durante poco tiempo. Esto hace que las `predicciones` de openMosix sobre el comportamiento de un proceso sean mejores ante procesos que tienen durante su evolución comportamientos similares durante largos periodos de tiempo.

**slowdecay**: ejecuta un proceso, indicando a openMosix que mantenga las estadísticas de uso de los recursos del sistema del proceso a largo plazo. Esto hace que las `predicciones` de openMosix sobre el comportamiento de un proceso sean mejores ante procesos que cambian constantemente de comportamiento.

**nodecay**: ejecuta un proceso, indicando a openMosix que mantenga las estadísticas de uso de los recursos del sistema del proceso desde el principio del programa hasta su finalización. Esto hace que las *predicciones* de openMosix sobre el comportamiento de un proceso sean mejores en procesos que están constantemente cambiando su comportamiento, y no podemos confiar en lo que hacían hace poco.

Como sucedía en el caso de `mosrun`, si lanzamos un proceso con una de estas utilidades en una máquina sin soporte openMosix habilitado, o con este mal configurado, el proceso se lanzará perfectamente de forma local.

#### openMosixView

No podemos hablar de openMosix pasando por alto openMosixView, que es una cómoda y amigable aplicación de monitorización de un cluster openMosix.

openMosixView no está en las herramientas de área de usuario de openMosix por defecto. Y la razón es muy simple: las herramientas de área de usuario son lo mínimo que necesita cualquier administrador o usuario de openMosix para poder trabajar. Y en la mayor parte de las instalaciones de openMosix, la mayor parte de los nodos son *cajas* sin monitor, ratón o teclado con una instalación mínima de Linux, por lo que en principio openMosixView solo sería un problema para el administrador, que puede no tener interés en instalar las QT y KDE en una máquina que sólo va a servir procesos. A diferencia de las herramientas de área de usuario, que tienen una necesidad de bibliotecas y compiladores preinstalados mínima, openMosixView necesita muchas bibliotecas instaladas para ejecutarse y más aún para compilar, lo que hace poco práctico compilar y usar openMosixView en un nodo minimal.

Sin embargo, esto no quita que openMosixView sea una excelente herramienta de gran utilidad, y que todo administrador de openMosix podría tenerla instalada en la máquina desde la que inspecciona todo el cluster. Por ello, aunque no la haya incluido, recomiendo a todo administrador que se la descargue y la instale en los nodos que quiera utilizar como terminal gráfico del cluster.

### 5.3.5. Optimizando el cluster

#### Ayudando al algoritmo de equilibrado de carga

El primer paso que podemos dar para mejorar aún más el rendimiento es ayudar al algoritmo de equilibrado de carga proveyéndole de más información sobre las características de los nodos que forman el cluster. En el caso de los clusters heterogéneos esto es fundamental; ya que queremos que los procesos migren preferentemente a los nodos más potentes, y que la migración libere a los nodos menos potentes.

Tal y como queda el cluster cuando acabamos de instalar openMosix, el cluster ya optimiza el aprovechamiento de recursos en un cluster homogéneo -es decir, en el que todas las máquinas son iguales en potencia-. Sin embargo, el aprovechamiento de los recursos en un cluster heterogéneo aún no llega al óptimo. Para solucionar esto, podemos modificar la potencia relativa que un nodo considera que tiene. En la fecha en la que se escribe este artículo no existe una herramienta de calibración automatizada, por lo que debemos hacer esto nodo a nodo con la herramienta manual de calibración, `mosctl`, con el modificador `setspeed`.

`mosctl` con el modificador `setspeed` es una herramienta que, ejecutada sobre un nodo, permite alterar el parámetro de la potencia computacional que un nodo cree que tiene. Junto con la información de la carga, el nodo transmite también este parámetro al resto de los nodos del cluster, por lo que cada nodo debe lanzar `mosctl`

`setspeed` en algún punto de la ejecución de los scripts de inicialización si el cluster tiene máquinas distintas y queremos aprovechar el cluster al máximo.

Actualmente en openMosix empleamos como unidad una diezmilésima de la potencia de cálculo de un Pentium-III a 1.4GHz. Esto es una unidad arbitraria, ya que la potencia depende también de la velocidad de memoria, de si el bus es de 33MHz o de 66MHz, y de la tecnología de la memoria. Además, para algunas tareas un procesador de Intel es más rápido que uno de AMD, mientras que para otras el mismo procesador de AMD puede ser más rápido que el mismo procesador de Intel.

Actualmente lo mejor que podemos hacer es estimar *a ojo* cuanto puede ser el procesador de cada nodo más lento o rápido que un Pentium-III a 1.4GHz para el tipo de tareas que lanzaremos en el cluster; y después asignamos dicha potencia relativa de prueba para cada nodo, usando para ello el comando:

```
mosctl setspeed valor
```

donde `valor` es la potencia computacional del procesador así calculada.

Una vez que ya tenemos el cluster en pruebas o en producción, siempre podemos ajustar el valor para que el cluster tenga el comportamiento que queremos. En esta segunda etapa, por lo tanto, ajustamos los valores `a ojo` en valores empíricos: si notamos que un nodo suele estar demasiado cargado, le bajamos el factor de potencia de cómputo. Por otro lado, si notamos que un nodo suele estar desocupado mientras que los otros nodos trabajan demasiado, siempre podemos subir su potencia computacional estimada con este comando. Esto se puede hacer también con el cluster en producción, sin ningún problema adicional, usando el comando anteriormente citado.

Un truco habitual en openMosix es jugar con la potencia computacional estimada del nodo para mejorar la respuesta del cluster al usuario. Para ello, aumentamos un 10 % de forma artificial la potencia computacional estimada de los nodos sin monitor ni teclado, que sólo se dedican al cálculo, mientras que bajamos un 10 % la potencia computacional estimada de los nodos con monitor y teclado en los que el usuario lanza tareas. De hecho, en algunos nodos específicos que sabemos que van muy justos porque tienen ya problemas para responder a un usuario común, bajamos un 20 % la potencia computacional estimada. Con esto estamos forzando a priori algunos sentidos de migraciones.

El desperdicio de recursos del cluster será ligeramente mayor, ya que estamos dando datos erróneos de forma intencionada al algoritmo de equilibrado automático de carga. A cambio, el usuario observará una mejor respuesta al teclado y al ratón, ya que los nodos de acceso con más frecuencia tendrán un porcentaje de procesador libre para responder a una petición instantánea del usuario, sobre todo con carga media en el cluster. De cualquier forma, este truco sólo es útil cuando tenemos un cluster mixto, con nodos dedicados y no dedicados.

#### Modificando las estadísticas de carga

El subsistema de migración automática de procesos necesita información sobre como evoluciona el comportamiento del cluster. Esta información con el tiempo se tiene que descartar, ya que lo que pasaba en el cluster hace varias horas no debería afectar al comportamiento actual del cluster.

La información no se almacena de forma `eterna`. Se va acumulando de forma temporal, pero la importancia de los históricos de los sucesos va decreciendo según se van haciendo estas estadísticas más antiguas. El hecho de mantener la información de los históricos durante un tiempo permite que los picos de comportamiento anómalo en el uso del cluster no nos enmascaren el comportamiento real del cluster. Pero, por otro lado, la información debe desaparecer con el tiempo, para evitar que tengamos en cuenta sucesos que ocurrieron hace mucho tiempo, pero que ahora no tienen importancia.

Hemos aprendimos como ajustar la estadística de los procesos, proceso por proceso. Sin embargo, ahora estamos hablando de un concepto distinto: ahora hablamos de la estadística de uso de los recursos de un nodo, y no de la estadística de un proceso en particular. Del mismo modo que en el artículo anterior aprendimos a ajustar el tiempo que se mantenían las estadísticas de un proceso, ahora veremos como se ajusta el parámetro para un nodo en concreto.

El comando que usamos para determinar el tiempo que se mantienen las estadísticas de un nodo es `mosctl`, con el modificador `setdecay`.

El uso de los recursos de un nodo en un cluster openMosix es una medida compuesta del uso de los recursos por los procesos que tienen un ritmo de decaída de la estadísticas lento, y el uso de los recursos por los procesos

que tienen un ritmo de decaídas rápido. La sintaxis de la instrucción que determina el cálculo de la medida será:

```
mosctl setdecay intervalo procentajelento porcentajerápido
```

Donde `intervalo` será el intervalo en segundos entre recálculos de las estadísticas, `porcentajelento` el tanto por mil de uso que se almacena originado por procesos con decaída lenta de estadísticas, y `porcentajerápido` el tanto por mil que se almacena de procesos con decaída rápida de estadísticas.

Como lo hemos explicado aquíqueda un poco difícil de entender, asíque lo veremos con un ejemplo:

```
mosctl setdecay 60 900 205
```

Esto hace que las estadísticas históricas por nodo se recalculen cada 60 segundos. Se recalculan utilizando para acumular resultados como factor de ponderación un 90 % para la carga de los procesos de decaída lenta de antes de los últimos 60 segundos, un 20,5 % para la carga de los procesos de decaída rápida de antes de los últimos 60 segundos, y la carga del sistema de los últimos 60 segundos sin ponderación.

Esto nos permite hacer que las estadísticas por nodo -no por proceso- evolucionen a más velocidad -lo que mejora el aprovechamiento cuando la carga del cluster más frecuente son procesos largos y de naturaleza constante-, o que sean las estadísticas más constantes en el tiempo -lo que mejora el aprovechamiento en clusters donde hay muchos procesos muy pequeños ejecutándose de forma aleatoria-.

Podemos también obtener la información de los parámetros de permanencia de históricos en un cluster openMosix para un nodo particular con el comando `mosctl` y el modificador `getdecay`. La sintaxis del comando es:

```
mosctl getdecay
```

### Programando openMosix

Para programar openMosix a bajo nivel -es decir, tomando el control del cluster y de la migración- podemos emplear tres mecanismos:

Hacer uso de las herramientas de área de usuario. Este es el mecanismo recomendado para scripts en Perl, o para usar en shell-scripts. Hacer uso del sistema de ficheros */proc*. En Linux tenemos un sistema de ficheros virtual en */proc*. Este sistema de ficheros no existe físicamente en el disco, y no ocupa espacio; pero los archivos y los directorios que en él nos encontramos nos modelan distintos aspectos del sistema, tales como la memoria virtual de cada proceso, la red, las interrupciones o la memoria del sistema, entre otras cosas. openMosix no puede ser menos, y también podemos obtener información sobre su comportamiento y darle órdenes a través de */proc*. Este método de operación es ideal en cualquier lenguaje que no tiene un método cómodo para llamar a procesos externos, pero nos permite acceder con facilidad a archivos, leyendo y escribiendo su contenido. Este es el caso de la práctica totalidad de los lenguajes de programación compilados. Encontramos en los cuadros adjuntos algunos de los ficheros más importantes del */proc* de openMosix que podemos tener interés en leer o escribir. Hacer uso de la biblioteca de openMosix. El área de usuario de openMosix incluye una biblioteca en C que puede ser utilizada para hacer todo aquello que pueden hacer las herramientas de área de usuario. Este mecanismo sólo funciona en C, pero es el más cómodo para los programadores en este lenguaje. No hablaremos más de él, aunque tenemos documentación disponible en el proyecto openMosix sobre como funciona. Las bibliotecas están obsoletas, y estoy trabajando en unas bibliotecas de área de usuario nuevas.

### Planes futuros de openMosix

En la fecha de la escritura de este artículo, el equipo de openMosix está trabajando en varias características nuevas de gran interés para la comunidad.

El equipo de área de kernel está trabajando en el porting de openMosix a IA64. Este porting está siendo subvencionado por HP, y supondrá el salto a los 64 bits de openMosix. Es un proyecto pensado para que openMosix sea un elemento clave en el campo de la industria.

Werner Almesberger ha desarrollado un parche para hacer los sockets migrables. El parche aún no está muy probado, aunque hablaremos de los sockets migrables bajo openMosix en el momento que este parche sea estable.

Por otro lado, David Santo Orcero está trabajando en una nueva biblioteca de área de usuario con soporte a semántica de grid. Tiene compatibilidad inversa con la biblioteca actual, por lo que tendremos SSI transparente

también con la nueva biblioteca; pero para usar las nuevas características de grid sería necesario usar el nuevo API.

#### Ficheros en */proc/hpc*

Los ficheros y directorios y directorios más importantes que encontramos en */proc/hpc* son:
*/proc/hpc/admin*: contiene algunos ficheros importantes de administración.
*/proc/hpc/config*: configuración del cluster openMosix.
*/proc/hpc/gateways*: número de saltos máximo entre redes para un paquete de openMosix.
*/proc/hpc/nodes*: directorio que contiene un subdirectorio por cada nodo del cluster, y que permite administrar el cluster desde cada nodo.

#### Ficheros en */proc/hpc/admin*

Los ficheros más importantes que tenemos en el directorio */proc/hpc/admin* son:
*/proc/hpc/admin/bring*: si escribimos en este fichero un 1 desde un nodo, los procesos lanzados desde dicho nodo volverán al nodo raíz. */proc/hpc/admin/block*: si escribimos en este fichero un 1 desde un nodo, bloqueamos la llegada al nodo de cualquier proceso que se haya generado en un nodo externo.

#### Ficheros de configuración e información de cada proceso

En Linux, cada proceso tiene un subdirectorio en */proc*, cuyo nombre es el PID del proceso, que contiene importante información del proceso. Si tenemos openMosix activado, encontraremos algunos ficheros adicionales para cada proceso en su directorio. De entre estos ficheros adicionales, destacamos:
*/proc/PID/cantmove*: si el fichero no puede migrar fuera del nodo, encontramos en este archivo la razón por la que el proceso no puede emigrar en el momento en el que se consulta este archivo. Esta condición puede cambiar si el proceso cambia su comportamiento, o entra/sale de modo virtual 8086.
*/proc/PID/nmigs*: número de veces que un proceso ha migrado. Si este número es demasiado alto, probablemente tenemos mal calibrado el cluster, y tenemos que aumentar el coste de migración.
*/proc/PID/goto*: si escribimos un número en este fichero, openMosix intentará que el proceso al que fichero `goto` le corresponda migre al nodo cuyo número de nodo sea el que hemos grabado en el fichero. Puede no migrar a donde hemos pedido que migre: la migración en este caso nunca es obligatoria, y el sistema, aunque hará lo que pueda por migrarlo, puede no hacerlo.
*/proc/PID/where*: dónde un proceso se está ejecutando en la actualidad. Observamos que mientras que */proc/PID/goto* es de escritura, */proc/PID/where* es de lectura.

#### Ficheros de configuración e información de cada nodo

Al igual que tenemos un directorio por proceso desde el que podemos manipular cada proceso, tenemos un directorio por nodo desde el que podemos manipular cada nodo. Los directorios están en `/proc/hpc/nodes`, y tienen como nombre de directorio el número de nodo openMosix.
Los ficheros más importantes que encontramos dentro del directorio de cada nodo son:
*/proc/hpc/nodes/identificadornodo/CPUs*: El número de procesadores que tiene el nodo cuyo identificador es `identificadornodo`, si es un nodo con más de un procesador y además tiene el soporte SMP activado en el kernel. Dos veces el número de procesadores que tiene el nodo, si es un nodo con procesadores Pentium IV e hyperthreading activado. 1 en otro caso no listado anteriormente.
*/proc/hpc/nodes/identificadornodo/load*: Carga asociada al nodo cuyo identificador es `identificadornodo`.
*/proc/hpc/nodes/identificadornodo/mem*: memoria disponible lógica para openMosix en el nodo cuyo identificador es `identificadornodo`.
*/proc/hpc/nodes/identificadornodo/rmem*: memoria disponible física para openMosix en el nodo cuyo identificador es `identificadornodo`.
*/proc/hpc/nodes/identificadornodo/tmem*: memoria total de la máquina, libre o no, en el nodo cuyo identificador es `identificadornodo`.
`/proc/hpc/nodes/identificadornodo/speed`: potencia del nodo cuyo identificador es `identificadornodo`. Recordemos que es el valor que hemos dicho al nodo que tiene. No es un valor calculado, se lo damos nosotros como administradores del cluster. Es el valor de velocidad del que hablamos en este artículo.

*/proc/hpc/nodes/identificadornodo/status*: estado del nodo cuyo identificador es `identificadornodo`. Estudiamos los estados de un nodo con detalle en el artículo anterior.

*/proc/hpc/nodes/identiciadornodo/util*: coeficiente de utilización del nodo cuyo identificador es `identificadornodo`. Estudiamos el coeficiente de utilización de un nodo en el artículo anterior.

## 5.4. ADMINISTRACIÓN DEL CLUSTER

| | |
|---|---|
| echo 1 > /proc/hpc/admin/block | bloquea la llegada de procesos remotos |
| echo 1 > /proc/hpc/admin/bring | lleva todos los procesos a su nodo raíz |

Cuadro 5.1: Administración: Cambiando los parámetros en /proc/hpc

| | |
|---|---|
| config | el fichero de configuración principal (escrito por la utilidad *setpe*) |
| block | permite/prohíbe la llegada de procesos remotos |
| bring | lleva todos los procesos a su nodo raíz |
| dfsalinks | lista los actuales enlaces DFSA |
| expel | envía los procesos huésped a su nodo raíz |
| gateways | numero máximo de gateways |
| lstay | los procesos locales se suspenderan |
| mospe | contiene el ID de nuestro nodo openMosix |
| nomfs | activa/desactiva MFS |
| overheads | para ajustes |
| quiet | detiene la obtención de información sobre la carga del sistema |
| decay-interval | intérvalo para recoger información sobre la carga |
| slow-decay | por defecto 975 |
| fast-decay | por defecto 926 |
| speed | velocidad relativa a un PIII/1GHz |
| stay | activa/desactiva el proceso de migrado automático |

Cuadro 5.2: Administración: Binarios en /proc/hpc/admin

El mantenimiento de un sistema resulta incluso más delicado y costoso (en tiempo) que su correcta instalación. En este capítulo se tomará contacto con todas las herramientas con las que cuenta openMosix para poder gestionar tu sistema.

La recomendación que lanzamos desde este manual es que pruebes con ellas para conocer exactamente la respuesta de tu cluster, ya que más tarde puede facilitarte la detección de errores o configuraciones poco adecuadas.

### 5.4.1. Administración básica

openMosix proporciona como principal ventaja la migración de procesos hacia aplicaciones HPC. El adminsitrador puede configurar el cluster utilizando las herramientas de área de usuario de openMosix (openMosix-user-space-tools[6]) o editando la interfície que encontraremos en */proc/hpc* y que será descrita con más detalle seguidamente.

### 5.4.2. Configuración

Los valores en los ficheros del directorio */proc/hpc/admin* presentan la configuración actual del cluster. El administrador del mismo puede configurar estos valores para cambiar la configuración en tiempo de ejecución, tal como se muestra en las tablas.

[6] `http://www.orcero.org/openmosix`

| | |
|---|---|
| clear | resetea las estadísticas |
| cpujob | informa a openMosix que el proceso está ligado al procesador |
| iojob | informa a openMosix que el proceso está ligado a la E/S |
| slow | informa a openMosix que actualice las estdísticas más lentamente |
| fast | informa a openMosix que actualice las estdísticas más rápidamente |

Cuadro 5.3: Administración: Escribiendo un '1' en /proc/hpc/decay

| | |
|---|---|
| /proc/[PID]/cantmove | razón por la cual el proceso ha sido migrado |
| /proc/[PID]/goto | a qué nodo el proceso podrá migrar |
| /proc/[PID]/lock | si el proceso se ve bloquead en su nodo raíz |
| /proc/[PID]/nmigs | el numero de veces que el proceso ha migrado |
| /proc/[PID]/where | donde el proceso se encuentra siendo computado actualmente |
| /proc/[PID]/migrate | same as goto remote processes |
| /proc/hpc/remote/from | el nodo raíz del proceso |
| /proc/hpc/remote/identity | información adicional del proceso |
| /proc/hpc/remote/statm | estadística de memoria del proceso |
| /proc/hpc/remote/stats | estadística del procesador del proceso |

Cuadro 5.4: Administración: Información adicional sobre los procesos locales

| | |
|---|---|
| /proc/hpc/nodes/[openMosix_ID]/CPUs | el número de CPUs que posee el nodo |
| /proc/hpc/nodes/[openMosix_ID]/load | la carga de openMosix en este nodo |
| /proc/hpc/nodes/[openMosix_ID]/mem | memoria disponible para openMosix |
| /proc/hpc/nodes/[openMosix_ID]/rmem | memoria disponible para Linux |
| /proc/hpc/nodes/[openMosix_ID]/speed | velocidad del nodo relativa a un PIII/1GHz |
| /proc/hpc/nodes/[openMosix_ID]/status | estado del nodo |
| /proc/hpc/nodes/[openMosix_ID]/tmem | memoria disponible |
| /proc/hpc/nodes/[openMosix_ID]/util | utilización del nodo |

Cuadro 5.5: Administración: Información de los otros nodos

### 5.4.3. Las herramientas de área de usuario

Estas herramientas permitirán un fácil manejo del cluster openMosix. Seguidamente se enumeran con todos sus parámetros.

`migrate [PID] [openMosix_ID]` envia una petición de migrado del proceso identificado con el ID, al nodo que indiquemos..

`mon` es un monitor de los *daemons* basado en el terminal y da información relevante sobre el estado actual que puede ser visualizada en diagramas de barras.

`mosctl` es la principal utilidad para la configuración de openMosix. Su sintaxis es:

```
mosctl  [stay|nostay]
       [block|noblock]
       [quiet|noquiet]
       [nomfs|mfs]
       [expel|bring]
       [gettune|getyard|getdecay]

        mosct whois [openMosix_ID|IP-address|hostname]
        mosct [getload|getspeed|status|isup|getmem|getfree|getutil] [openMosix_ID]
        mosctl setyard [Processor-Type|openMosix_ID||this]
        mosctlsetspeed interger-value
        mosctlsetdecay interval [slow fast]
```

| | |
|---|---|
| stay | desactiva la migración automática |
| nostay | migración automática (defecto) |
| lstay | local processes should stay |
| nolstay | los procesos locales podran migrar |
| block | bloquea la llegada de otros procesos |
| noblock | permite la llegada de procesos |
| quiet | desactiva la posibildiad de dar información sobre la carga del nodo |
| noquiet | activa la posibildiad de dar información sobre la carga del nodo |
| nomfs | desactiva MFS |
| mfs | activa MFS |
| expel | envía fuera del nodo los procesos que han llegado previamente |
| bring | traerá todos los procesos migrados hacia su nodo raíz |
| gettune | muestra el parámetro de overhead |
| getyard | muestra la utilización actual de Yardstick |
| getdecay | muestra el estado del parámetro decay |
| whois | nos muestra el openMosix-ID, la dirección IP y los nombres de host del cluster |
| getload | muestra la carga (openMosix-) |
| getspeed | muestra la velocidad (openMosix-) |
| status | muestra el estado y la configuración actual |
| isup | nos informa de si un nodo está funcionando o no (ping openMosix) |
| getmem | muestra la memoria lógica libre |
| getfree | muestra la memoria física libre |
| getutil | muestra la utilización del nodo |
| setyard | establece un nuevo valor para Yardstick |
| setspeed | establece un nuevo valor para la velocidad (openMosix-) |
| setdecay | establece un nuevo valor para el intervalo del decay |

Cuadro 5.6: Administración: Parámetros de `mosctl` con más detalle

Con `mosrun` ejecutaremos un comando especialemte configurado en un nodo establecido.
Su sintaxis: `mosrun [-h|openMosix_ID| list_of_openMosix_IDs] command [arguments]`

El comando `mosrun` puede ser ejecutado con diversas opciones. Para evitar complicaciones innecesarias viene con ciertas pre-configuraciones para ejecutar las tareas con configuraciones especiales de openMosix.

| | |
|---|---|
| nomig | runs a command which process(es) won't migrate |
| runhome | ejecuta un comando bloqueado en el nodo raíz |
| runon | ejecutará un comando el cuál será directamente migrado y bloqueado a cierto nodo |
| cpujob | informa a openMosix que el proceso está ligado a la CPU |
| iojob | informa a openMosix que el proceso está ligado a la E/S |
| nodecay | ejecuta un comando e informa al cluster de no refrescar las estadisticas de carga |
| slowdecay | ejecuta un comando con intervalo de decay grande para acumular en las estadísticas |
| fastdecay | ejecuta un comando con intervalo de decay pequeño para acumular en las estadísticas |

Cuadro 5.7: Administración: Parámetros adicionales para `mosrun`

`setpe` es una utilidad de configuración manual del nodo *sintaxis:*

```
setpe   -w -f   [hpc_map]
setpe   -r [-f  [hpc_map]]
setpe   -off
```

`-w` lee la configuración de openMosix desde un fichero (normalmente /etc/hpc.map).
`-r` escribe la configuración actual de openMosix en un fichero (normalmente /etc/hpc.map).
`-off` desactiva la configuración actual del cluster.

`tune` es una utilidad de calibración y optimización de openMosix (para más información recurra a las páginas *man* de tune).

Existen utilidades adicionales a la interfície de */proc* y a las línias de comandos. Por ejemplo existen unos parches para `ps` y para `top` (llamados `mps` y `mtop`) los cuales muestran adicionalmente el ID de nuestro nodo openMosix en una columna. Esta posibilidad es interesante para encontrar dónde ha migrado un cierto proceso.

Para clusters pequeños pueden sernos muy útiles las utilidades de openMosixView, una GUI para las tareas de administración más comunes y que más adelante se detalla en un capítulo.

### 5.4.4. Detección automática de nodos

El demonio de auto-detección de nodos, `omdiscd`, proporciona un camino automático para la configuración de nuestro cluster openMosix. Con él podremos eliminar la necesidad de configuraciones manuales como son la edición del fichero */etc/mosix.map* .

`omdiscd` genera un envío de paquetes *multicast* (a todas las direcciones, en nuestro caso, nodos) para notificar a los otros nodos que hemos añadido uno nuevo. Esto significa que al añadir un nodo sólo tendremos que iniciar `omdiscd` en él.

Debemos ocuparnos de algunos requisitos previos como pueden ser una buena configuración de la red de interconexión de los nodos, principalmente para el correcto enrutamiento de paquetes. Sin una ruta por defecto deberemos especificar a `omdiscd` la interfície con la opción `-i`. De otra forma acabará con un error parecido al siguiente:

```
Aug 31 20:41:49 localhost omdiscd[1290]: Unable to determine address of
default interface.  This may happen because there is no default route
configured.  Without a default route, an interface must be: Network is
unreachable
Aug 31 20:41:49 localhost omdiscd[1290]: Unable to initialize network.
Exiting.
```

Un ejemplo de buena configuración podría ser la siguiente:

```
[root@localhost log]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
0.0.0.0         10.0.0.99       0.0.0.0         UG    0      0        0 eth0
```

La importancia de poder automatizar el reconocimiento de nuevos nodos conectados al sistema ha facilitado que se llegue a la simplicidad con la que contamos actualmente para iniciar dicha detección, con el comando `omdiscd`.

Ahora echando un vistazo a los *logfiles* tendríamos que ver algo parecido a

```
Sep  2 10:00:49 oscar0 kernel: openMosix configuration changed: This is openMosix
#2780 of 6 configured)
Sep  2 10:00:49 oscar0 kernel: openMosix #2780 is at IP address 192.168.10.220
Sep  2 10:00:49 oscar0 kernel: openMosix #2638 is at IP address 192.168.10.78
Sep  2 10:00:49 oscar0 kernel: openMosix #2646 is at IP address 192.168.10.86
Sep  2 10:00:49 oscar0 kernel: openMosix #2627 is at IP address 192.168.10.67
Sep  2 10:00:49 oscar0 kernel: openMosix #2634 is at IP address 192.168.10.74
```

Tendremos el cluster listo para ser utilizado.

`omdiscd` tiene otras opciones entre las que cuentan poder ejecutarse como un demonio (por defecto) o en *background* (segundo plano) donde la salida será la pantalla (la salida estándar), con el comando `omdiscd -n`. La interfície, como ya se ha indicado, debe ser especificada con la opción `-i`.

Ahora vamos a ver brevemente la herramienta `showmap`. Esta utilidad nos mostrará el nuevo mapa.

```
[root@oscar0 root]# showmap
My Node-Id: 0x0adc
```

```
Base Node-Id Address          Count
------------ ---------------- -----
0x0adc       192.168.10.220   1
0x0a4e       192.168.10.78    1
0x0a56       192.168.10.86    1
0x0a43       192.168.10.67    1
0x0a4a       192.168.10.74    1
```

Existen otras muchas utilidades que pueden sernos útiles para la detección automática de nodos, como un mecanismo de routing para clusters con más de una red de conexión. Toda esta información es actualizada constantemente y podremos encontrarla en los ficheros README y DESIGN de las herramientas de usuario.

### Compilaciones

Si queremos obtener este módulo a partir de las fuentes tendremos que hacer una pequeña modificación en el fichero *openmosix.c*. Una de las línias, concretamente

```
#define ALPHA
```

tendremos que comentarla ya que nosotros estamos trabajando en plataforma x86.

Si quisiéramos tener un historial más detallado podemos editar *main.c* para escribir
`log_set_debug(DEBUG_TRACE_ALL);` (en la línia 84 aproximadamente)
ahora podremos ejecutar

```
make clean
make
```

### Problemas

Algunas veces la auto-detección no funcionará tal como podríamos esperar, per ejemplo cuando un nodo debería ser detectado y no ve el tráfico multicast que se lleva a cabo en la red.

Esto ocurre con algunas targetas PCMCIA. Una solución posible sería poner la interfície en *modo promíscuo*, tal como se detalla seguidamente:

```
Aug 31 20:45:58 localhost kernel: openMosix configuration changed:
        This is openMosix #98 (of 1 configured)
Aug 31 20:45:58 localhost kernel: openMosix #98 is at IP address 10.0.0.98
Aug 31 20:45:58 localhost omdiscd[1627]: Notified kernel to activate
openMosix  Aug 31 20:45:58 localhost kernel:
        Received an unauthorized information request from 10.0.0.99
```

Algo que podríamos probar es forzar manualmente nuestro NIC a modo promíscuo y/o multicast, así:

```
ifconfig ethx promisc
```

o

```
ifconfig ethx multicast
```

Podremos ejecutar igualmente

```
tcpdump -i eth0 ether multicast
```

Si se nos muestra **simulated** es que seguramente hemos olvidado poner el comentario a la línia `#define ALPHA`, sería:

```
Aug 31 22:14:43 inspon omdiscd[1422]: Simulated notification to activate openMosix
[root@inspon root]# showmap
My Node-Id: 0x0063

Base Node-Id Address          Count
------------ ---------------- -----
```

```
0x0063       10.0.0.99        1
[root@inspon root]# /etc/init.d/openmosix status
OpenMosix is currently disabled
[root@inspon root]#
```

## 5.5. AJUSTES EN EL CLUSTER

*An expert is a man who has made all the mistakes*
*which can be made in a very narrow field.*

*Niels Bohr*

### 5.5.1. Testeo de rendimiento con *Stress-Test*

**Descripción general**

Este stress-test está hecho para evaluar un cluster openMosix. Realizará muchísimas evaluaciones a sus aplicaciones y a su kernel, para testear la estabilidad y otras cuestiones relacionadas con openMosix (por ejemplo migración de procesos y mfs). Durante este test el cluster se verá sobrecargado, es por eso que debería detener cualquier otra aplicación que tenga corriendo antes de iniciarlo. Al finalizar se generará un reporte detallado acerca de cada componente que ha sido evaluado.

**Descripción detallada**

Stress-Test corre una cantidad importante de programas para evaluar la funcionalidad de todo el sistema. A continuación encontrará una descripción de cada test.

**# distkeygen** Esta aplicación es usada para generar 4000 pares de llaves RSA con una longitud de 1024 bits. Se distribuye en tantos procesos como procesadores haya en su cluster Openmosix vía fork.

*Requerimientos:* Compilador gcc y la librería OpenSSL.
Copyright (C) 2001 Ying-Hung Chen (GPL)
`http://www.yingternet.com/mosix`http://www.yingternet.com/mosix

**#portfolio** `Portfolio` es un programa realizado en lenguaje Perl que simula distintos portfolios de acciones para un determinado período de tiempo. Está basado en el libro *The intelligent asset Allocator* de William Bernstein.
Este programa está realizado bajo licencia GPL.
Autor: Charles E. Nadeau Ph.D.,(C) 2002 - `charlesnadeau at hotmail dot com`

**#eatmen** Simplemente calcula funciones senoidales y raíces cuadradas para un valor determinado, lo hace un millón de veces, mientras escribe a un archivo el valor del contador del bucle (archivo que aumenta su tamao enormemente). Este test es iniciado automáticamente tantas veces (en forma simultánea) según la cantidad de procesadores que haya en su cluster openMosix.

**#forkit** Este test es similar al anterior pero en cambio usa la llamada a sistema *fork()* para crear múltiples procesos (tres veces el número de procesadores de su cluster. No escribe la salida a ningún archivo como lo hace `eatmen`.
**#mfstes** Este programa crea un archivo de 10MB y lo copia hacia y desde todos los nodos. Es para chequear al oMFS.

**#test kernel syscall** El *Linux™ Test Project* es un proyecto que nace de la unión de SGI™, IBM®, OSFL™, y Bull® con el objetivo de dar a la comunidad open source programas de testeo (test suites) para validar la confiabilidad, robustez y estabilidad de Linux. El *Linux Test Project* es una colección de herramientas para evaluar el kernel. El objetivo es mejorar el kernel. Los Interesados en contribuir son invitados a unirse a este proyecto.
Más información en `http://ltp.sf.net` ó en `http://ltp.sourceforge.net`

**#moving** El archivo `moving.sh` moverá a `start_openMosix_test.sh` a cada nodo en su cluster openMosix mientras este corriendo el test. Entonces `'start_openMosix_test.sh'` migrará cada minuto hacia otro nodo durante la ejecución del mismo. Dependiendo de la duración del test en su cluster migrará de 20 a 40 veces.

**Instalación**

- Desde las fuentes
  Ubicándose por ejemplo en `/usr/local:`

  ```
  gunzip ontest.tar.gz
  ```

  ```
  tar -xvf omtest.gz
  ```

  Después `cd /usr/local/omtest` y ejecute:

  ```
  ./compile_tests.sh
  ```

  Esto instalará los módulos y compilará los archivos necesarios. Necesitará privilegios de administrador para ello (rot). Pudiendo luego correr el openMosix stress-test como simple usuario (quizás deba ahora borrar los archivos temporales de la ejecución como administrador de `/tmp` porque no tendrá permiso de sobrescribirlos luego como simple usuario. Puede ejecutar el test con el siguiente comando:

  ```
  ./start_openMosix_test.sh
  ```

- Usando un paquete RPM
  Instálelo con el siguiente comando:

  ```
  rpm -ihv omtest.rpm
  ```

  Ahora puede iniciar el openMosix Stress-test con el siguiente comando:

  ```
  start_openMosix_test.sh
  ```

  (el paquete RPM será instalado en `/usr/local/omtest`)

**Download**[7]

**Version 0.1-4 del openMosix stress-test**
omtest-0.1-4.tar.gz (sources-package)
omtest-0.1-4.i386.rpm (RPM-package)
Cambios:
-se incluyó un archivo con la versión version.txt
-se actualizó ltp test-package
-se agregó lmbench al stress-test.
(debe ser ejecutado manualmente por `run_lmbench.sh`)

**Version 0.1-3 del openMosix stress-test**
omtest-0.1-3.tar.gz (sources-package)
omtest-0.1-3.i386.rpm (RPM-package)
Cambios:
-stderr ahora también reporta hacia stdout después de cada test.
-se corrigió un pequeño bug en kernel-syscall start-script (directorio tmp).
-se corrigió un mensaje de error que se producíia durante el borrado de los archivos temporales (distkeygen test).
-Usted puede ahora correr también el stress-test para openMosix como usuario común.
(solamente la instalación requiere privilegios de administrador)

**Version 0.1-2 del openMosix stress-test**
omtest-0.1-2.tar.gz (sources-package)
omtest-0.1-2.i386.rpm (RPM-package)
Cambios:
-stderr es copiado al informe generado por el test
-se agregó un chequeo para nodos que son parte del cluster pero que estan sin funcionar.

[7] `http://www.openmosixview.com/omtest/#down`

**Version 0.1-1 of the openMosix stress-test**
omtest-0.1-1.tar.gz (sources-package)
omtest-0.1-1.i386.rpm (RPM-package)

**Informe de Ejemplo**[8]

Este es un ejemplo del informe generado por este test en un cluster openMosix versión 2.4.18-1 de 5 nodos
openMosix-stress-test-report.txt (ejemplo)

**Descargo de responsabilidad** Todos los fuentes de los programas son entregados sin garantíias. Use el openMosix stress-test a su propio riesgo y siéntase libre de contribuir con sus propias ideas. El autor de este test para cluster no es responsable de ningún error y sus consecuencias mientras esté corriendo el stress-test en su sistema. Asegúrese de hacer una copia de respaldo antes de empezar con este test, debido a que quizás pueda sobrecargar y hasta *colgar* su sistema. Matt Rechenburg - `mosixview@t-online.de`

[8] `http://www.openmosixview.com/omtest/openMosix-stress-test-report.txt`

## 5.6. OPENMOSIXVIEW

*...when men were men and wrote their own device drivers...*
*Linus Torvalds*

El entorno openMosixview es la siguiente versión (y totalmente reescrita) de Mosixiew.

Es una interfície gráfica (GUI) libre para la adminstración y mantenimiento de un cluster openMosix que podemos bajarnos de la web del proyecto[9].

La *suite* openMosixview contiene 7 aplicaciones altamente útiles y eficaces tanto para la administración como para la monitorización del cluster.

- **openMosixview** principal aplicación de monitorización y administración.
- **openMosixprocs** aplicación para la administración de procesos.
- **openMosixcollector** captura la información del cluster proporcionada por los demonios.
- **openMosixanalyzer** analizador de la información capturada por openMosixcollector.
- **openMosixhistory** historial de monitorización de procesos del cluster.
- **openMosixmigmon** visor que representa la migración de procesos.
- **3dmosmon** visor para monitorización de datos en 3D.

Todos los componentes son accesibles desde la ventana de la aplicación principal. Este entorno facilita la interacción con el usuario puesto que le permite ejecutar los comandos de consola más comunes con unos pocos clicks de ratón.

### 5.6.1. Instalación

Requerimientos:

- tener instaladas las librerías QT según marque la versión,
- derechos de superusuario -*root*-,
- las herramientas de usuario de openMosix -*userland-tools*-.

#### Instalación desde paquetes RPM

Los paquetes RPM tienen como directorio de instalación la ruta */usr/local/openMosixview-<versión>*. Tras bajar la última versión de los paquetes RPM de openMosixview, habrá que instalarlos como cualquier otro paquete, sea desde l'ınea de comandos:

```
rpm -i openMosixview-<versión>-<distribución>.rpm
```

Esto nos instalará los ficheros binarios en el directorio */usr/bin/*. Para desinstalarlos ejecutaremos

```
rpm -e openMosixview
```

#### Instalación desde las fuentes

Con la versión en forma de *tarball* y lo descomprimiremos -y desempaquetaremos- así:

```
tar -zxvf openMosixview-<versión>.tar.gz
```

Script de Configuración Automático

Sólo será necesario entrar al directorio openMosixview y ejecutar:

[9] http://www.openmosixview.com

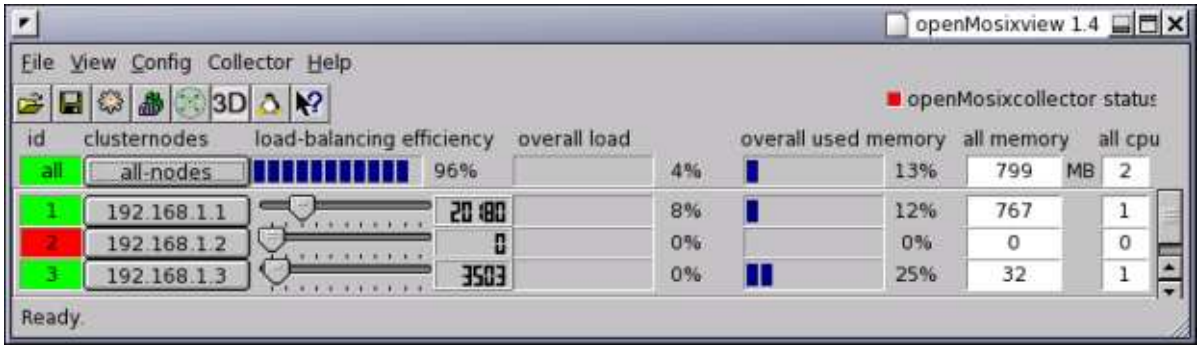



Figura 5.1: openMosixview: Aplicación principal

```
./setup [directorio_de_instalación_qt_<versión>]
```

COMPILACIÓN MANUAL

Será necesario situar la variable QTDIR hacia el directorio de la distribución QT, por ejemplo:

```
export QTDIR=/usr/lib/qt-2.3.0 (para bash)
```

ó

```
setenv QTDIR /usr/lib/qt-2.3.0 (para csh)
```

Tras lo qual tendría que ejecutar con éxito la configuración y la compilación:

```
./configure
make
```

Luego será necesario hacer lo mismo en los subdirectorios de openMosixcollector, openMosixanalyzer, openMosixhistory and openMosixviewprocs, para poder disponer de los binarios ejecutables de estas aplicaciones. Tras este procedimiento copiaremos todos los binarios a */usr/bin/* siguiendo:

```
cp openMosixview/openMosixview /usr/bin
cp openMosixviewproc/openMosixviewprocs/mosixviewprocs /usr/bin
cp openMosixcollector/openMosixcollector/openMosixcollector /usr/bin
cp openMosixanalyzer/openMosixanalyzer/openMosixanalyzer /usr/bin
cp openMosixhistory/openMosixhistory/openMosixhistory /usr/bin
```

y el script de iniciación de `openMosixcollector` en el directorio de iniciación[10]:

```
cp openMosixcollector/openMosixcollector.init /etc/init.d/openMosixcollector
```

Ahora, si queremos que estas aplicaciones de monitorización puedan ejecutarse localmente en cualquier nodo, tendrán que copiarse los binarios a cada nodo del cluster -en el directorio */usr/bin/*

## 5.6.2. Utilizando openMosixview

### Aplicación principal

La `Figura 5.1` muestra la ventana de la aplicación. El usuario podrá interaccionar con el API de openMosix a través de sus controles. Para cada nodo del cluster -cada fila-: una luz, un botón, un *speed-slider*, un número lcd, dos barras de progreso porcentual y un par de etiquetas.

[10]El directorio de destino puede variar según la distribución linux que usemos, de otra forma el directorio de iniciación puede estar en */etc/rc.d/init.d* .

Las luces de la izquierda nos muestran el ID del nodo y su estado. El fondo rojo significa que el nodo no se encuentra operativo, y verde lo contrario.

Si hacemos clic en el botón que muestra la dirección IP de un nodo habremos invocado al diálogo de configuración que nos mostrará los botones para ejecutar los comandos de `mosctl` más comunes -ver sección *Las herramientas de área de usuario* del capítulo 5.

Con los *speed-sliders* podemos establecer la velocidad que considerará el cluster para cada nodo. Este parámetro se muestra numéricamente en el *display* lcd dse su derecha. Hay que tener en cuenta que estos ajustes tienen un efecto directo sobre el algoritmo de balanceo de carga de openMosix, puesto que intervienen en la ponderación de velocidad que se debe considerar para cada nodo. De esta manera los procesos migrarán más fácilmente hacia un nodo cuya velocidad sea más elevada. Recordemos que este concepto de velocidad no tiene que ser el que realmente posea la computadora, es simplemente el parámetro que queremos que openMosix considere para cada máquina.

Las barras de progreso dan una idea general de la carga de cada nodo del cluster. La primera se refiere a la carga del procesador y muestra un porcentaje que será una aproximación del valor escrito por openMosix en el fichero */proc/hpc/nodes/x/load*. La segunda barra nos muestra la memoria utilizada en cada nodo. El box de la izquierda nos muestra el total de memoria -en megabytes- que posee cada nodo. Finalmente el último box muestra el número de procesadores de cada nodo.

Hasta ahora se ha definido la interfaz para cada nodo en particular, no obstante la misma aplicación muestra información general del cluster.

El box *load-balancing efficiency* es un indicador de la eficiencia del algoritmo de balanceo de carga. Un valor próximo al 100 % indica que la carga computacional ha podido dividirse equitativamente entre los nodos; este es precisamente el fin que persigue openMosix.

Bajo la barra de menús existen varios iconos que lanzan las demás aplicaciones del entorno openMosixview. Podemos utilizar los menús de `collector-` y `analyzer-` para administrar openMosixcollector y openMosixanalyzer, respectivamente.

Estas dos partes de las aplicaciones openMosixview son muy adecuadas para construir un historial del trabajo hecho -y la manera en como se ha hecho- en el cluster. Cuando hagamos iniciado la captura de datos el led etiquetado como `openMosixcollector status` se mostrará verde.

### La ventana de configuración

Esta ventana emergente de la `Figura 5.2` aparecerá tras hacer clic en el botón de la IP de cualquier nodo, permite una fáci configuración de la máquina con dicha dirección.

Todos los comandos que esta ventana invoca pueden ser ejecutados a través de *rsh* o *ssh* en los nodos remotos -y también en el nodo local, evidentemente- siendo *root* y sin necesidad de contraseñas.

Si hay instalado `openMosixprocs` en los nodos remotos del cluster podremos clicar en el botón *remote proc-box* para invocar openMosixprocs remotamente.

Se nos mostrará en pantalla los parámetros [xhost+hostname] y serán configurados para apuntar a nuestro nodo local. Además el cliente es ejecutado en el remoto con *rsh* o *ssh* (recordemos que tendríamos que tener copiados los binarios de openmsoxiprocs en el directorio /usr/bin de cada nodo).

openMosixprocs nos permitirá una administración de nuestros programas.

Si hemos entrado a nuestro cluster desde una estación de trabajo remota podremos introducir nuestro nombre de nodo local en la edit-box, debajo de la *remote proc-box*. Luego openMosixprocs se nos mostrará en nuestra estación de trabajo y no en el nodo del cluster donde hemos entrado (podemos configurar [xhost+clusternode] en nuestra estación de trabajo). Podemos encontrar un historial en el combo-box así como el nombre de nodo que hayamos escrito.

### Ejecución avanzada

Si queremos iniciar trabajos en nuestro cluster el diálogo de *advanced execution* mostrado en la `Figura 3` podrá ayudarnos para convertirlo a modo gráfico.

Elegiremos el programa a iniciar con el botón `run-prog` (en File-Open-Icon) y especificaremos cuál y donde se encuentra el trabajo que queremos iniciar en este diálogo de ejecución. Hay diferentes opciones que se describen en la próxima tabla.

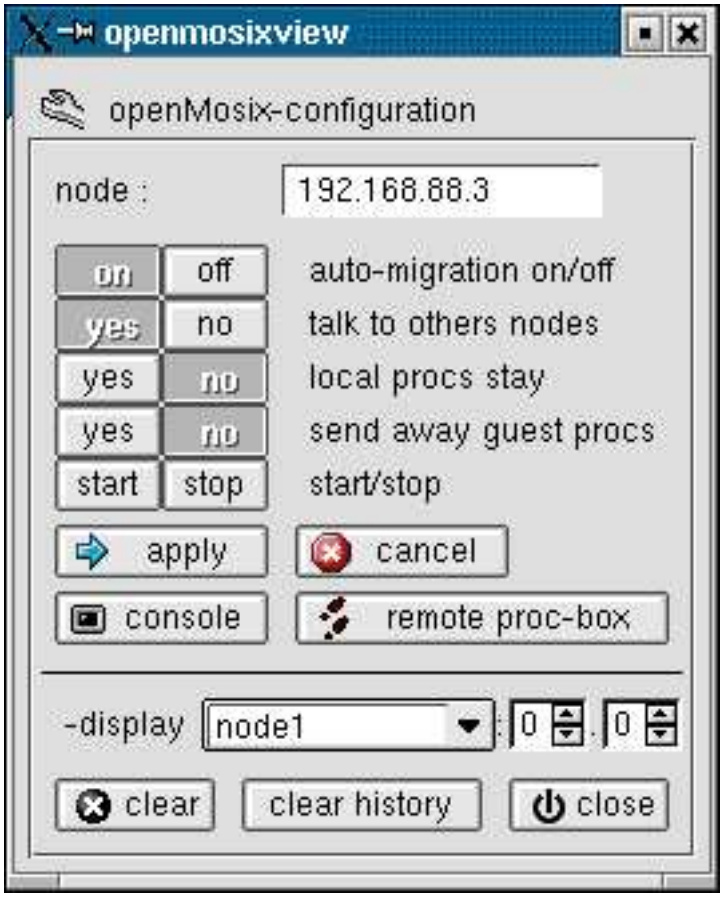



Figura 5.2: openMosixview: Propiedades de los nodos

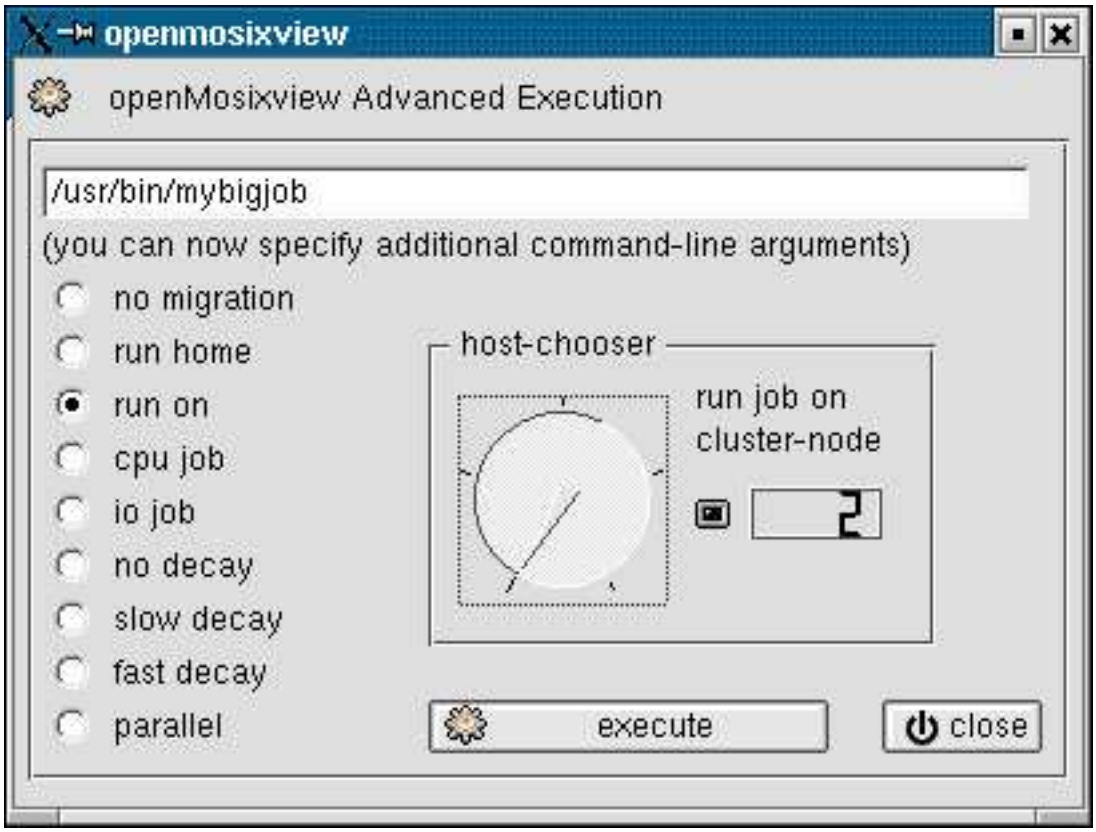



Figura 5.3: openMosixview: Ejecución avanzada

| | |
|---|---|
| no migration | iniciar un proceso local que no migrará |
| run home | iniciar un proceso local |
| run on | iniciar un trabajo en el nodo que elijamos con ”nodo-chooser” |
| cpu job | iniciar un proceso cpu-intensivo en el nodo (nodo-chooser) |
| io job | iniciar un proceso IO-intensivo en el nodo (nodo-chooser) |
| no decay | iniciar un proceso sin decay (nodo-chooser) |
| slow decay | iniciar un proceso con decay lento (nodo-chooser) |
| fast decay | iniciar un proceso con decay rápido (nodo-chooser) |
| parallel | iniciar un proceso paralelo en todos o algunos nodos (special nodo-chooser) |

Cuadro 5.8: openMosixview: Condiciones de inicio avanzadas para procesos

#### La línea de comandos

Podremos especificar los argumentos a los que antes podíamos acceder gráficamente a través de comandos en el *lineedit-widget* en la parte superior de la ventana, tal como se muestra en la `Figura 3`.

#### El host-chooser

Para todas las tareas que ejecutemos de manera remota sólo tenemos que elegir un nodo que la hospede con el dial-widget. El ID de openMosix del nodo se nos muestra también en forma de lcd. Luego pulsaremos *execute* para iniciar el trabajo.

#### El host-chooser paralelo

Podemos configurar el primer y el último nodo con 2 spinboxes. Luego el comando será ejecutado en todos los nodos, desde el primero hasta el último. Podremos también invertir esta opción.

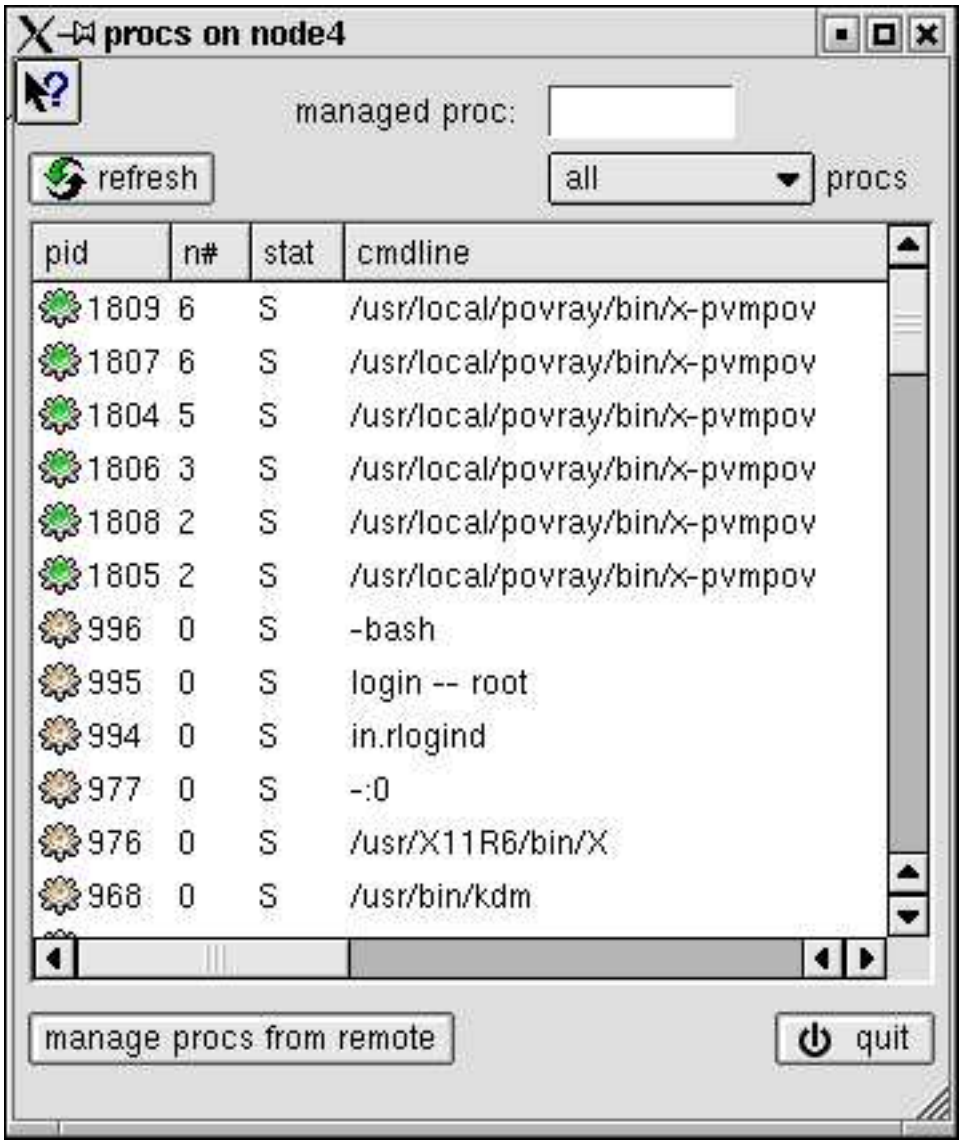



Figura 5.4: openMosixprocs: Administración de procesos

### 5.6.3. openMosixprocs

#### Introducción

La ventana de procesos mostrada en la `Figura 4` es muy útil para la administración de los procesos de nuestro cluster.

Deberemos instalar esta aplicación en cada nodo. La lista de procesos da una idea de lo que se está ejecutando y donde. La segunda columna informa del ID del nodo donde se está ejecutando la tarea. Con un doble clic sobre cualquier proceso invocaremos la ventana para administrar su migración (siguiente figura).

Un `'0'` significa 'local', los demás valores significan 'remoto'. Los procesos migrados están marcados con un icono verde y a los procesos que no pueden migrar se les añade un cerradero.

Hay también varias opciones para migrar el proceso remoto: enviarle las señales SIGSTOP y SIGCONT o simplemente cambiarle la prioridad, con el comando `nice` por ejemplo.

Si hacemos clic en el botón *manage procs from remote* invocaremos a una ventana emergente (llamada *remote-procs windows*) que nos mostrará el proceso actualmente migrado hacia ese nodo.

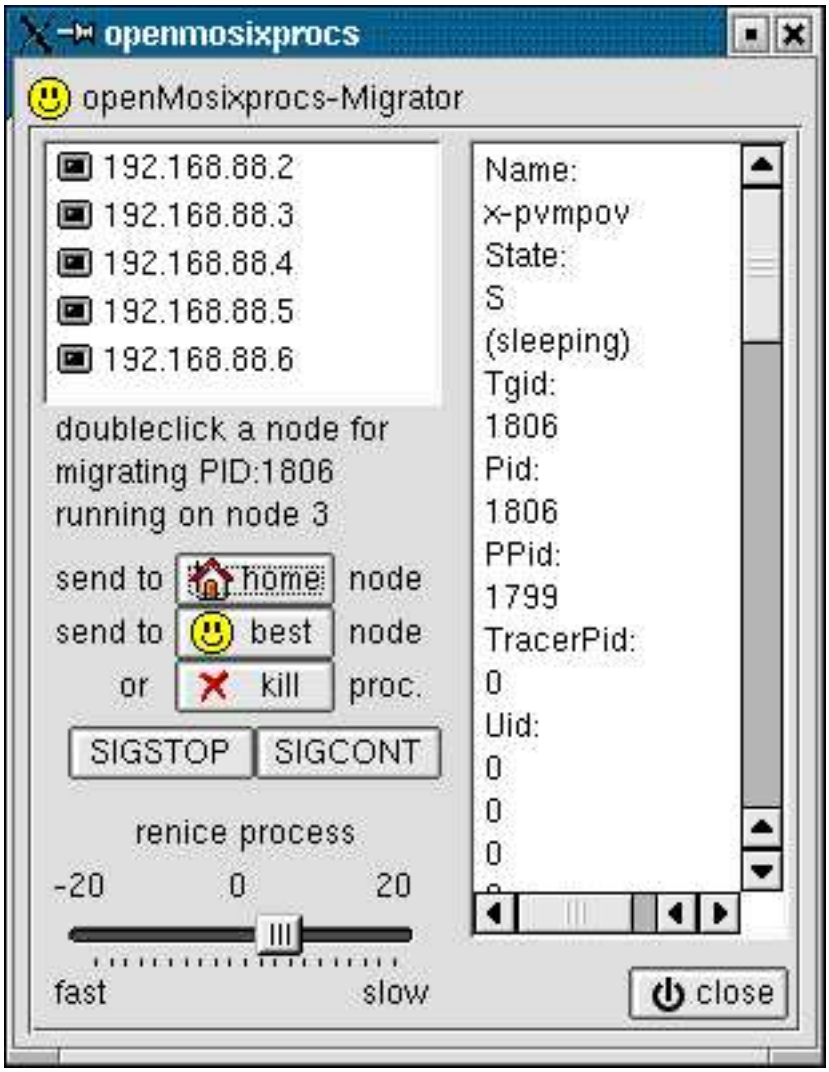



Figura 5.5: openMosixprocs: La ventana de migrado de un proceso(1)

**La ventana de migración**

El diálogo de la `Figura 5` emergerá si clicamos sobre cualquier proceso en la ventana anterior.

La ventana openMosixview-migrator muestra todos los nodos de nuestro cluster, arriba a la izquierda. Esta ventana sirve para administrar un proceso (con información adicional).

Si hacemos doble clic en un nodo migrará a este nodo.

Tras breves instantes el icono del proceso cambiará a verde, lo que significa que está siendo ejecutado remotamente.

El botón *home* envia un proceso a su nodo raíz. Con el botón *best* el proceso será enviado hacia el mejor nodo disponible en el cluster.

Esta migración se ve influenciada por la carga, velocidad, número de procesadores y su mayor o menor velocidad. Con el botón *kill* mataremos al proceso.

Para pausar un programa sólo tendremos que pulsar en el botón etiquetado como SIGSTOP y para reanudarlo, en SIGCONT.

Con el slider de `renice` podremos cambiar la prioridad de los procesos para una administración más completa, así el valor -20 se traduce en una mayor prioridad para terminar el trabajo, 0 le da prioridad normal y 20 le sitúa en una prioridad muy baja para tratarlo.

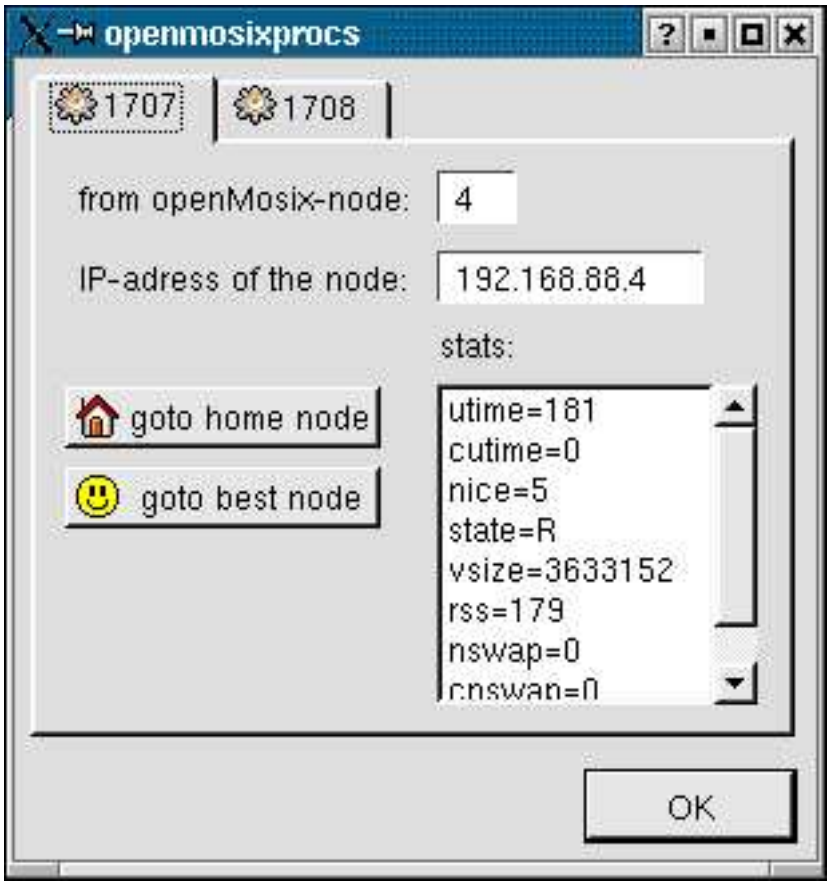



Figura 5.6: openMosixprocs: La ventana de migrado de un proceso(2)

**Administrando procesos remotamente**

El diálogo mostrado en la `Figura 6` aparecerá si clicamos en el botón *manage procs from remote*.

Las pestañas muestran los procesos que han migrado al nodo local. De forma parecida a los 2 botones en la ventana de migrado, el proceso será enviado a su nodo raíz si pulsamos *goto home node* y enviado al mejor nodo si pulsamos *goto best node*.

### 5.6.4. openMosixcollector

openMosixcollector es un *daemon* (demonio) que debe/puede ser invocado en cualquier miembro del cluster.

Genera un historial de la carga de cada nodo del cluster al directorio */tmp/openMosixcollector/** Este historial puede servir de entrada para openMosixanalyser (explicado posteriormente) para ofrecer una vista general de la carga, memoria y procesos en nuestro nodo durante un intervalo de tiempo.

Existe otro historial principal llamado */tmp/openMosixcollector/cluster*

Y adicionalemnte a éstos existen ficheros en los directorios donde se escriben los datos.

Al iniciar, openMosixcollector escribe su PID (ID del proceso) en */var/run/openMosixcollector.pid*

El demonio de openMosixcollector reinicia cada 12 horas y guarda el actual historial a */tmp/openMosixcollector[date]/** Estos *backups* se hacen automaticamente pero siempre podremos lanzarlos manualmente.

Hay una opción que permite escribir un *checkpoint* al historial. Estos *checkpoints* podemos verlos gráficamente como una fina línea azul vertical si analizamos el historial con openMosixanalyser. Por ejemplo podemos poner un *checkpoint* cuando iniciamos cierto trabajo en el cluster y otro cuando éste finaliza.

Aquí tenemos una referencia de los posibles argumentos para la consola:

`openMosixcollector -d` //inicia el collector como un daemon
`openMosixcollector -k` //detiene el collector

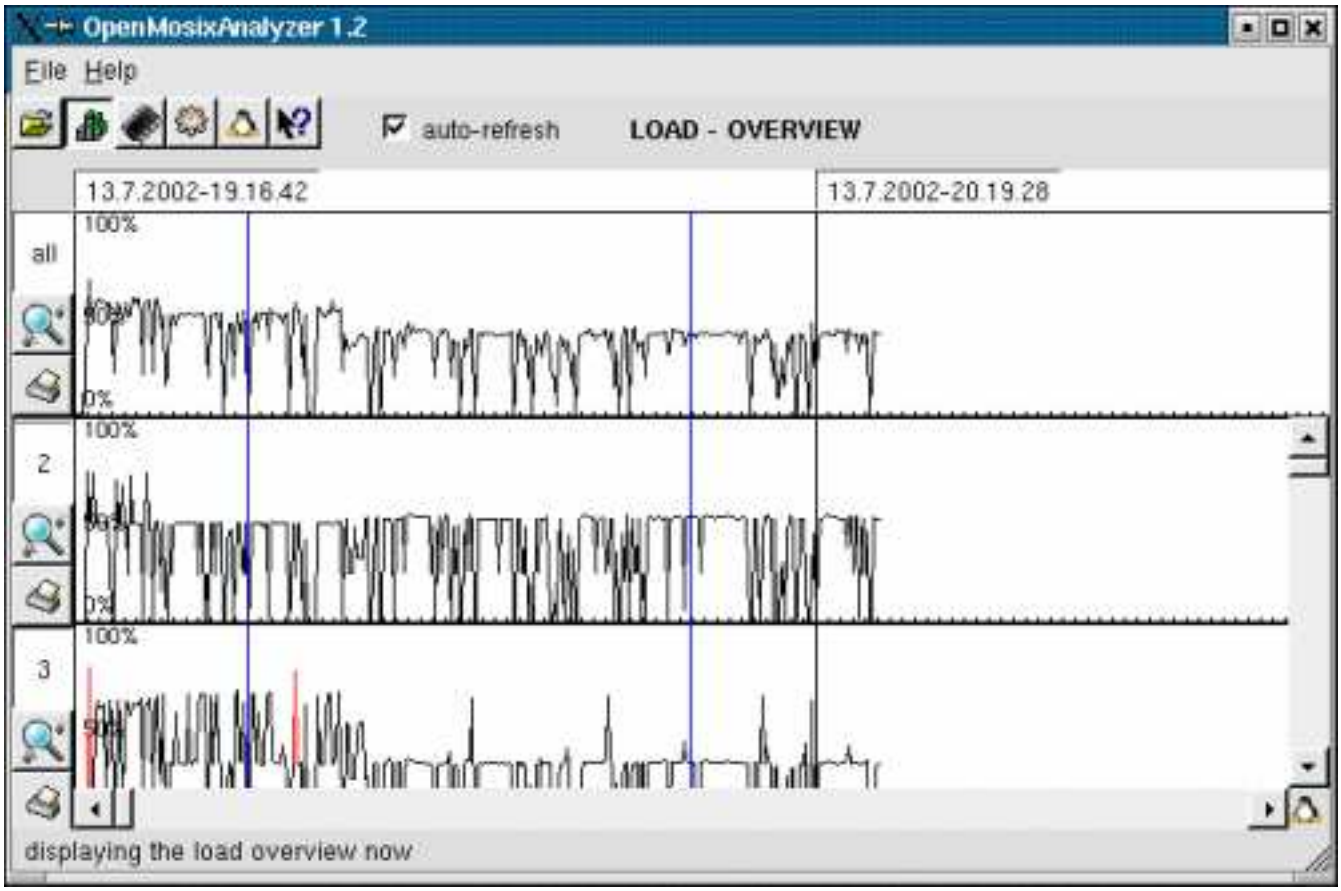



Figura 5.7: openMosixanalyzer. Historial de actividad de procesamiento del cluster

`openMosixcollector -n` //escribe un *checkpoint* en el historial
`openMosixcollector -r` //guarda el actual historial y empieza uno de nuevo
`openMosixcollector` //saca por la salida estándar una ayuda rápida

Podemos iniciar este demonio con su script de iniciación, que se encuentra en */etc/init.d o en /etc/rc.d/init.d* Hemos de tener un enlace simbólico a uno de los *runlevels* para un inicio automático.

La forma para analizar los historiales creados la describimos en la siguiente sección, el openMosixanalyzer.

### 5.6.5. openMosixanalyzer

**Una visión general de la carga del sistema**

La siguiente figura nos muestra de forma gráfica la carga en el openMosixanalyzer.

Con el openMosixanalyzer tendremos un historial contínuo de nuestro cluster. Los historiales generados por openMosixcollector se mostrarán ahora de forma gráfica, además de contínua, lo que nos permitirá ver la evolución del rendimiento y demás parámetros de nuestro cluster a través del tiempo. openMosixanalyzer puede analizar los historiales a tiempo real (datos generados a tiempo real) y evidentemente también puede abrir antiguos *backups* con el menu File.

Los historiales serán guardados en */tmp/openMosixcollector/** (y los backups los tendremos en */tmp/openMosixcollector[date]/**) y sólo tendremos que abrir el historial principal del cluster para visualizar antiguos historiales de informaciones de carga. (el campo [date] en los ficheros de backup se refiere a la fecha en que han sido guardados)

La hora de inicio del backup podemos verla en la parte superior.

Si utilizamos openMosixanalyzer para tratar historiales a tiempo real tendremos que activar el check *refresh*

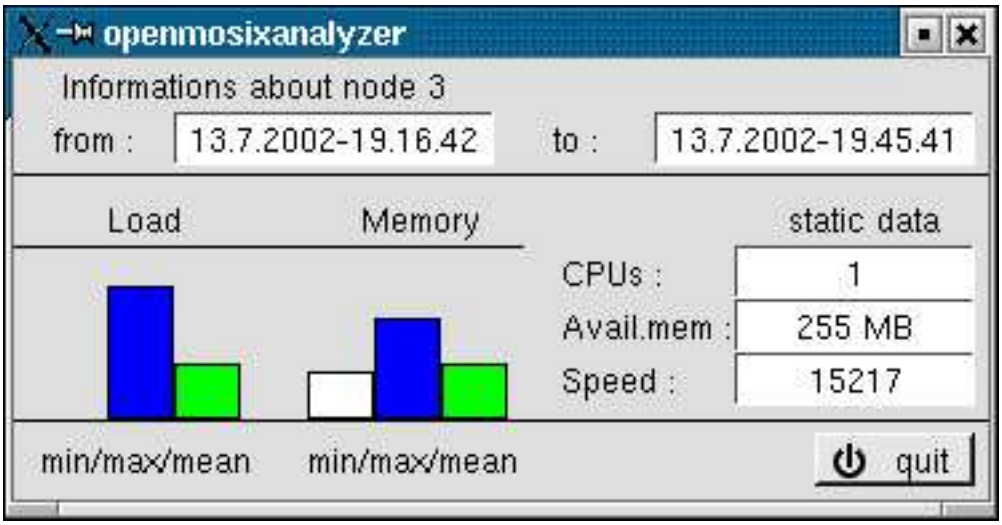



Figura 5.8: openMosixanalyzer. Estadísticas de los nodos

para que se actualice automáticamente.

Las líneas que indican la carga son normalmente de color negro. Si la carga se incrementa a >75 las líneas se volverán rojas.
Estos valores se obtienen desde los ficheros */proc/hpc/nodes/[openMosix ID]/**

El botón *Find-out* de cada nodo calcula diversos valores para las estadísticas. Si lo clicamos abriremos una nueva ventana en la cual veremos las medias de la carga y memoria y algunas informaciones adicionales (estáticas y dinámicas) sobre el nodo en concreto.

#### Estadísticas sobre los nodos del cluster

Si hay *checkpoints* escritos en el historial (generado por openMosixcollector) podremos verlos traducidos como líneas azules verticales. Esto permite comparar valores de carga a posteriori de forma fácil. Véase la `Figura 8`.

#### Monitorización de la memoria

La `Figura 9` muestra las gráficas de memoria proporcionadas por openMosixanalyzer

La filosofía de monitorización de la memoria es idéntica a la explicada anteriormente con la carga y evidentemente sirven también para poder ver como se ha comportado openMosix para administrar nuestros recursos de memoria, en este caso.

Igualmente podemos tener un monitorización a tiempo real o abrir antiguos historiales.

Para mostrar sus gráficas, openMosixnalyzer obtiene información de estos ficheros

/proc/hpc/nodes/[openMosix-ID]/mem.
/proc/hpc/nodes/[openMosix-ID]/rmem.
/proc/hpc/nodes/[openMosix-ID]/tmem.

Los *checkpoints* se muestran de igual forma que en la carga, con fina líneas verticales de color azul.

### 5.6.6. openMosixhistory

Con openMosixhistory podremos acceder a la lista de procesos ejecutados en el pasado, véase `Figura 10`. Conseguiremos un lista de los procesos que se ejecutaron en cada nodo. openMosixcollector guarda la lista de procesos de cada nodo cuando lo iniciamos y con el openMosixhistory podremos navegar en dicha información para ver el comportamiento que desarrolló nuestro cluster.

Podremos cambiar facilmente el tiempo de navegación con un *slider* situado en la parte superior, para así ajustar la ventana del pasado.

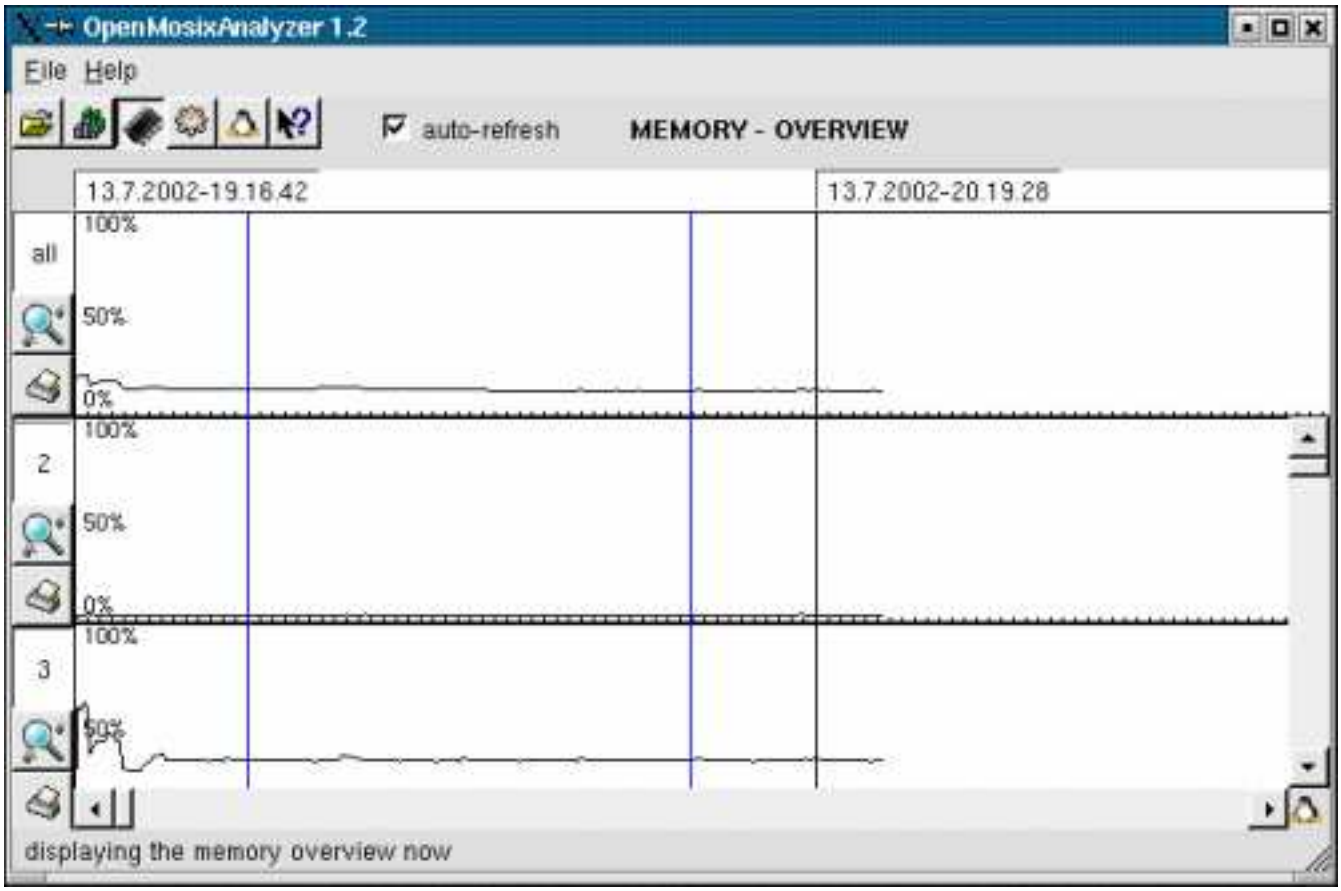



Figura 5.9: openMosixanalyzer. Historial de actividad de memoria de nuestro cluster

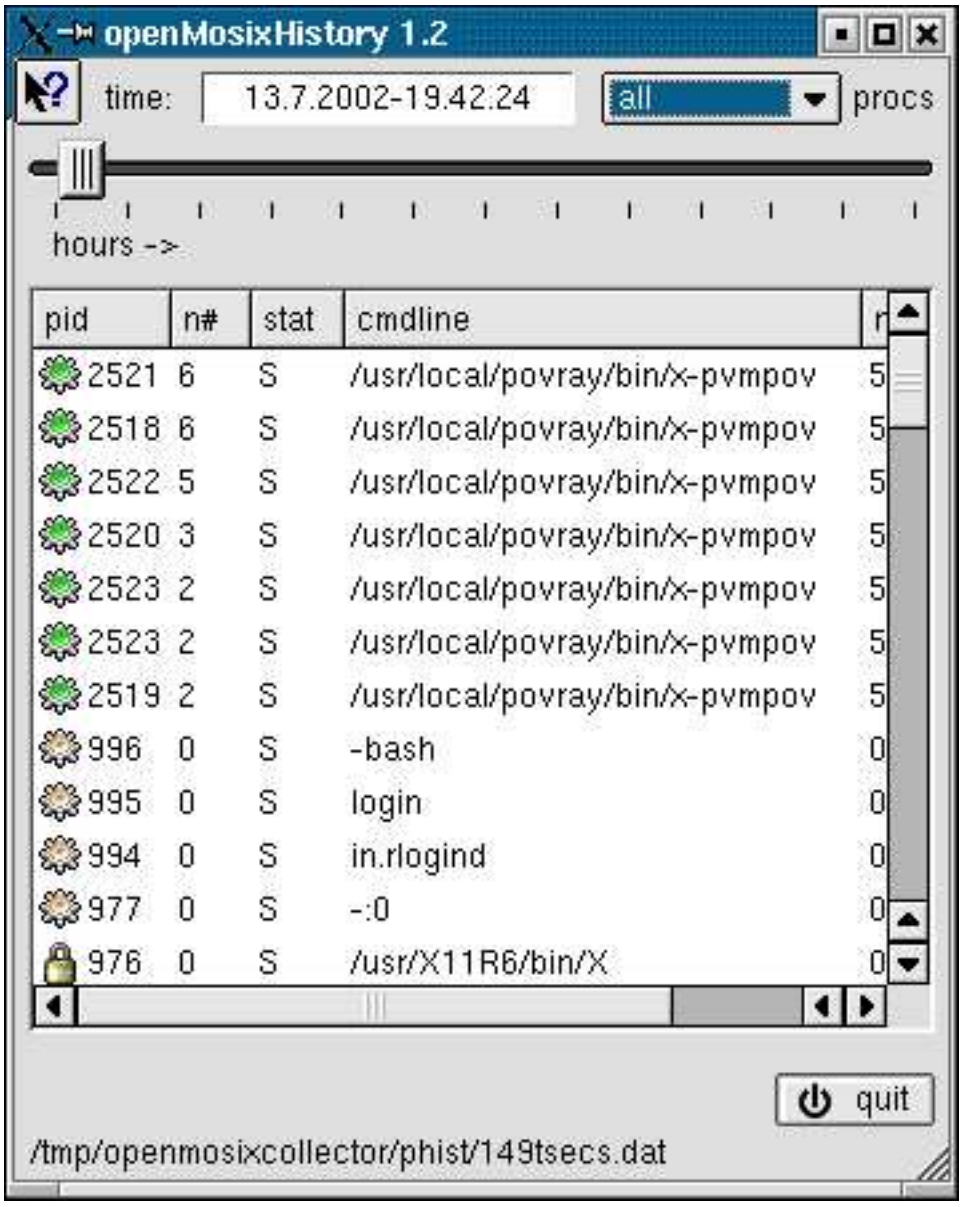



Figura 5.10: openMosixhistory. Un historial de los procesos ejecutados

openMosixhistory puede analizar en tiempo real los historiales, y también abrir antiguos de forma idéntica a como lo hemos podido hacer en los otros analizadores de datos explicados anteriormente.

Los historiales se obtienen nuevamente de */tmp/openMosixcollector/** y sólo tendremos que abrir el historial principal para poder navegar en los ficheros de información sobre la carga dada tiempo atrás.

El tiempo de inicio del monitorización podemos verlo en la parte superior y tenemos 12 horas de vista.

### 5.6.7. openMosixview + SSH2

***Texto original Matthias Rechenburg. Todo error tipográfico, gramatical, semántico u ortográfico envíenlo al traductor: Marcelo (*kuntx@kx99.hn.org*).***

Usted puede leer la razón por la que debe usar SSH en vez de RSH cada día en el diario, cuando otro script-kiddy haya hackeado un sistema/red inseguro. Entonces se dará cuenta que SSH es una buena decisión después de todo.

**Libertad x seguridad = constante** (sacado de un grupo de noticias de seguridad.)

Es por eso que es un poco difícil de instalar SSH. SSH es seguro incluso si usted lo usa para logarse sin usar contraseña. Seguidamente daremos una forma de como configuralo.

En principio es requerido que este corriendo el demonio del secure-shell.

Si no está instalado instálelo.

```
rpm -i [sshd_rpm_package_from_your_linux_distribution_cd]
```

Si no está ya corriendo inícielo con:

```
/etc/init.d/ssh start
```

Ahora usted debe generar en su computadora un par de llaves para el ssh hagalo con:

```
ssh-keygen
```

Se le pedirá una frase de contraseña para ese par de llaves creado. La frase de contraseña es normalmente más larga que la contraseña incluso llegando quizás a ser una oración completa. El par de llaves es encriptado con esa frase de contraseña y guardado en:

*root/.ssh/identity*//su llave privada

*/root/.ssh/identity.pub*//su llave pública

**¡¡No le entregue su llave privada a nadie!!**

Ahora copie el contenido completo de */root/.ssh/identity.pub* (su llave pública que debería tener una longitud de una línea) en */root/.ssh/authorized_keys* al host remoto (también copie el contenido de */root/.ssh/identity.pub* a su (local)*/root/.ssh/authorized_keys* como usted hizo con el nodo remoto, debido a que openMosixview necesita logar al nodo local sin que se le pida una contraseña.

Si ahora usted se trata de conectar mediante ssh al host remoto se le preguntará por la frase de contraseña de su llave pública respondiendo correctamente debería poder loguearse. ¿Cuál es la ventaja entonces?
La frase de contraseña es normalmente más larga que la contraseña.

La ventaja usted la puede obtener usando el ssh-agent. Éste maneja la frase de contraseña durante el ssh login:
`ssh-agent`

El ssh-agent es iniciado y entrega 2 variables de entorno Usted debe configurarlas (si no lo estan ya), escriba:

```
echo $SSH_AUTH_SOCK
```

y

```
echo $SSH_AGENT_PID
```

para ver si han sido exportadas a su shell. Si esto no es así corte y péguelas desde su terminal de la siguiente forma:

- Para bash-shell:

```
SSH_AUTH_SOCK=/tmp/ssh-XXYqbMRe/agent.1065
export SSH_AUTH_SOCK
SSH_AGENT_PID=1066
export SSH_AGENT_PID
```

- Para csh-shell:

```
setenv SSH_AUTH_SOCK /tmp/ssh-XXYqbMRe/agent.1065
setenv SSH_AGENT_PID 1066
```

Con estas variables el demonio sshd remoto puede conectarse a su ssh-agent local usando el archivo socket en */tmp* (en este ejemplo */tmp/ssh-XXYqbMRe/agent.1065*). El ssh-agent ahora puede darle la frase de contraseña al host remoto usando este socket (obviamente es una transferencia encriptada).

Usted apenas tiene que agregar su llave pública al ssh-agent con el comando ssh-add:

```
ssh-add
```

Ahora usted debe poder logarse usando ssh al host remoto sin que se le pida contraseña alguna. Usted puede (debe) agregar los comandos ssh-agent y ssh-add a su login profile:

```
eval ‘ssh-agent‘
ssh-add
```

Con ello será iniciado cada vez que se loguee en su estación de trabajo local. Felicidades, le deseo que tenga logins seguros de ahora en más con openMosixview.

Finalmente y ya para terminar con este COMO, hay una entrada en el menú de openmosixView que permite seleccionar con que trabajar rsh/ssh, actívela y usted prodrá usar openMosixview incluso en redes inseguras.También debe grabar esta configuración, (la posibilidad de grabar la configuración actual en openMosixview fue agregada a partir de la versión 0.7), porque toma datos de inicio del esclavo usando rsh ó ssh ( como usted lo haya configurado). Si usted elige un servicio que no está instalado correctamente openMosixview no funcionará.

Si usted no se puede conectar mediante rsh a un esclavo sin que se le pida una contraseña usted no podrá usar openMosixview con RSH.

Si usted no se puede conectar mediante ssh a un esclavo sin que se le pida una contraseña usted no podrá usar openMosixview con SSH.
Como creado por M. Rechenburg. Me olvidé de algo? Seguro. Envíeme un mail seguramente será agregado.

### 5.6.8. FAQ de openMosixview *-preguntas más frecuentes*

**¡No puedo compilar openMosixview en mi equipo!**

Antes que nada, como ya se ha comentado, es fundamental disponer de las librerías QT>=2.3.x.

La variable de entorno *QTDIR* tiene que estar configurada en *QT-installation* como se describe en el fichero de instalación INSTALL.

En versiones anteriores a la 0.6 podemos ejecutar `make clean` y borrar los dos ficheros:

```
/openmosixview/Makefile
/openmosixview/config.cache
```

y probar ahora de compilar otra vez ya que el autor, Matt Rechenburg, ha dejado los binarios y los objetos en estas versiones más antiguas.

En caso de tener otro tipo de problemas podemos postear en la lista de correo o directamente informar al autor.

**¿Puedo utilizar openMosixview con SSH?**

Sí desde la versión 0.7 se ha incorporado soporte para SSH.
Tendr'ıamos que ser capaces de acceder con ssh a cada nodo del cluster sin contraseña (lo mismo que es requerido para un acceso con rsh).

**He intentado iniciar openMosixview pero se me bloquea antes de iniciarse. ¿Qué he hecho mal?**

No deberemos ejecutar openMosixview en *background*, esto es llamarlo desde la consola con el comando `openMosixview&`.

También puede darse el caso que no podamos acceder a él a través de rsh/ssh (dependiendo de lo que usemos) como root sin contraseñas. Entonces deberíamos probar el comando `rsh hostname` como root. No se nos debería pedir ninguna contraseña pero se nos mostrará el shell para entrar nuestro login (nombre de usuario). Si estamos utilizando SSH el comando sería `ssh hostname`.

Tendremos que poder acceder como root en el sistema porque es el único modo de poder ejecutar los comandos administrativos que openMosixview requiere.

Si sólo tuviéramos instalado SSH en nuestro cluster, crearemos el fivhero */root/.openMosixview* y escribiremos `1111` en él.

Éste es el fichero de configuración principal y el último 1 se refiere a utilizar ssh en lugar de rsh.

**El cliente *openMosixviewprocs/mosixview_client* no funciona.**

El cliente openMosixview_client se ejecuta por rsh (o ssh, lo que tengamos configurado) en el host remoto. Debe estar instalado en */usr/bin/* en cada nodo.

Si usamos RSH podemos probar

```
xhost +hostname
rsh hostname /usr/bin/openMosixview_client -display nombre_de_host_local:0.0
```

y si usamos SSH

```
xhost +hostname
ssh hostname /usr/bin/openMosixview_client -display nombre_de_host_local:0.0
```

Si esto funciona funcionará también en openMosixview.

**openMosixview genera un *segmentation fault***

Puede que estemos utilizando versiones antiguas de openMosixview.

**¿Por qué los botones del diálogo openMosixview-configuration no están preseleccionados? (automigración on/off, bloqueo on/off...)**

Sería preferible que estuviesen preseleccionados, el problema se encuentra al obtener información del nodo.

Tenemos que logar en cada nodo ya que esta información no se encuentra dentro del cluster en sí .

El estado de cada nodo se guarda en el directorio */proc/hpc/admin/* de cada nodo.

## 5.7. PROBLEMAS MÁS COMUNES

*To err is human - and to blame it on a computer is even more so.*
*Robert Orben*

### 5.7.1. No veo todos los nodos

Antes de nada deberemos asegurarnos de estar utilizando la misma versión del kernel en todas las máquinas y que todos ellos hayan estado debidamente parcheados.

Ejecute `mosmon` y pulse `t` para ver el total de nodos que están funcionando. Para ver los nodos *muertos* simplemente bastará con darle a la tt d y aparecerá la palabra`DEAD` sobre todos aquellos IDs que correspondan a nodos que, apareciendo en el mapa de openmosix, no se encuentran funcionales.

Si este proceso le advierte que no tiene openMosix funcionando en alguna máquina -o no aparecen todas- asegúrese de haber incluido su dirección ip en el */etc/openmosix.map* (no use 127.0.0.1 ya que probablemente habrá problemas con el servidor de nombres/dhcp).

Si openMosix puede ver una máquina pero esta no recibe o envia los paquetes que debiera, entonces probablemente es que está protegida por un firewall que no deja actuar debidamente a openMosix.

Si el problema sigue sin solucionarse y se da el caso de tener varias targetas de red en cada nodo habrá que editarse el fichero */etc/hosts* para tener una línea con el formato

```
non-cluster_ip  cluster-hostname.cluster-domain cluster-hostname
```

Tendremos que configurar debidamente la tabla de enrutamiento, que podemos visualizar con

```
netstat -nr
```

No obstante esta tarea cae fuera del alcance de este libro.

Otros posibles problemas pueden venir por los diferentes parámetros de configuración del kernel, especialmente si usamos clusters con la opción de topología de red compleja. Con esa opción hay que vigilar de no utilizar el mismo valor que aparece en la opción de *Maximum network-topology complexity support* en cada nodo.

### 5.7.2. FAQ de openMosix *-preguntas más frecuentes-*

**General**

**¿Qué es openMosix?**

El sistema openMosix es una extensión del kernel del sistema operativo GNU/Linux que amplía grandemente sus posibilidades en computación paralela, a la vez que implementa una arquitectura SSI.

Está basado en el proyecto MOSIX del profesor A. Barak, pero con licencia GNU (General Public License, GPL).

**¿Qué significan las siglas SSI *-single system image-?***

Hay muchas variedades de clusters y un cluster SSI tiene copias de un único kernel de sistema operativo en varios nodos que permiten ver el conjunto como un todo.

Esto aporta ventajas a diversos niveles, entre los cuales se cuenta una mayor facilidad de uso y transprencia frente al usuario, procesos y los propios nodos.

**¿Existe una página web para openMosix?**

Sí es `http://www.openmosix.org`. También está la página web del proyecto en SourceForge en `http://sourceforge.ne`
.

**¿Hay una lista de correo para openMosix?**

Sí claro, existen dos:

- Para *discusiones generales* está `openmosix-general@lists.sourceforge.net` , cuya página de información general es `http://lists.sourceforge.net/lists/listinfo/openmosix-general`
- Para *desarrolladores* úsese `openmosix-devel@lists.sourceforge.net`, cuya página de información general es `http://lists.sourceforge.net/lists/listinfo/openmosix-devel`

**¿Puedo contribuir al proyecto openMosix?**

¡Por supuesto! El proyecto cuenta con un gran número de contribuidores. A diferencia del sistema de mantenimiento del kernel de linux, Moshe Bar (coordinador principal de openMosix) nombra desarrolladores oficiales y les otorga a éstos la clave *commit bit* del directorio CVS para el código fuente de openMosix, de manera similar al sistema usado para FreeBSD.

Se están buscando programadores familiarizados con el kernel de linux para trabajar en nuevas caracterí sticas. Si usted mismo está interesado escriba a `moshe@moelabs.com` .

**¿Quién posee la licencia de openMosix?**

Todo el código fuente de MOSIX está licenciado por el profesor Amnon Barak de la Hebrew University of Jerusalem. Todo el código fuente de openMosix está licensiado por Moshe Bar -residente en Tel Aviv-. El sistema openMosix no contiene ningún código fuente que no esté licenciado bajo la GPL.

**¿Es openMosix una versión de MOSIX?**

Originalmente openMosix era una versión de MOSIX, pero se ha desarrollado en una plataforma avanzada de clustering. Comparado con MOSIX, hay cierta cantidad de características que ya han sido agregadas:

- una versión para la arquitectura UML (linux de modo de usuario),
- nuevo y más ordenado código de migración,
- un mejor balanceador de procesos,
- reducción de latencias del kernel,
- soporte para Dolphin e IA64 (arquitectura 64-bit),
- un sistema de instalación simplificado usando paquetes RPM,
- gran cantidad de documentatción,
- y gran número de parches[11] sobre el propio de openMosix para dotarle de mejores caracteristicas.

**¿Por qué se separó openMosix de MOSIX?**

La cuestión principal era que MOSIX **no** está licenciado bajo una licencia abierta. Esto evita que puedas ver el código fuente, que puedas desarrollar e implementar las mejoras que creas convenientes y, sobretodo, que crezca la comunidad de usuarios.

### Obteniendo, compilando, instalando y funcionando con openMosix

**¿Dónde puedo conseguir openMosix?**

Los fuentes oficiales para openMosix pueden encontrarse en SourceForge [12]. Asegúrese de leer la documentación antes de bajar los archivos para comprobar que adquiere la versión adecuada a su sistema.

**¿Puedo mezclar nodos MOSIX y openMosix en el mismo cluster?**

[11] Se trata de código beta que se agrega a openMosix una vez se garantiza su funcionalidad.

[12] `http://sourceforge.net/project/showfiles.php?group_id=46729`

¡NO! Como en MOSIX, no es recomendado mezclar nodos porque los protocolos están ligados a cambiar sin anuncios entre versiones. Además, cada nueva versión contempla correcciones de errores anteriores que vale la pena eliminar.

**¿Cómo compilar openMosix?**

Primero desempaquete tanto el código fuente del kernel de linux como la distribución correspondiente de openMosix en un directorio, por ejemplo

```
$ cd /usr/src
$ tar xzvf linux-2.X.Y.tar.gz
$ gunzip openMosix2.X.Y.gz
$ cd linux-2.X.Y
```

Luego aplique los parches de openMosix al código fuente original del kernel de linux con el comando

```
$ patch -p1 openMosix2.X.Y-Z
```

El directorio */usr/src/linux-2.X.Y* contiene el código fuente del kernel 2.X.Y de linux con los parches de openMosix aplicados. Compile e instale el nuevo kernel como cualquier otro.

**¿Qué son las *userland-tools* -herramientas de usuario-?**

Las herramientas de usuario son una colección de herramientas administrativas para examinar y controlar un nodo openMosix. Son estrictamente necesarias y pueden obtenerse desdel mismo directorio donde se encuentra en parche para el kernel.

Para mayores problemas consulte la sección *Las herramientas de área de usuario* del capítulo 5.

**Preguntas del kernel (núcleo)**

**¿Qué versiones del kernel están soportadas por openMosix?**

El primer kernel kernel de linux soportado es el 2.4.17. Versiones futuras del la serie 2.4 -la serie 2.6- también van a ser soportadas por openMosix.

Transcurre poco más de una semana entre la aparicición de un nuevo kernel linux y el parche openMosix.

**Estoy tratando del compilar el kernel con el parche de openMosix. ¿Qué versión de compilador debo usar?**

La versión recomendada para un kernel parcheado con openMosix es la misma que para la versión del kernel linux. Por tanto la correcta compilación por parte de `gcc` es no sólo un requerimiento de openMosix, sino del kernel de linux.

**He compilado el kernel de las fuentes. ¿Cómo hago para agregarlo al cargador de inicio (*bootloader*) (LILO, GRUB, otro)?**

El kernel de openMosix es como cualquier otro kernel porque openMosix es tan solo una extensión del kernel, y va a ser tratado como un kernel estándar. Use los métodos normales del cargador de inicio para agregar un kernel.

**He instalado una distribución de linux que viene con el kernel X.Y. El README de openMosix dice que no debemos mezclar kernels. ¿Significa esto que el RPM para el kernel openmosix-X.Y+1-Z no va a funcionar en mi máquina?**

No. Cuando se dice que no se deben mezclar versiones de kernels se refiere a los parches de openMosix. Mientras todas las máquinas usen las mismas versiones del kernel de openMosix no habrá problemas.

Nótese que con versión nos referimos al número de versión de openMosix; esto implica que se pueden mezclar kernels de openMosix que son para diferentes arquitecturas de hardware. Por ejemplo, instalando el kernel openmosix-2.4.18-4 se pueden usar tanto el RPM openmosix-2.4.18-4-i686 en una máquina con Pentium II y el kernel openmosix-2.4.18-4-athlon en una máquina con un procesador Athlon.

### Sistema de ficheros

**¿Cuál es la diferencia entre MFS y DFSA, y por qué me conviene tener DFSA?**

DFSA es el acrónimo de *Direct File System Access* (acceso directo al sistema de archivos) y es una optimización. Permite que procesos remotos ejecuten llamadas de sistema localmente en vez de mandarlas al nodo local, factor que mejora sustancialmente el rendimiento de sistemas SSI. MFS significa Mosix File System (sistema de archivos Mosix) y permite que todos los nodos en un cluster tengan acceso al sistema de archivos de todos los otros nodos. DFSA se ejecuta encima de un sistema de archivos de cluster, en este caso MFS.

**¿Qué es MFS, cómo lo uso, y dónde lo consigo?**

El sistema de archivos de openMosix (MFS) es la implementación de MFS en openMosix (véase la pregunta anterior para detalles acerca de MFS). El soporte para el tipo `mfs` se adquiere con la configuración de un kernel openMosix con la opción MFS (viene estándar con los archivos RPM). Si ud. instala su propio kernel, le sugerimos que también use la opción DFSA (vea la pregunta anterior para detalles acerca de DFSA). El uso y la administración de MFS es muy similar a NFS, pero MFS provee ciertas ventajas sobre NFS como

- consistencia de cache
- timestamp
- link consistency

### Programando openMosix

**En general, ¿cómo escribo un programa que utilice openMosix?**

Escriba sus programas de manera normal. Cualquier proceso que sea creado es un candidato a ser migrado a otro nodo.

Un consejo para poder aprovechar al máximo la tecnología que openMosix pone a su alcance sería servirse de la sentencia *fork()* de UNIX. Como se ha comentado en este documento, openMosix hereda de MOSIX el *fork-and-forget* que realiza la migración automática de cualquier nuevo proceso -siempre que se mejore el rendimiento del cluster-.

**¿Es posible escribir programas para openMosix con Perl?**

Evidentemente sí . Ud. deberá usar `Parallel::ForkManager`, disponible en CPAN.

### Miscelánea

**Estoy recibiendo un mensaje que advierte de**

```
setpe the supplied table is well-formatted,
but my IP address (127.0.0.1) is not there!
```

Se deberá modificar el archivo */etc/hosts*, donde se debe aparecer una línea similar a

```
127.0.0.1 localhost
```

para que todo funcione correctamente. Por ejemplo

```
192.168.10.250  hostname.domain.com
127.0.0.1       localhost
```

**Quiero instalar openMosix en mi máquina, pero temo que es demasiado lenta.**

El proyecto opeMosix no se desentiende de la filosofía de linux frente al hardware antiguo. Cualquier máquina puede beneficiarse con el uso de openMosix mientras sea del x586 compatible

El cluster puede ser heterogéneo, es decir, ni siquiera hace falta tener todas las máquinas del mismo tipo.

**¿Bajo qué condiciones se puede usar VMware con openMosix?**
Si se desea ejecutar VMware en una máquina con openMosix instalado para aprovechar la capacidad de computación de las máquinas remotas, no hay problema. Por otro lado, ud. no podrá correr openMosix dentro de sesiones de VMware y dejar que las sesiones balanceen su carga porque VMware tiene un defecto en su emulación de Pentium que causa que VMware (no openMosix) falle cuando un proceso migre.

**¿Qué arquitecturas aparte de x86 (por ejemplo SPARC, AXP, PPC) son soportadas por openMosix?**
Por el momento openMosix solo corre en x86.

**¿Existe una utilidad paralela de `make` para openMosix similar a `MPmake`?**
Se puede compilar usando el `make` de gcc con la opción `-j`. Así por ejemplo

```
make -j 10
```

ejecuta make con 10 procesos en paralelo.

## 5.8. PARA MÁS INFORMACIÓN

*The important thing is not to stop questioning.*

*Albert Einstein*

Para obtener información actualizada inscríbete a las listas de correo[13] del proyecto openMosix y de openMosixview[14].

Puede ser realmente útil ver la página *wiki* que tiene montada Kris Buytaert[15], donde podrás encontrar los aspectos pendientes de documentación y los comentarios sobre testeos de diferentes instalaciones de clusters que algunos usuarios han querido compartir con la comunidad. También tiene una lista de enlaces.

Puedes leer el log de la conferencia que dio David Santo Orcero[16], desarrollador de las herramientas de usuario, para UMEET2002. Puedes leer la traducción en castellano[17] -en el canal #redes- .

[13] http://openmosix.sourceforge.net/mailing.html
[14] http://lists.sourceforge.net/lists/listinfo/mosixview-user
[15] http://howto.ipng.be/openMosixWiki/
[16] http://umeet.uninet.edu/umeet2002/talk/2002-12-17-linux1.txt.html
[17] http://umeet.uninet.edu/umeet2002/talk/2002-12-17-redes1.txt.html

# Capítulo 6

# openMosix a fondo

## 6.1. *The openMosix internals* (Moshe Bar)

*In the good old days physicists repeated each other's experiments, just to be sure.*
*Today they stick to FORTRAN, so that they can share each other's programs, bugs included.*
*Edsger W. Dijkstra*

## Aspectos generales de openMosix

Un cluster openMosix es del tipo SSI. Se ha argumentado mucho sobre si estas arquitecturas pueden considerarse un cluster como tal. Los primeros clusters SSI fueron el IBM SySPlex y un cluster de DEC. En el cluster DEC se podía acceder con telnet a la dirección del cluster y trabajar desde allí hasta que decidiéramos terminar la conexión. El usuario nunca tenía consciencia sobre en cual de los nodos se encontraba y cualquier programa que se lanzara se ejecutaba en el nodo que mejor pudiera servir las necesidades del proceso.

Bien, openMosix es exactamente lo mismo. Sólo que funciona sobre nuestro sistema operativo favorito: Linux!

openMosix es una extensión del kernel. Antes de instalar openMosix tendremos que lanzar el *script* de instalación que hará efectivos los cambios necesarios al kernel de linux -por lo tanto deberemos disponer de las fuentes del kernel en nuestro disco-. Los cambios se traducen entorno a un 3 % del código total del núcleo, lo que realmente no es muy significativo.

Cuando habremos recompilado el nuevo kernel y tras arrancarlo, dispondremos de un nuevo nodo openMosix. Haciendo réplicas de este proceso en las máquinas que tengamos en nuestra red llegaremos a crear el cluster.

Hay un fichero de configuración en */etc/openmosix.map* que debe ser configurado para dejar ver en el nodo local -el poseedor del fichero- los demás nodos. Cada nodo envía su estado de carga actual a una lista aleatoria de otros nodos, para hacerlo conocer. La mejor característica de openMosix es que podemos ejecutar un programa y el mismo cluster decide dónde debe ejecutarse.

openMosix nos brinda una nueva dimensión de la escalabilidad de un cluster bajo linux. Permite la construcción de arquitecturas de alto rendimiento, donde la escalabilidad nunca se convierte en un cuello de botella para el rendimiento general del cluster. Otra ventaja de los sistemas openMosix es la respuesta en condiciones de alta impredicibilidad producida por distintos usuarios y/o situaciones.

Entre las características más relevantes se encuentra su política de distribución adaptativa y la simetría y flexibilidad de su configuración. El efecto combinado de estas propiedades implica que los usuarios no tengan que saber el estado actual de los distintos nodos, ni tan siquiera su número. Las aplicaciones paralelas pueden ser ejecutadas permitiendo el asignamiento del lugar de ejecución de los procesos a openMosix. Como puede darse en sistemas SMP.

No obstante hay áreas donde openMosix no genera grandes beneficios. Para aplicaciones que usen memoria compartida (como servidores de web o de bases de datos) los procesos no podrán migrar y por tanto permanecerán en el nodo desde el cual se han invocado.

## Detalles de openMosix

openMosix es una herramienta para kernels Unix, como Linux, consistente en unos algoritmos para adaptación de recursos compartidos. Permite múltiples nodos uniprocesadores -UPs- y/o SMPs que funcionen bajo la misma versión del kernel para cooperar conjuntamente. Los algoritmos de openMosix se han diseñado para responder a variaciones a tiempo real en el uso de los recursos de varios nodos. Esto es posible migrando procesos de un nodo a otro transparentemente, teniendo en cuenta la carga de cada uno de ellos y evitando errores debido al uso de memoria *swap*. El éxito es una mejora en el rendimiento total y la creación de un entorno multiusuario y de tiempo compartido para la ejecución de aplicaciones, tanto secuenciales como paralelas. El entorno de trabajo estándar de openMosix es un cluster donde se encuentran disponibles todos y cada uno de los recursos de cada nodo.

La implementación actual de openMosix se ha diseñado para configurar clusters con máquinas basadas en x86 conectadas en redes LAN estándar. Las posibles configuraciones abarcan desde pequeños clusters con redes de 10Mbps hasta entornos con servidores SMP y redes ATM, Gigabit LAN o Myrinet. La tecnología openMosix consiste en dos partes:

1. un **mecanismo de migración de procesos** llamado Migración Preferente de Procesos (PPM),
2. un conjunto de dos **algoritmos para la compartición adaptativa de recursos**,
3. y un sistema para el **acceso a ficheros** de cualquier nodo del cluster.

Ambas partes están implementadas a nivel de kernel utilizando módulos, la interfaz del kernel permanece sin modificar. Todo ello permanece transparente a nivel de aplicación. Se profundiza sobre cada una de ellas en las próximas subsecciones.

La PPM puede migrar cualquier proceso, en cualquier instante de tiempo y a cualquier nodo disponible. Usualmente las migraciones estan basadas en información provinente de uno de los algoritmos de compartición de recursos, pero los usuarios pueden invalidar cualquier decisión automática y hacer las migraciones manualmente.

Una migración debe ser iniciada por el proceso o por una petición explícita de otro proceso del mismo usuario (o del super-usuario root). Las migraciones manuales de procesos pueden ser útiles para implementar una política particular o para testear diferentes algoritmos de planificación (*scheduling*). Cabe destacar que el super-usuario tiene privilegios adicionales sobre la PPM, como la definición de políticas generales o qué nodos podrán ser incluidos en el cluster y cuáles no.

Cada proceso tiene un único nodo de origen (UHN, *unique home node*) donde es creado. Normalmente éste es el nodo en el cual el usuario ha logado. El modelo SSI de openMosix es un modelo CC (*cache coherent*), en el cual cualquier proceso parece ejecutarse en su nodo local y donde todos los procesos de una sesión de usuario comparten el entorno de la ejecución del UHN. Los procesos que migran a otro nodo usarán los recursos de éste, siempre que sea posible, pero interaccionarán con el usuario a través del UHN. Por ejemplo, podemos suponer que un usuario invoca ciertos procesos, algunos de los cuales pasan a migrarse a nodos remotos. Si el usuario ejecuta a continuación el comando `ps` podrá ver el estado de todos sus procesos, incluyéndose los procesos migrados. Si alguno de los procesos migrados pide la hora actual a través de la primitiva `gettimeofday()` obtendrá la hora del UHN.

La política que implementa PPM es la principal herramienta para los algoritmos de gestión de los recursos. Mientras recursos como el uso de procesador o de memoria principal se encuentren bajo cierto umbral, los procesos serán confinados al UHN. Cuando los requerimientos de los recursos exceda de ciertos niveles, algunos procesos se migrarán a otros nodos para aprovechar las ventajas de los recursos remotos. La principal meta es maximizar la eficiencia de recursos a través de la red. La unidad de granularidad del trabajo de distribución en openMosix es el proceso[1]. Los usuarios pueden ejecutar aplicaciones paralelas inicializando múltiples procesos en un nodo, y luego permitiendo al sistema la asignación de dichos procesos a los mejores nodos disponibles. Si durante la ejecución de los procesos aparecen nuevos recursos usables, los algoritmos de gestión de recursos compartidos se encargarán de tenerlos en cuenta. Esta capacidad para asignar y reasignar procesos es particularmente importante para facilitar la ejecución y proporcionar un entorno multiusuario y de tiempo compartido eficiente.

openMosix no dispone de un control centralizado -o jerarquizado en maestros/esclavos-: cada nodo puede operar como un sistema autónomo, y establece independientemente sus propias decisiones. Este diseño permite la configuración dinámica, donde los nodos pueden añadirse o dejar el cluster con interrupciones mínimas. Adicionalmente permite una gran escalabilidad y asegura que el sistema permanezca funcionando correctamente en grandes o pequeñas configuraciones. Dicha escalabilidad se consigue incorporando aleatoriedad en los algoritmos de control del sistema, donde cada nodo basa sus decisiones en un conocimiento parcial sobre el estado de cada uno de los demás nodos. Por ejemplo, en el algoritmo probabilístico de difusión de información, cada nodo envía, a intervalos regulares, información sobre sus recursos disponibles a un conjunto de nodos elegidos aleatoriamente[2]. Al mismo tiempo, mantiene una pequeña ventana con la información llegada más reciente. Este esquema soporta escalabilidad, difusión de información uniforme y configuraciones dinánimcas.

### Mecanismo de migración de procesos PPM

La mejor aportación de openMosix es un nuevo algoritmo para seleccionar en qué nodo debe ejecutarse un proceso ya iniciado. **El modelo matemático para este algoritmo de planificación (*scheduling*) se debe al campo de la investigación económica**. Determinar la localización óptima para un trabajo es un problema complicado. La complicación más importante es que los recursos disponibles en el cluster de computadoras Linux son heterogéneos. En efecto, el coste para memoria, procesadores, comunicación entre procesos son incomparables: no se miden con las mismas unidades. Los recursos de comunicación se miden en términos de ancho de banda, la memoria en términos de espacio libre y los procesadores en términos de ciclos por segundo.

[1] Algunos clusters experimentales disponen de una granularidad más fina, permitiendo la migración de *threads* (hilos).
[2] Si enviase la información de su carga a todos los nodos, el nodo con menor carga pronto se vería sobrecargado.

openMosix implementa la migración de procesos completamente transparente al usuario, al nodo y al proceso. Tras su migración el proceso continúa interaccionando con el entorno de su localización. Para implementar PPM la migración de procesos se divide en dos áreas:

1. **remote:** es el área de usuario y puede ser migrada.
2. **deputy:** es el área de sistema, que es dependiente del UHN y que no puede migrar.

A continuación se detalla sobre cada una de ellas.

#### I.- Remote

Contiene el código del programa, el **stack**, los **datos**, los **mapas de memoria** y los **registros de los procesos**. *Remote* encapsula el proceso cuando está ejecutándose a nivel de usuario.

#### II.- Deputy

El área de sistema, llamada también *parte delegada*, contiene una descripción de los recursos a los que el proceso está ligado y un *kernel-stack* para la ejecución del **código de sistema del proceso**. *Deputy* encapsula el proceso cuando está ejecutándose en el kernel. Mantiene la dependencia de las partes del contexto del sistema de los procesos, por lo tanto debe mantenerse en el UHN. Mientras el proceso puede migrar -y varias veces- entre nodos diferentes, el *deputy* nunca lo hará. La interfície entre las dos áreas está bien definida en la capa de enlace, con un canal de comunicación especial para la interacción de tipo *mosix_link*.

El tiempo de migración tiene dos componentes:

- un componente fijo, para establecer un nuevo marco de proceso en el remoto,
- y un componente linear, proporcional al número de páginas de memoria a transferir.

Para minimizar el tráfico de migraciones sólo se transfieren las tablas de páginas y las páginas ocupadas por los procesos.

Las llamadas al sistema son una forma síncrona de interacción entre las dos áreas. Todas las que son ejecutadas por el proceso son interceptadas por la capa de enlace del remoto. Si la llamada de sistema no depende del UHN se ejecuta -localmente- en el nodo remoto. Si no, la llamada de sistema es redirigida al *deputy* el cual ejecuta la llamada a sistema del proceso.

Otras formas de interacción entre las dos áreas son la entrega de señales y los eventos para reactivar procesos, a medida que lleguen datos por la red. Estos eventos requieren que el *deputy* localice e interaccione asíncronamente con el remoto.

Este requisito de localización *mosix_link* es resuelto por el canal de comunicaciones existente entre ellos. En un escenario típico, el kernel del UHN informa al *deputy* del evento. *Remote* monitoriza el canal de comunicación para percatarse de eventos asíncronos, como pudieran ser señales, justo antes de volver a la ejecución de nivel de usuario.

Una desventaja es la sobrecarga en la ejecución de las llamadas al sistema y sobretodo por accesos a la red. Todos los enlaces de red -*sockets*- son creados en el UHN, así se impone comunicación a través de ellos si el proceso migra fuera del UHN. Para solucionar este problema el equipo de la Universidad Hebrea está desarrollando los **socketa migrables**, los cuales migran con el proceso y así permite un enlace directo entre los procesos migrados. Normalmente esta carga que añadimos puede reducirse significativamente con una distribución inicial de los procesos de comunicación en nodos diferentes, como en PVM. Si el sistema se desequilibra, los algoritmos de openMosix reasignarán los procesos para mejorar la eficiencia del sistema.

**Los algoritmos para la compartición adaptativa de recursos**

Los algoritmos de compartición de recursos en openMosix son dos:

1. el de **equilibrado dinámico de carga:** *Load Balancing*,
2. y el encargado de la **gestión de memoria:** *Memory Ushering*.

I.- LOAD BALANCING

El algoritmo de equilibrado dinámico de carga continuamente intenta reducir la diferencia de carga entre pares de nodos, migrando procesos desde los más cargados hacia los menos cargados, independientemente del par de nodos. El número de procesadores de cada nodo y su velocidad son factores importantes para tomar este tipo de decisiones. Este algoritmo responde a cambios en la carga de los nodos o a las características en la ejecución de los procesos. A lo largo de este proceso de toma de decisiones prevalece el porcentage de memoria libre o la carga de procesos y no la escasez de recursos.

II.- MEMORY USHERING

*Memory Ushering* inspira su nombre en una pol´tica que los desarrolladores de openMosix llaman *prevención de agotamiento*. Intenta disponer el máximo número de procesos en la RAM del sistema -entendiéndose la totalidad de la memoria del cluster- para evitar en lo posible el *thrashing* y el *swapping* de los procesos. El algoritmo se invoca cuando un nodo empieza una paginación excesiva, debida generalmente a una escasez de memoria libre.

### Acceso a ficheros

Uno de los mayores problemas de los clusters SSI es que cada nodo tiene que ser capaz de ver el mismo sistema de ficheros que cualquier otro nodo. Pongamos por ejemplo un programa que abre el fichero */tmp/moshe* para escritura y lectura, y luego este proceso migra a otro nodo del cluster. El fichero en cuestión deberá ser capaz de permanecer como de lectura y escritura.

Hasta ahora había dos opciones para hacer esto. Una es que el cluster openMosix interceptara todas las E/S de los trabajos migrados, y las reenviara a su correspondiente UHN para su procesamiento posterior.

Alternativamente podría crearse una visión global de un sistema de ficheros a través de NFS. La primera es más difícil de desarrollar, pero más fácil de gestionar. La segunda es más fácil de implementar, pero puede ser un rompecabezas montar todos los sistemas de ficheros de manera inteligente, permitiendo que cada nodo pueda acceder a cualquier otro nodo. Adicionalmente, se tendría que asegurar que todos los identificadores de usuario y de grupo son consistentes entre todos los nodos del cluster, de otro modo nos toparíamos con serios problemas de permisos.

Buscando entre las mejores investigaciones de los modernos sistemas de ficheros y aplicando estas ideas a openMosix surgió DFSA (Direct File System Access). El DFSA fue diseñado para reducir el exceso de carga de la ejecución de llamadas de sistema orientadas a E/S de los procesos migrados. Esto se consiguió permitiendo la ejecución de la mayoría de llamadas a sistema de forma local, es decir, en el nodo donde se encuentre el proceso. En adición a DFSA se generó un nuevo algoritmo que hace un recuento de las operaciones de E/S. Este algoritmo se basa en que a un proceso que requiera un número medio/alto de operaciones de E/S se le tiende a migrar al nodo donde realizará la mayoría de estas operaciones. Una ventaja obvia es que los procesos de E/S tienen mayor flexibilidad para migrar desde su respectivo UHN para mejorar el equilibrado de carga. No obstante, a diferencia de NFS que sirve los datos desde el servidor hacia los nodos, openMosix intentará migrar el proceso al nodo en el que el fichero resida en aquel momento.

## La implementación

### Mecanismos en el *remote* y en el *deputy*

El *deputy* es la parte que representa al proceso remoto en el UHN. Debido a que la totalidad del espacio de usuario en memoria reside en el nodo remoto, el *deputy* no mantiene un mapa de memoria de él mismo. En lugar de esto comparte el mapa principal del kernel de forma similar a una hebra del kernel.

En algunas actividades del kernel, como pueden ser la ejección de las llamadas de sistema, se hace necesario transferir datos entre el espacio de usuario y el espacio de kernel. En openMosix cualquier operación de memoria del kernel que implique acceso al espacio de usuario requiere que el *deputy* comunique con su correspondiente *remote* para transferir los datos necesarios.

Con el fin de poder eliminar excesivas copias remotas se implementó una cache especial que reduce el número de interacciones requeridas con *pre-fetching*, transfiriendo el mayor volumen de datos posible durante la petición inicial de llamada a sistema, mientras se almacenan datos parciales en el *deputy* para devolverlos al remoto al final de la llamada a sistema.

Los procesos remotos no son accesibles por los otros procesos que se ejecutan en el mismo nodo y *vice versa*. No pertenecen a ningún usuario en particular en el nodo donde se ejecutan ni pueden recibir señales desde el nodo en el que se estan ejecutando. Su memoria no puede ser accedida y solo pueden ser forzados, por parte del propio administrador del nodo en el que se están ejecutando, a migrar fuera del nodo.

#### Cuando no puede aplicarse la migración

Ciertas funciones del kernel de Linux no son compatibles con la división de áreas de los procesos, para que estas se ejecuten remotamente. Algunos ejemplos obvios son las manipulaciones directas de dispositivos de E/S, tales como accesos directos a instrucciones privilegiadas del bus de E/S, o el acceso directo a la memoria del dispositivo. Otros ejemplos incluyen memoria compartida de escritura y planificadores a tiempo real.

Un proceso que usa alguna de estas funciones es asignado indefinidamente a su UHN. Si el proceso ya había estado migrado, se retorna al UHN.

## Muestreo de información

Las estadísticas acerca del comportamiento del cluster son muestreadas regularmente. Estas incluyen todas las llamadas de sistema o cada vez que un proceso accede a datos de usuario. Esta información es usada para determinar si el proceso debe devolverse a su UHN. Estas estadísticas caducan en el tiempo, para ajustarse a procesos que cambian su perfil de ejecución.

Cada proceso tiene cierto control sobre el muestreo de información y sobre la caducidad de sus estadísticas. Toda la información acerca del API de openMosix puede encontrarse en la sección *El API de openMosix*.

# 6.2. MODELIZACIÓN MATEMÁTICA DE PROCEDIMIENTOS

*Pure mathematics is, in its way, the poetry of logical ideas.*
*Albert Einstein*

## Formalización de recursos. Objectivo de rendimento

La formalización de los recursos que posee cada nodo permitirá construir una visión general del funcionamiento de ponderación de tales recursos dentro del cluster. La nomenclatura a partir de ahora utilizada identificará

- $I = \{i_1, i_2, ..., i_n\}, \quad n \in \mathbb{N} \quad \equiv$ el conjunto de $n$ nodos que conforman el cluster.

Luego los $m$ recursos que posee un nodo $i_k$ cualquiera se identificaran por el conjunto $R(i_k)$ de esta forma:

- $R(i_k) = \{r_c(i_k),\ r_m(i_k),\ ...\}, \quad 1 \leq k \leq n \quad \equiv$ conjunto de recursos del nodo $i_k$ .
- $|R(i_k)| = m$ .

Esto permite ver a cada nodo como un cúmulo de recursos, dispuestos para ser utilizados por el cluster. por lo tanto, extrapolando conceptos, un cluster no deja de ser también una conjunción de recursos, así

- $R(I) = \{R(i_1) \wedge R(i_2) \wedge ... \wedge R(i_n)\} = \{\ \{r_c(i_1),\ r_m(i_1),\ ...\} \wedge \{r_c(i_2),\ r_m(i_2),\ ...\} \wedge ... \wedge \{r_c(i_n),\ r_m(i_n),\ ...\}\ \}$

Un nodo puede tener muchos recursos a la vez: procesador, memoria, targetas PCI, dispositivos de E/S, *etc.*. Sin embargo no es ahora tan importante saber cuantos recursos gestiona openMosix sino cómo lo hace. Esto es, ¿cómo se llega a la ponderación de cada nodo?

Antes de responder es importante apuntar lo que debiera haberse comprendido en la sección anterior: openMosix basa sus decisiones de migración en una algorítmica basada en modelos económicos, donde los recursos estan ponderados en una misma escala para simplificar los cálculos. Estos cálculos deben ser computables en el menor intervalo de tiempo posible para dotar al sistema de la máxima adaptabilidad a las condiciones cambiantes que los usuarios -los procesos en última instancia- puedan propiciar.

No se entrará -por el momento- en averiguar cómo se equiparan parámetros tan dispares como velocidad de procesamiento o cantidad de memoria. Lo relevante ahora es ver qué permite hacer esta asignación. Se supondrá, para cualquier nodo $i$:

- $r_c(i_k),\ c \in R(i_k) \equiv$ velocidad de procesador de $i_k$ ,
- $r_m(i_k),\ m \in R(i_k) \equiv$ tamaño de memoria de $i_k$ .

Igualmente interesa poder estimar los recursos que necesitarán las distintas tareas $j$, de cara a predecir el nodo donde se ejecutarían con más celeridad. Así se definen para las tareas cuatro parámetros.

- $a(j, i_k) \equiv$ tiempo de llegada de la tarea $j$ en el nodo $i_k$ ,
- $c(j, i_k) \equiv$ tiempo de procesador estimado necesario para la ejecución,
- $m(j, i_k) \equiv$ cantidad de memoria necesaria,
- $T(j, i_k) \equiv$ tiempo real invertido en la finalización de la ejecución.

Se asume, para una algorítmica *no ideal* -y por tanto suponiéndose implementable- que $m(j, i_k)$ y $c(j, i_k)$ son conocidos y $T(j, i_k)$ es desconocido en $a(j, i_k)$.

La finalidad de la definición de los parámetros que hasta ahora se han visto es poder decidir diversos aspectos que conciernen a la migración de procesos -en última instancia, el objetivo de openMosix-: decisiones sobre qué proceso migrar y dónde hacerlo se apoyan precisamente en los parámetros vistos. Esto es, para definir qué proceso deberá migrarse deben medirse los recursos -y la cantidad de ellos- que ocupará en el nodo que lo hospede. Igualmente para saber qué nodo es el mejor para aceptar dicho proceso deberá medirse la carga

computacional de un cierto número de ellos para saber cual dispone de mayor cantidad de ella y, en consecuencia, cual será capaz de retornar el resultado de proceso en el menor tiempo posible.

El poder computacional de un nodo queda definido no solamente por la ponderación de sus recursos, sino por la utilización de ellos tanto en el actual instante como en el futuro que venga marcado por la planificación de tal recurso.

Debería definirse la carga de un nodo. Esto es para un nodo $i_k$ en el instante $t$:

- $J(t, i_k) \in \mathbb{N} \equiv$ procesos en ejecución,
- $carga_c(t, i_k) \propto J(t, i_k) \equiv$ carga de procesador ($\in \mathbb{Z}^+$) ,
- $carga_m(t, i_k) = \sum_{j \in J(t,i_k)} m(j) \equiv$ carga de memoria ($\in \mathbb{Z}^+$).
- $LOAD = \sum [carga_c(t, i_k) + carga_m(t, i_k)]$ , $\forall\, i_k \in I \equiv$ es la carga total en el sistema en el instante $t$ .

Análogamente, la carga de un nodo se mide en función de los recursos de que ya no dispone. En este caso se han dado las expresiones para procesador y memoria.

No obstante para medir la carga de un nodo openMosix no utiliza solamente esta aproximación. Hay condiciones que pueden desajustar notablemente los valores para la carga calculada y la carga real, mayormente cuando el nodo hace *swapping*.

La implemenación de openMosix soluciona este apartado sirviéndose de un parámetro $s$, que se convertirá en un factor que multiplicará a la expresión de carga del nodo y que por tanto sobredimensionará el valor calculado tras el repaso de la ponderación de recursos. Esto tiene una completa lógica ya que con acceso a *swap* el procesador tiene que utilizar muchos más ciclos que trabajando con memoria, para recuperar una página.

Para formalizar esta idea puede escribirse:

- si $carga_m(t, i_k) \leq r_m(i_k) \Longrightarrow carga_c(t, i_k) = J(t, i_k)$ ,
- si $carga_m(t, i_k) > r_m(i_k) \Longrightarrow carga_c(t, i_k) = J(t, i_k) * s$ .

#### Objetivo de rendimiento de openMosix

El objetivo de los cálculos de openMosix es optimizar el aprovechamiento de la totalidad de los recursos del sistema. Así por ejemplo, para cada nuevo proceso iniciado no se estudia solamente el aumento de carga en el nodo local, sino que se tiene en cuenta el contexto de carga de todo el sistema. Cuando existen suficientes procesadores en el cluster para encargarse de su carga, cada proceso se servirá al procesador menos cargado y se devolverá la respuesta tan rápido como pueda hacerlo.

Si por el contrario no existen suficientes procesadores para encargarse de los procesos pendientes de ejecución, se evaluará el tiempo de ejecución para un nodo y para el resto de los nodos. De esta manera se llega a un completo estudio sobre el impacto del aumento de carga que éste propicia al sistema, para luego decidir cual es el mejor nodo de ejecución. Antes de ver esto se define:

- $N = \sum_{i_k \in I} r_c(i_k) \equiv$ es el número de unidades de procesamiento del cluster.
- $L = carga_c(t, i_k) \equiv$ es la carga de los elementos procesadores de cualquier nodo.

Se deduce que la carga total de todos los procesadores del cluster será $NL$ . Si la carga para cualquier nodo es $L$, la carga de los restantes es $N(L-1)$ . La carga :

- sin tener en cuenta el nuevo proceso se necesitarán $\frac{N(L-1)}{N}$ unidades de tiempo para completarse,
- con el nuevo proceso serían necesarias $\frac{NL}{N} = L$ unidades de tiempo.

La pérdida de tiempo comparativa para la ejecución de los $NL-1$ procesos en función de estas dos situaciones se puede expresar como:

$$L - \frac{1}{N} .$$

La algorítmica de openMosix intentará minimizar esta pérdida.

## Optimizando la planificación

Hasta el momento se ha estado viendo la modelización de openMosix para cada nodo, ahora sería conveniente fijar como objetivo el cálculo para el algoritmo que reconoce esta modelización y opera en consecuencia a ella. El punto de partida será el algoritmo óptimo. Esto permitirá que, tras la comparación con el que openMosix implementa, sea factible comparar resultados para ver cuánto se acerca el caso real, al ideal.

Pero la algorítmica utilizada depende enormementede la topología de los nodos el cluster, es decir, no supone el mismo esfuerzo computacional hacer los cálculos sobre efectos de carga para un conjunto de máquinas *iguales*[3] -donde la carga repercutirá numéricamente igual y por lo tanto solo será necesario hacer la predicción una vez- que para un conjunto de máquinas donde la carga supone una sobrecarga distina.

Esta situación debe tenerse en cuenta, y se modeliza de la siguiente forma:

- **Nodos idénticos:** todos y cada uno de los nodos del cluster poseerá todos y cada uno de los recursos de cualquier otro nodo. Esto implica que un proceso supondrá el mismo augmento de carga para cualquier nodo. Véase:

  $IDENTICOS = \{ [R(i_a), R(i_b)] \mid R(i_a) = R(i_b) \ \forall\, i_a, i_b \in I \ \wedge\ r_c(i_a) = r_c(i_b), r_m(i_a) = r_m(i_b) \} \Rightarrow\ c(j, i_a) = c(j, i_b)$ .

- **Nodos relacionados:** todos los nodos del cluster poseen los mismos recursos, aunque pueden diferir en su ponderación. Imagínese por ejemplo un cluster de nodos idénticos aunque con los procesadores a diferentes velocidades.

  $RELACIONADOS = \{ [R(i_a), R(i_b)] \mid R(i_a) = R(i_b) \ \forall\, i_a, i_b \in I \ \wedge\ r_c(i_a) \neq r_c(i_b) \vee r_m(i_a) \neq r_m(i_b) \} \Rightarrow c(j, i_a) \neq c(j, i_b)$ .

- **Nodos no relacionados:** los nodos tienen diferentes recursos.

  $NO\_RELACIONADOS = \{ [R(i_a), R(i_b)] \mid \exists\, R(i_a) \neq R(i_b) \ \forall\, i_a, i_b \in I \}$ .

Dadas estas diferentes situaciones podrá darse un algoritmo para resolver la planificación en cada una de ellas, y como se ha apuntado se medirá su ratio de competencia con el algoritmo óptimo. Por lo tanto van a compararse.

Un algoritmo que se ejecuta a la vez que los procesos -a tiempo real- es competitivo de grado $c$ si para una secuencia de entrada $I$, $algoritmo(I) \leq c * optimo(I) + \alpha$ , siendo $\alpha$ una constante definida y $optimo$ el algoritmo ideal que no se calcula.

Esta expresión toma diferentes valores para $c$ dependiendo del algoritmo utilizado para resolver la planificación, y dicha planificación depende de la topología del cluster que anteriormente ya se ha clasificado. Esta clasificación imprime diferencias a diferentes niveles. El más immediato, como se ha citado, es en la decidibilidad del algoritmo de migración. El grado de competición $c$ según la topología se define:

- Para un cluster con nodos idénticos se toma el *algoritmo egoista*, que asigna procesos a la máquina que tenga la menor carga en el momento de tomar la decisión de asignación de tal proceso. El ratio de competición frente al algoritmo óptimo para este algoritmo es $\rightarrow\ c = 2 - \frac{1}{n}$ .

- Para un cluster con nodos relacionados se ha implementado el *algoritmo Assign-R* el cual asigna cada trabajo que llega al nodo más lento. El coste de esta asignación es inferior al doble de la asignación hecha por el algoritmo óptimo, lo que da $\rightarrow\ c = 2$ .

- Para un cluster con nodos no relacionados no se conoce todavía un algoritmo con un ratio de competición mejor que $n$. Esto se debe a que en este caso hay que tener en cuenta muchas variables que en los demás casos podían obviarse.

  Existe no obstante el *algoritmo Assign-U* pensado especialmente para máquinas que no tienen relación entre sí para trabajos que se mantienen en un mismo nodo, basado en una función exponencial para el coste de un nodo en particular para una carga dada. Este algoritmo asigna cada trabajo a uno de los nodos para minimizar el coste total de todos los nodos del cluster. Este algoritmo es:

  $U_i(j) = \alpha^{carga_i(j)+p_i(j)} - \alpha^{carga_i(j)}$ , donde

[3] Refiriéndose a una misma arquitectura y una ponderación de los recursos equivalente.

- $1 \leq \alpha \leq 2$ . En el caso concreto de openMosix $\alpha = 2$.
- $carga_i(j)$ es la carga que tiene un nodo $i$ antes de que le sea asignado un trabajo $j$ .
- $p_i(j)$ es la carga que un trabajo $j$ añadirá a un nodo $i$ .

Como se puede ver, la formula favorece a los nodos que tienen cargas menores. Por ejemplo supongamos que tenemos 2 nodos, uno de ellos con una carga 5 y otro nodo con una carga 10. En esta configuración hemos decidido que el parámetro $\alpha$ tenga un valor de 2 (para simplificar nuestros cálculos). Se inicia un nuevo trabajo que añade una carga de 1. Para nuestros dos nodos tenemos:

$U_1(j) = 2^{5+1} - 2^5 = 64 - 32 = 32$ ,
$U_2(j) = 2^{10+1} - 2^{10} = 2048 - 1024 = 1024$ .

Claramente el primer valor que corresponde al nodo con menor carga (5) minimiza la carga del sistema por lo tanto se elegirá el primer nodo como nodo destinatario del nuevo proceso. Las cargas de los nodos se ponderan en potencias de base 2 -gracias a $\alpha$- y esto permite agrupar los nodos con cargas iguales. La resolución del nodo con menor carga final puede calcularse a aprtir de un árbol binario. Este algoritmo es competitivo de grado $O(\log_2 n)$, para máquinas no relacionadas y trabajos permanentes.

Se puede extender este algoritmo y el ratio de competitividad a los trabajos que migran usando como máximo $O(\log_2 n)$ migraciones por trabajo. Para este nuevo tipo de situación necesitamos un parámetro más $h_i(j)$ que es la carga de un nodo $i$ justo antes de que $j$ haya sido por última vez asignado a $i$. Cuando un trabajo ha finalizado, este algoritmo comprueba la estabilidad del sistema para cada trabajo $j$ en cada nodo $M$. Esta es una condición de estabilidad que siendo $i$ el nodo donde el trabajo $j$ está actualmente, se definde como:

$$\alpha^{h_i(j)+p_i(j)} - a^{h_i(j)} \leq 2 * (\alpha^{carga_M(j)+P_M(j)} - \alpha^{carga_M(j)})$$

Si esta condición no es satisfecha por algún trabajo, el algoritmo reasigna el trabajo que no satisface la condición a un nodo $M$ que minimiza $U_M(j)$ .

Esta fórmula es quizás más omplicada de ver que las demás. Se detallan seguidamente sus factores:

Entre las reflexiones que pueden sacarse de este estudio se encuentra la explicación a la clara tendencia a construir clusters con nodos idénticos o con las mínimas diferencias entre nodos. El rendimiento será mucho más generoso.

## Modelo simplificado de ponderación

Una vez formalizados los recursos, los nodos y su relación dentro del mecanismo que pone en marcha el cluster para ser capaz de realizar sus decisiones, solo falta ver la ponderación, es decir, el valor numérico que todo ello representa. La modelización que gestiona openMosix para trabajar con los elementos del cluster es un grafo poderado $G(A, V, P)$, donde:

- $A$: conjunto de arcos $a$ que se corresponden al medio de transporte de datos,
- $V$: conjunto de vértices -nodos-,
- $P$: función de incidencia del grafo, definida en $A \rightarrow V$, -se ve más adelante-.

Se define también una secuencia de llegada de peticiones, en tiempos arbitrarios, que como puede preverse son los procesos que llegan a un nodo -localmente por el usuario o desde un remoto- para ser ejecutados. Véase cada elemento con más detalle.

A cada arco -físicamente un enlace de red- se le asocia cierta ponderación que irá en aumento cada vez que se genere una comunicación sirviéndose de él. Esta situación se produce cada vez que un proceso crea un nexo entre su *remote* y su *deputy*, puesto que ambas partes deben permanecer continuamente comunicadas. Al referirse a un canal real de comunicación, es necesario asociarles una capacidad -ancho de banda-, sea $B(a)$.

La petición producida por la tarea $j$ que sea asignada a un camino -conjunto de arcos por los que circulará la petición- desde una fuente $i_s$ a un destino $i_d$ disminuye la capacidad $B(a)$ en cada uno de los arcos que pertenecen a este camino en una cantidad $carga_a(j)$. El objetivo es **minimizar la congestión** $C(a)$ máxima de cada uno de los enlaces, que es el ratio entre el ancho de banda requerido de un enlace y su capacidad:

$$C(a) = \frac{carga_a(j)}{B(a)} \equiv \text{congestión de la arista } a \text{ provocada por la tarea } j$$

La solución viene dada por la minimización del máximo uso de procesador y memoria. Tras ello puede ser reducida a un algoritmo -en tiempo de ejecución- del problema de encaminamiento de los circuitos virtuales. Esta reducción funciona como sigue:

1. Se toman dos nodos, $i_s$ e $i_d$, y $n$ caminos sin tramos superpuestos que tengan como origen $i_s$ y como destino el nodo $i_d$. Se considerarán únicamente los caminos que consten solamente de 2 arcos.

2. El coste desde el nodo $i_s$ hasta el $i_d$ será representado por uno de estos caminos. Los arcos se dividirán ponderando al primero de ellos con la capacidad de memoria de $i_s$, es decir $r_m(i_s)$. Al segundo de ellos se le ponderará con $r_c(i_s)$.

Esta política acota la congestión máxima del enlace al valor máximo entre la máxima carga de procesador y el máximo uso de la memoria. De esta manera se extiende para solucionar el problema del encaminamiento.

Este algoritmo es competitivo de grado $O(\log_2 n)$. Por reducción, produce un algoritmo para manejar recursos heterogéneos que es competitivo también de grado $O(\log_2 n)$ en su máximo uso de cada recurso. Queda claro que cuando se escoge un camino se está realmente eligiendo un nodo y que las diferencias entre los dos problemas se solucionan por la función que mapea la carga de los enlaces de uno de los problemas con la carga de los recursos del otro problema. Esto hace que la forma de extender sea muy simple porque lo único que es necesario para tener en cuenta un recurso más en el sistema es añadir un arco, además de una función que mapee el problema de ese recurso en particular en el problema de encaminamiento de circuitos virtuales.

La **ponderación para un nodo** $i_k$, poseedor de $m$ recursos consiste en una expresión que relaciona el número total de nodos $n$ del cluster con la utilización de cada recurso. La elección de los parámetros parece correcta -ha sido elegida por los desarrolladores- puesto que la parte de potencia que un nodo significa en el seno de un clsuter depende del número de nodos existentes. De igual forma un nodo potente no servirá de mucho si tiene todos sus recursos utilizados.
La expresión que se indica tendría la forma:

$$\sum_{i=1}^{k} f(n, u_{r_i}) \mid r_i \in R(i)\,, \qquad \text{donde } u_{r_i} \text{ marca la utilización del recurso } r_i\,.$$

Particularmente esta función de relación entre los parámetros indicados tiene la forma:

$$f(n, u_{r_i}) = \sum_{i=1}^{k} n^{\frac{u_{r_i}}{max[u_{r_i}]}}, \qquad \text{donde se relaciona para cada recurso su uso actual con su uso máximo}^4.$$

NOTA: Tras estudios experimentales se ha probado que el algoritmo consegue una máxima pérdida de rendimiento dentro de $O(\log_2 n)$ de la máxima pérdida de rendimiento del algoritmo óptimo.

Esta relación entre los usos de memoria funciona tal cual para el recurso de memoria. El coste por una cierta cantidad de uso de memoria en un nodo es proporcional a la utilización de memoria -relación memoria usada/memoria total-.
Para el recurso procesador se debería saber cuál es la máxima carga posible, esto no es tan sencillo como con la memoria y va a ser necesario tener que hacer algún manejo matemático. Se asume que $L$ es el entero más pequeño en potencia de dos, mayor que la mayor carga que se haya visto hasta el momento, y que este número es la carga máxima posible. Aunque esta aproximación no sea realmente cierta sirve perfectamente para resolver el problema.
Tras estas aclaraciones las particularizaciones de ponderación para el tipo de recurso procesador y memoria quedaría respectivamente:

$$\text{memoria} \rightarrow n^{\frac{u_{r_m(i)}}{r_m(i)}} \qquad \text{y procesador} \rightarrow n^{\frac{u_{r_c(i)}}{L}}\,.$$

[4] Valor éste dado por la capacidad del recurso o por algún valor de acotación puesto para evitar la completa apropiación del mismo.

## 6.3. ./arch/*

The arch subdirectory contains all of the architecture specific kernel code. It has further subdirectories, one per supported architecture.

```
patching file arch/i386/config.in
patching file arch/i386/defconfig
patching file arch/i386/kernel/entry.S
patching file arch/i386/kernel/i387.c
patching file arch/i386/kernel/ioport.c
patching file arch/i386/kernel/Makefile
patching file arch/i386/kernel/offset.c
patching file arch/i386/kernel/process.c
patching file arch/i386/kernel/ptrace.c
patching file arch/i386/kernel/signal.c
patching file arch/i386/kernel/sys_i386.c
patching file arch/i386/kernel/traps.c
patching file arch/i386/kernel/vm86.c
patching file arch/i386/lib/usercopy.c
patching file arch/i386/mm/fault.c
patching file arch/ia64/kernel/entry.S
```

### 6.3.1. *config.in*

Se configura el menú de openMosix que se mostrarán en el *front-end* de configuración, invocado por ejemplo por el comando `make menuconfig`. Véase el código:

```
comment 'openMosix'
bool 'openMosix process migration support' CONFIG_MOSIX
if [ "$CONFIG_MOSIX" = "y" ]; then
        bool 'Support clusters with a complex network topology' CONFIG_MOSIX_TOPOLOGY
        if [ "$CONFIG_MOSIX_TOPOLOGY" = "y" ]; then
            int  'Maximum network-topology complexity to support (2-10)' CONFIG_MOSIX_MAXTOPOLOGY 4
        fi

        bool 'Stricter security on openMosix ports' CONFIG_MOSIX_SECUREPORTS
        int  'Level of process-identity disclosure (0-3)' CONFIG_MOSIX_DISCLOSURE 1
        #bool 'Create the kernel with a "-openmosix" extension' CONFIG_MOSIX_EXTMOSIX
        bool 'openMosix File-System' CONFIG_MOSIX_FS
        if [ "$CONFIG_MOSIX_FS" = "y" ]; then
           define_bool CONFIG_MOSIX_DFSA y
        fi
        bool 'Poll/Select exceptions on pipes' CONFIG_MOSIX_PIPE_EXCEPTIONS
        bool 'Disable OOM Killer' CONFIG_openMosix_NO_OOM
        bool 'Load Limit' CONFIG_MOSIX_LOADLIMIT
fi
endmenu
```

### 6.3.2. *defconfig*

Se guardarán todas las opciones del kernel seleccionadas por el usuario.

### 6.3.3. *entry.S*

Este fichero contiene todas las rutinas que manejan las llamadas a sistema. También implementa el gestor de interrupciones para los cambios de contexto en un *switch* de procesos.

```
#ifdef CONFIG_MOSIX
#include "mosasm.H"                    /*  libreria de codigo ensamblador de openMosix */
#endif /* CONFIG_MOSIX */
```

Para pasar de modo sistema a modo usuario:

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
#ifdef CONFIG_MOSIX
        testl $(DTRACESYS1|DTRACESYS2),DFLAGS(%ebx)
        jne adjust_trace_before_syscall
adjusted_trace:
        testb $DREMOTE,DFLAGS(%ebx)
        je local_syscall
on_remote:
```

```
        pushl %eax
        call *SYMBOL_NAME(remote_sys_call_table)(,%eax,4)
        addl $4,%esp
        movl %eax,EAX(%esp)
        jmp ret_from_sys_call
local_syscall:
#endif /* CONFIG_MOSIX */
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
#ifdef CONFIG_MOSIX
        call SYMBOL_NAME(mosix_local_syscall)
#endif /* CONFIG_MOSIX */
ENTRY(ret_from_sys_call)
#ifdef CONFIG_MOSIX
        testl $(DTRACESYS1|DTRACESYS2),DFLAGS(%ebx)
        jne adjust_trace_before_syscall
ret_check_reschedule:
#endif /* CONFIG_MOSIX */
        cli                             # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
#ifdef CONFIG_MOSIX
straight_to_mosix:
        call SYMBOL_NAME(mosix_pre_usermode_actions)
        testl %eax,%eax
        jne ret_from_sys_call
#endif /* CONFIG_MOSIX */
restore_all:
        RESTORE_ALL

        ALIGN
signal_return:
        sti                             # we can get here from an interrupt handler
        testl $(VM_MASK),EFLAGS(%esp)
        movl %esp,%eax
        jne v86_signal_return
        xorl %edx,%edx
        call SYMBOL_NAME(do_signal)
#ifdef CONFIG_MOSIX
        jmp straight_to_mosix
#else
        jmp restore_all
#endif /* CONFIG_MOSIX */

        ALIGN
v86_signal_return:
        call SYMBOL_NAME(save_v86_state)
        movl %eax,%esp
        xorl %edx,%edx
        call SYMBOL_NAME(do_signal)
#ifdef CONFIG_MOSIX
        jmp straight_to_mosix
#else
```

```
        jmp restore_all
#endif /* CONFIG_MOSIX */

        ALIGN
tracesys:
        movl $-ENOSYS,EAX(%esp)
        call SYMBOL_NAME(syscall_trace)
#ifdef CONFIG_MOSIX
adjust_trace_before_syscall:            # only arrive here with DTRACESYS(1|2)
        testl $DDEPUTY,DFLAGS(%ebx)
        jne straight_to_mosix           # no mess with signals/syscalls/tracesys
        testl $DREMOTE,DFLAGS(%ebx)
        je no_need_to_unsync
        call wait_for_permission_to_continue
no_need_to_unsync:
        testl $DTRACESYS2,DFLAGS(%ebx)
        jne second_tracesys             # skipping system-call
        orl $DTRACESYS2,DFLAGS(%ebx)    # next time we skip the system-call
        movl $-ENOSYS,EAX(%esp)
        movl ORIG_EAX(%esp),%eax
        cmpl $(NR_syscalls),%eax
        jae second_tracesys             # prevent system-call out of range trick
        jmp adjusted_trace              # now do the system-call
second_tracesys:                        # note: "syscall_trace" clears the flags
#else
        movl ORIG_EAX(%esp),%eax
        cmpl $(NR_syscalls),%eax
        jae tracesys_exit
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
tracesys_exit:
#endif /* CONFIG_MOSIX */
        call SYMBOL_NAME(syscall_trace)
        jmp ret_from_sys_call
badsys:
        movl $-ENOSYS,EAX(%esp)
        jmp ret_from_sys_call

        ALIGN
ENTRY(ret_from_intr)
        GET_CURRENT(%ebx)
ret_from_exception:
        movl EFLAGS(%esp),%eax          # mix EFLAGS and CS
        movb CS(%esp),%al
        testl $(VM_MASK | 3),%eax       # return to VM86 mode or non-supervisor?
#ifdef CONFIG_MOSIX
        jne ret_check_reschedule
#else
        jne ret_from_sys_call
#endif /* CONFIG_MOSIX */
        jmp restore_all

        ALIGN
reschedule:
        call SYMBOL_NAME(schedule)    # test
```

```
        jmp ret_from_sys_call
```

### 6.3.4. *i387.c*

Guarda el contexto de los registros eprtenecientes al coprocesador i387.

### 6.3.5. *ioport.c*

Este fichero gestiona un mapa de bits que corresponden a los permisos que posee un proceso sobre los dispositivos de E/S.

`static void set_bitmap()`

Configura la máscara de permisos, esto no difiere del codigo del kernel *vanilla* de Linux. Al mapa *bitmap* se guarda el valor *turn_on*.

```
static void set_bitmap(unsigned long *bitmap, short base, short extent, int new_value)
{
        int mask;
        unsigned long *bitmap_base = bitmap + (base >> 5);
        unsigned short low_index = base & 0x1f;
        int length = low_index + extent;

        if (low_index != 0) {
                mask = (~0 << low_index);
                if (length < 32)
                                mask &= ~(~0 << length);
                if (new_value)
                        *bitmap_base++ |= mask;
                else
                        *bitmap_base++ &= ~mask;
                length -= 32;
        }

        mask = (new_value ? ~0 : 0);
        while (length >= 32) {
                *bitmap_base++ = mask;
                length -= 32;
        }

        if (length > 0) {
                mask = ~(~0 << length);
                if (new_value)
                        *bitmap_base++ |= mask;
                else
                        *bitmap_base++ &= ~mask;
        }
}
```

`asmlinkage int sys_ioperm()`

Esta función modifica los permisos de E/S para el proceso en curso. Se invocará cuando un proceso deba dejar de mapear memoria del dispositivo de E/S o, en el caso de openMosix, cuando un proceso migrado quiera acceder a un dispositivo que se encuentre local al UHN.

```
asmlinkage int sys_ioperm(unsigned long from, unsigned long num, int turn_on)
{
        struct thread_struct * t = &current->thread;            /*proceso en curso */
        struct tss_struct * tss = init_tss + smp_processor_id();

        if ((from + num <= from) || (from + num > IO_BITMAP_SIZE*32))
                return -EINVAL;
        if (turn_on && !capable(CAP_SYS_RAWIO))
                return -EPERM;
#ifdef CONFIG_MOSIX                                              /* si openmosix debe devolver el
        if(turn_on && !mosix_go_home_for_reason(1, DSTAY_FOR_IOPL))
                return(-ENOMEM);                                 /* proceso a su UHN por razones d
#endif /* CONFIG_MOSIX */
        /* Si el proceso aun no habia invocado a ioperm() se carga el mapa en memoria */
        if (!t->ioperm) {
                memset(t->io_bitmap,0xff,(IO_BITMAP_SIZE+1)*4);
                t->ioperm = 1;
        }

        /* se deberan copiar las modificaciones a los threads y TSS */
        /* TSS -Task State Segment. Es un segmento especifico en arquitecturas
        x86 para guardar contextos hardware */
        set_bitmap(t->io_bitmap, from, num, !turn_on);
        if (tss->bitmap == IO_BITMAP_OFFSET) {
                set_bitmap(tss->io_bitmap, from, num, !turn_on);
        } else {
                memcpy(tss->io_bitmap, t->io_bitmap, IO_BITMAP_BYTES);
                tss->bitmap = IO_BITMAP_OFFSET; /* Activa el mapa modificado en el TSS */
        }

        return 0;
}
```

### 6.3.6. *offset.c*

Este fichero proporciona rutinas a las librerrías utilizadas en `entry.S` para las operaciones específicas de openMosix. Como:

- Desplazamientos -*offsets*- constantes para los registros contenidos en la estructura *task_struct*.
- Bits para tests de `current->dflags`
- Una copia de la tabla de llamadas a sistema -`remote_sys_call_table`- con todas las llamadas a sistema precedidas por el prefijo *remote_*.

### 6.3.7. *ptrace.c*

Configura y gestiona los registros SSE i FXSR de las arquitecturas Intel x86, a partir del P III-. SSE es el acrónimo de *Streaming SIMD Extensions* y mejora el rendimiento de la arquitectura Intel en 4 vesantes:

- 8 nuevos registros de punto flotante de tamaño 128bit con direccionamiento directo
- 50 nuevas instrucciones para trabajar con paquetes de datos de punto flotante
- 8 nuevas instrucciones para el control de la cache para los datos MMX
- 12 nuevas instrucciones de extensión de MMX.

### 6.3.8. *signal.c*

En este fichero se maneja todo lo referente a las señales qe el sistema puede enviar a los procesos. El parche openMosix debe hacer considerables modificaciones, puesto que en los procesos migrados -procesos divididos en *remote* y *deputy*- la comunicación también debe ser posible, además de transparente. Igualmente tiene que ser posible hacer llegar a *remote* el señal para informarle de su vuelta, *etc.*

`int copy_siginfo_to_user()`

Para que el área de usuario esté informado del estado de los señales, controlado por el área de kernel, se le debe pasar la información sobre el mapa de interrupciones.

En procesos migrados esto es importante, puesto que supone una comunicación entre *remote* y *deputy*.

```
int copy_siginfo_to_user(siginfo_t *to, siginfo_t *from)
{
        if (!access_ok (VERIFY_WRITE, to, sizeof(siginfo_t)))
                return -EFAULT;
        if (from->si_code < 0)
                return __copy_to_user(to, from, sizeof(siginfo_t));
        else {
#ifdef CONFIG_MOSIX
                int sz = offsetof(struct siginfo, _sifields);

                switch(from->si_code >> 16)
                {
                        case __SI_FAULT >> 16:
                                sz += sizeof(to->_sifields._sigfault);
                                break;
                        case __SI_CHLD >> 16:
                                sz += sizeof(to->_sifields._sigchld);
                                break;
                        case __SI_MIGRATION >> 16:
                                sz += sizeof(to->_sifields._sigmig);
                                break;
                        default:
                                sz += sizeof(to->_sifields._kill);
                                break;
                }
                return(__copy_to_user(to, from, sz));
#else
                int err;

                err = __put_user(from->si_signo, &to->si_signo);
                err |= __put_user(from->si_errno, &to->si_errno);
                err |= __put_user((short)from->si_code, &to->si_code);
                /* First 32bits of unions are always present.  */
                err |= __put_user(from->si_pid, &to->si_pid);
                switch (from->si_code >> 16) {
                case __SI_FAULT >> 16:
                        break;
                case __SI_CHLD >> 16:
                        err |= __put_user(from->si_utime, &to->si_utime);
                        err |= __put_user(from->si_stime, &to->si_stime);
                        err |= __put_user(from->si_status, &to->si_status);
                default:
                        err |= __put_user(from->si_uid, &to->si_uid);
```

```
                        break;
                }
                return err;
#endif /* CONFIG_MOSIX */
        }
}
```

`asmlinkage int sys_sigreturn()`

Envía el señal para mandar de regreso el *remote*.

```
asmlinkage int sys_sigreturn(unsigned long __unused)
{
        struct pt_regs *regs = (struct pt_regs *) &__unused;
#ifdef CONFIG_MOSIX
        struct sigframe *frame;
#else
        struct sigframe *frame = (struct sigframe *)(regs->esp - 8);
#endif /* CONFIG_MOSIX */
        sigset_t set;
        int eax;

#ifdef CONFIG_MOSIX
        mosix_obtain_registers(BIT_OF_REGISTER(esp));
        frame = (struct sigframe *)(regs->esp - 8);
#endif /* CONFIG_MOSIX */
```

La estructura `sigframe` contiene:

```
    char *pretcode;
int sig;
struct sigcontext sc;
struct _fpstate fpstate;
unsigned long extramask[_NSIG_WORDS-1];
char retcode[8];
```

```
        if (verify_area(VERIFY_READ, frame, sizeof(*frame)))   /* verifica la correcta lectura al r
                goto badframe;
        if (__get_user(set.sig[0], &frame->sc.oldmask)
            || (_NSIG_WORDS > 1
                && __copy_from_user(&set.sig[1], &frame->extramask,
                                    sizeof(frame->extramask))))
       /* con una correcta comunicacion, copia desde el remoto al UHN */
                goto badframe;

        sigdelsetmask(&set, ~_BLOCKABLE);
        spin_lock_irq(&current->sigmask_lock);   /* bloquea la seccion critica con semaforos */
        current->blocked = set;                  /* aplica la mascara de los registros al proceso e
                                                 /* los registros bloquean el proceso, para recibir
        recalc_sigpending(current);              /* desbloquea la proteccion */
        spin_unlock_irq(&current->sigmask_lock);

#ifdef CONFIG_MOSIX
        if(current->mosix.dflags & DDEPUTY)
        {
                if (mosix_deputy_restore_sigcontext(&frame->sc, &eax))
```

```
                /* restaura el contexto de los registros en el deputy */
                        goto badframe;
        }  /* si no se recupera bien el registro eax */
        else
#endif /* CONFIG_MOSIX */
        if (restore_sigcontext(regs, &frame->sc, &eax))
                goto badframe;
        return eax;

badframe:
        force_sig(SIGSEGV, current);
        /* en caso de que no se proceda satisfactoriamente, se envia SIGSEGV para abortar*/
        return 0;
}
```

`asmlinkage int handle_signal()`

Esta rutina atiende a las señales que recibe un proceso.

```
static void
handle_signal(unsigned long sig, struct k_sigaction *ka,
              siginfo_t *info, sigset_t *oldset, struct pt_regs * regs)
{
#ifdef CONFIG_MOSIX
        mosix_obtain_registers(
           BIT_OF_REGISTER(orig_eax)|BIT_OF_REGISTER(eax)|BIT_OF_REGISTER(eip));
#endif /* CONFIG_MOSIX */
        /* si se esta ejecutando una llamada a sistema, como es una se˜nal */
        if (regs->orig_eax >= 0) {
                /* en este caso se informa al proceso, a traves del registro eax */
                switch (regs->eax) {
                        case -ERESTARTNOHAND:
                                regs->eax = -EINTR;
                                break;

                        case -ERESTARTSYS:
                                if (!(ka->sa.sa_flags & SA_RESTART)) {
                                        regs->eax = -EINTR;
                                        break;
                                }
                        /* fallthrough */
                        case -ERESTARTNOINTR:
                                regs->eax = regs->orig_eax;
                                regs->eip -= 2;
                }
        }

        /* Set up the stack frame */
#ifdef CONFIG_MOSIX
   /* si el proceso no es un proceso Linux completo, sino un deputy, se lo
trata como debe hacerse, mediante una llamada a mosix_deputy_setup_frame().
La mayor diferencia es que el deputy carece de segmento de datos para
almacenar esta informacion*/
        if(current->mosix.dflags & DDEPUTY)
                mosix_deputy_setup_frame(sig, ka, *info, oldset);
```

```
        else
#endif /* CONFIG_MOSIX */
        if (ka->sa.sa_flags & SA_SIGINFO)
                setup_rt_frame(sig, ka, info, oldset, regs);
        else
                setup_frame(sig, ka, oldset, regs);

        if (ka->sa.sa_flags & SA_ONESHOT)
                ka->sa.sa_handler = SIG_DFL;

        if (!(ka->sa.sa_flags & SA_NODEFER)) {
                spin_lock_irq(&current->sigmask_lock);   /* se bloquea la recepcion de interrupcior
                sigorsets(&current->blocked,&current->blocked,&ka->sa.sa_mask);
                sigaddset(&current->blocked,sig);        /* se actualiza el estado de las interrupc
                recalc_sigpending(current);              /* se atiende a las interrupciones pendier
                spin_unlock_irq(&current->sigmask_lock); /* se desbloquea */
        }
}
```

### 6.3.9. *vm86.c*

Se gestiona la ejecución de los procesos en emulación vm86. La mayor aportación de código openMosix en esta sección se debe a que aún no se permite la ejecución de tales procesos estando migrados, es un aspecto que debe controlarse.

```
asmlinkage int save_v86_state()
```

Guarda el estado del proceso en emulación. Se le indica la opción de bloqueo para que PPM no intente migrarlo.

```
struct pt_regs * save_v86_state(struct kernel_vm86_regs * regs)
{
        struct tss_struct *tss;
        struct pt_regs *ret;
        unsigned long tmp;

#ifdef CONFIG_MOSIX
        if(current->mosix.dflags & DREMOTE)
                panic("remote save_v86");
#endif /* CONFIG_MOSIX */
        if (!current->thread.vm86_info) {
                printk("no vm86_info: this is BAD\n");
                do_exit(SIGSEGV);
        }
        set_flags(regs->eflags, VEFLAGS, VIF_MASK | current->thread.v86mask);
        tmp = copy_to_user(&current->thread.vm86_info->regs,regs, VM86_REGS_SIZE1);
        tmp += copy_to_user(&current->thread.vm86_info->regs.VM86_REGS_PART2,
                &regs->VM86_REGS_PART2, VM86_REGS_SIZE2);
        tmp += put_user(current->thread.screen_bitmap,&current->thread.vm86_info->screen_bitmap);
        if (tmp) {
                printk("vm86: could not access userspace vm86_info\n");
                do_exit(SIGSEGV);
        }
        tss = init_tss + smp_processor_id();
#ifdef CONFIG_MOSIX
```

```
        lock_mosix();   /* ptrace checks saved_esp0 under the mosix-lock */
#endif /* CONFIG_MOSIX */
        tss->esp0 = current->thread.esp0 = current->thread.saved_esp0;
        current->thread.saved_esp0 = 0;
        ret = KVM86->regs32;
#ifdef CONFIG_MOSIX
        unlock_mosix();
        task_lock(current);
        current->mosix.stay &= ~DSTAY_FOR_86;    /* se marca la opcion de bloqueo del proceso */
        task_unlock(current);
#endif /* CONFIG_MOSIX */
        return ret;
}
```

`asmlinkage int sys_vm86old()`

En caso de tratarse de un proceso en emulación y de haberse iniciado la migración antes de percatarse de tal condición, se manda el señal para volver el proceso al UHN indicando la razón DSTAY_FOR_86.

```
asmlinkage int sys_vm86old(struct vm86_struct * v86)
{
        struct kernel_vm86_struct info;
        struct task_struct *tsk;
        int tmp, ret = -EPERM;

#ifdef CONFIG_MOSIX
        if(!mosix_go_home_for_reason(1, DSTAY_FOR_86))
        {
                ret = -ENOMEM;
                goto out;
        }
#endif /* CONFIG_MOSIX */
        tsk = current;
        if (tsk->thread.saved_esp0)
                goto out;
        tmp  = copy_from_user(&info, v86, VM86_REGS_SIZE1);
        tmp += copy_from_user(&info.regs.VM86_REGS_PART2, &v86->regs.VM86_REGS_PART2,
                (long)&info.vm86plus - (long)&info.regs.VM86_REGS_PART2);
        ret = -EFAULT;
        if (tmp)
                goto out;
        memset(&info.vm86plus, 0, (int)&info.regs32 - (int)&info.vm86plus);
        info.regs32 = (struct pt_regs *) &v86;
        tsk->thread.vm86_info = v86;
        do_sys_vm86(&info, tsk);
        ret = 0;        /* we never return here */
out:
        return ret;
}
```

`asmlinkage int sys_ioperm()`

## 6.4. ./Documentation/*

The Documentation subdirectory contains ascii files for support.

```
patching file Documentation/Configure.help
patching file Documentation/DFSA
patching file Documentation/filesystems/00-INDEX
patching file Documentation/filesystems/mfs.txt
patching file Documentation/sysctl/vm.txt
patching file Documentation/vm/overcommit-accounting
```

## 6.5. ./drivers/*

The Drivers subdirectory contains

```
patching file drivers/char/console.c
patching file drivers/char/defkeymap.c
patching file drivers/char/drm/i810_dma.c
patching file drivers/char/mem.c
patching file drivers/char/pc_keyb.c
patching file drivers/char/serial.c
patching file drivers/char/sysrq.c
patching file drivers/char/tty_io.c
patching file drivers/char/vt.c
patching file drivers/scsi/cpqfcTSworker.c
patching file drivers/sgi/char/graphics.c
patching file drivers/sgi/char/shmiq.c
patching file drivers/usb/storage/usb.c
```

## 6.6. ./fs/*

All of the file system code.
This is further sub-divided into directories, one per supported file system,
for example vfat and ext2.

```
patching file fs/binfmt_aout.c
patching file fs/binfmt_elf.c
patching file fs/buffer.c
patching file fs/dcache.c
patching file fs/dnotify.c
patching file fs/exec.c
patching file fs/ext3/super.c
patching file fs/fcntl.c
patching file fs/file.c
patching file fs/file_table.c
patching file fs/inode.c
patching file fs/ioctl.c
patching file fs/lockd/svc.c
patching file fs/Makefile
patching file fs/mfs/ccontact.c
patching file fs/mfs/client.c
patching file fs/mfs/complete.c
patching file fs/mfs/convert.c
patching file fs/mfs/count.c
patching file fs/mfs/file.c
patching file fs/mfs/Makefile
patching file fs/mfs/scontact.c
patching file fs/mfs/server.c
patching file fs/mfs/socket.c
patching file fs/namei.c
patching file fs/namespace.c
patching file fs/nfsd/auth.c
patching file fs/nfsd/export.c
patching file fs/nfsd/vfs.c
patching file fs/open.c
patching file fs/pipe.c
patching file fs/proc/array.c
patching file fs/proc/base.c
patching file fs/proc/generic.c
patching file fs/proc/inode.c
patching file fs/proc/proc_misc.c
patching file fs/proc/root.c
patching file fs/readdir.c
patching file fs/read_write.c
patching file fs/stat.c
patching file fs/super.c
patching file fs/umsdos/ioctl.c
```

## 6.7. ./hpc/*

```
Some modules. This subdirectory is created by openMosix patch.
Has migration, load balancing and memory ushering algorithms.

patching file hpc/badops.c
patching file hpc/balance.c
patching file hpc/comm.c
patching file hpc/config.c
patching file hpc/copy_unconf
patching file hpc/decay.c
patching file hpc/deputy.c
patching file hpc/dfsa.c
patching file hpc/div.c
patching file hpc/export.c
patching file hpc/freemem.c
patching file hpc/hpcadmin.c
patching file hpc/hpcproc.c
patching file hpc/info.c
patching file hpc/init.c
patching file hpc/kernel.c
patching file hpc/load.c
patching file hpc/Makefile
patching file hpc/mig.c
patching file hpc/mkdefcalls.c
patching file hpc/prequest.c
patching file hpc/remote.c
patching file hpc/rinode.c
patching file hpc/service.c
patching file hpc/syscalls.c
patching file hpc/ucache.c
```

### 6.7.1. *badops.c*

Contiene 228 líneas de funciones con el solo contenido de retornar un código de error. Estas funciones se invocan al no completarse satisfactoriamente cualquier operación de openMosix y el valor de retorno puede servir para ayudar localizar tal error.

### 6.7.2. *balance.c*

En este fichero se implementa la parte de balanceo de recursos de nuestro cluster.

Para resetear las estadisticas acumuladas se eliminan los valores anteriores del `struct mosix_task` tras apuntarlo a la tarea actual que se está tratando. `mosix_clear_statistics` lo implementa así :

`void mosix_clear_statistics()`

```
void
mosix_clear_statistics(void)
{
        register struct mosix_task *m = &current->mosix;

        m->ndemandpages = 0;
        m->nsyscalls = 0;
        m->ncopyouts = 0;
        m->copyoutbytes = 0;
        m->ncopyins = 0;
        m->copyinbytes = 0;
        m->iocounter = 0;
        m->cutime = 0;
        m->dctime = 0;
        m->pagetime = 0;
        read_lock(&tasklist_lock);
        m->decsecs = 0;
        read_unlock(&tasklist_lock);
        spin_lock_irq(&whereto_lock);
        m->last_consider = 0;
        m->last_mconsider = time_now();
        spin_unlock_irq(&whereto_lock);
        if(m->dflags & (DREMOTE|DDEPUTY))
                m->uttime = 0;
        else
                m->uttime = -current->times.tms_utime;
#ifdef CONFIG_MOSIX_DFSA
        m->copy_ins = 0;
        m->bytes_in = 0;
#endif /* CONFIG_MOSIX_DFSA */
#ifdef CONFIG_MOSIX_FS
        if(m->mfs_stats)
                m->mfs_stats->nnodes = 0;
#endif /* CONFIG_MOSIX_FS */
}
```

`inline int cpuload()`

Esta función determina la pérdida de eficiencia en la ejecución de un nodo. La base teórica puede recuperarse de la modelización matemática, cuando se habla de la ponderación de los nodos.

```
#define INFLOAD 0x8000000
#ifdef CONFIG_MOSIX_LOADLIMIT
unsigned long load_limit = 0;
unsigned long cpu_limit = 0;
inline int
altload(int load, unsigned long load_remote, unsigned long load_local,
        unsigned long cpu_remote, unsigned long cpu_local,
        unsigned long speed, int ncpus,
        unsigned long load_limit,
        unsigned long llimitmode,
        unsigned long cpu_limit,
        unsigned long cpulimit_mode
        )
```

Si se da el caso que se ha definido la variable CONFIG_MOSIX_LOADLIMIT se llamará a la función con los parámetros de configuración, sean estos el límite de uso del procesador, la carga máxima del nodo, *etc.*

```
#else
inline int
altload(int load, unsigned long speed, int ncpus)
```

Si no se han definido limitaciones para la apropiación de recursos podrá llamarse la función para que se adapte a las condiciones del sistema -a los recursos disponibles y a la carga de los mismos-. No se pasarán parámetros puesto que se gestionarán automáticamente.

```
#endif
{
        int threshold = MF * STD_SPD / speed;
```

Se define el umbral como el producto entre la varibale MF -inicializada con valor 100, es una constante obtenida por heurística en openMosix-, la STD_SPD, que contiene un valor de referencia para las velocidades de cada nodo y speed, que es la velocidad del nodo sobre el que se está estudiando su rendimiento -o tiempo de ejecución-.

```
#if CONFIG_MOSIX_LOADLIMIT
        switch(llimitmode)
          {
          case 0:
            if (load_limit && load > load_limit)
              return INFLOAD;
          break;
          case 1:
            if (load_limit && load_remote  > load_limit)
              return INFLOAD;
          case 2:
            if (load_limit && load_local  > load_limit)
              return INFLOAD;
            break;
          }
        switch(cpulimit_mode)
          {
          case 0:
            if (cpu_limit && ((cpu_remote+cpu_local) > cpu_limit))
              return INFLOAD;
          break;
          case 1:
            if (cpu_limit && cpu_remote  > cpu_limit)
              return INFLOAD;
          case 2:
```

```
            if (cpu_limit && cpu_local  > cpu_limit)
              return INFLOAD;
            break;
          }
#endif
        if(load <= threshold)
                return(threshold);
        if(load <= threshold + threshold/ncpus)
                return(threshold + ncpus * load * (load-threshold) / threshold);
        return(2 * load - threshold / ncpus);
}
```

Al salir de `altload()` se habrá obtenido el valor de retorno, que para las distintas situaciones que se han evaluado podrá ser:

- **INFLOAD:** cuando el límite de carga o de procesador en el nodo local sea menor a sí mismo y del existente en el nodo local, o en el remoto.
- **threshold:** cuando la carga del nodo sea menor que el umbral definido, o que el umbral más el cociente entre éste y el número de procesadores.
- **2 * load - threshold / ncpus:** valor por defecto que se sirve de un valor experimental que sirve para la mayor aprte de los casos.

Una función importante es la `choose()` que concretamente se encarga de decidir el proceso que se migrará, el bucle principal de espera de comunicaciones y otras.

Esta función es llamada cuando ya se ha elegido el proceso a migrar y sirve para llevar la cuenta de la carga de cada nodo.

Primero calcularemos el tiempo mínimo que tardaría el proceso a ejecutarse en un nodo. La línea

```
mintime = MILLION * acpuse / (smp_num_cpus * MF);
```

es lógica puesto que la carga que pueda tener un sistema SMP de por ejemplo 8 procesadores no puede ser la misma para los mismos procesos que un sistema con 1 solo procesador por lo tanto aquíse considera el caso ideal de que la carga se divide de forma perfecta entre los procesadores.

Ahora generamos una cota inferior `very_mintime`, tres veces inferior a la calculada (hecho que solo supone una medida de margen). Servirá para comparaciones de valor.

Luego se calcula la carga mínima de procesador del nodo en el momento actual (para fines de ponderación y comparación con los demás nodos) y seguidamente asignamos como mejor elección la dada por el valor mínimo.

La mejor carga aún está por definir. La implementación es

```
very_mintime = mintime / 3;
minload = 4 * acpuse / smp_num_cpus;    /* normally 4*MF */
bestpri = mintime;
bestload = -1;
```

Lo siguiente que se hace es ejecutar un bucle `for` para recorrer todas las cargas de cada nodo del cluster. En `m.load` es donde guardamos la información que nos llega desde otros nodos. Esta información es la que usamos para decidir si migrar un proceso o no,

Existe un número máximo de la ventana de conocimiento, una ventana de conocimiento completa (conocer todos los nodos) haría el cluster poco escalable tanto por comunicaciones como por algoritmo. Como se puede imaginar para tener una ventana de conocimiento completa, todos los nodos tendrían que enviar toda su información a todos los nodos, por lo tanto un nuevo nodo en el cluster incluiría bastante más información que enviar y no se escalaría de forma lineal (la relación número de nodos/información transmitida no sería lineal).

En cambio en la implementación de openMosix solamente se envía un número fijo de información a unos nodos elegidos aleatoriamente, esto hace que el cluster escale de forma lineal. Así mismo el algoritmo necesita

procesar toda la información que tiene sobre los otros nodos para decidir si son mejores candidatos que él mismo, por lo que si el número de nodos creciera y cada uno enviara su información, la cantidad de información que se debería procesar es mayor.

Cuando se encuentra una de las estructuras con información de los nodos que coincide con el nodo al que hemos decidido enviar el proceso, se procede a actualizar los datos referentes a ese nodo para que no lo volvamos a considerar la próxima vez por no estar algo más sobrecargado.

La forma que tiene openMosix de hacerlo es una aproximación, como no tiene información directa del otro nodo hace una aproximación ruda de la carga que el proceso haría en el otro nodo.

Realmente es una carga un tanto superior (alrededor de un 2 % superior) porque siempre un nodo que viene de fuera crea más carga (por ejemplo las caches de los procesadores no tienen datos de este nuevo proceso).

En la línea número 14 simplemente restamos las páginas que este proceso lleva consigo a las páginas disponibles en aquel nodo.

Este proceso ha sido elegido para la migración, pero no sabemos si ha sido por causa de que queremos balancear la carga o de que queremos balancear la memoria, por eso se hace unas comprobaciones, si le se elimina a `m->dflags` si se está balanceando con respecto a carga (DBALANCING) o respecto a memoria (DMBALANCING), en caso de que se esté balanceando con respecto a carga se quita ese valor comentado y se vuelve a ejecutar la función que busca para todos los procesos alguno que se pueda ejecutar mejor en otro nodo que en el nodo donde actualmente se encuentran.

Como esta función lleva al actual código, mientras siga el desbalanceo de la carga se seguirá eligiendo y migrando procesos. Si la carga es de memoria se vuelve a ejecutar la función que intenta repartir justamente la memoria. Con esto nos aseguramos que migraremos suficientes procesos para que la situación vuelva a la normalidad.

En este trozo de código hemos visto como se implementa el manejo de información de carga de CPU y memoria cuando un nodo migra.

### 6.7.3. *mig.c*

Se ha visto cómo openMosix decide migrar -y cuando-, no obstante aún falta ver como se realiza el proceso en síSi el lector está interesado en el algoritmo de migración deberá referirse a los ficheros correspondientes, pues aquí se supone que la decisión ya se ha hecho y se toma el proceso en ejecución para modificarlo y hacerlo apto para su ejecucición de forma remota.

Aquí se detalla pues cómo se dividen los procesos en las dos partes: el contexto de usuario -*remote*- y el contexto de sistema -*deputy*- para iniciar la ejecución fuera del nodo local. Ante la migración de un proceso pueden darse tres situaciones, tan distintas como lo son las estratégias requeridas para abordarlas:

- **De local a remoto.** Se realiza la comunicación -con confirmación- con el nodo remoto para el alojamiento del proceso que se le hará llegar.
- **De remoto a local.** Se alojará un proceso al nodo local, proviniente de uno remoto. Como en el caso anterior deberá producirse el diálogo que marca el protocolo.
- **De remoto a remoto.** Para conseguir que un proceso que ya no se está ejecutando en su nodo local migre hacia otro nodo también remoto habrá que generarse un canal de comunicación entre ellos, para luego destruirlo y generarlo entre el neuvo nodo y el local.

A continuación se abarcará cada uno de estos casos de forma separada, porque así se ha implementado en openMosix. Para cada caso pues se verá la comunicación entre las distintas partes del proceso y los diferentes nodos que aseguran que no se pierde información durante el proceso de migración, la posterior ejecución y finalmente el retorno de resultados.

#### Migrar un proceso local a un nodo remoto

Para este caso se ha implementado la función `mig_local_passto_remote` a la que hay que pasar como parámetros la localización del nodo remoto y la razón de la migración, a efectos de trazabilidad.

Los pasos que sigue esta función pueden enumerarse en cuatro más una fases:

1. reduce el proceso a *deputy* guardando todos los registros,
2. se crea una capa de enlace tipo *mosix_link* para la comunicación con el *remote*,
3. se comprueba la conexión con un *request* y su correspondiente asentimiento *ack*,
4. se migran al nodo remoto las páginas necesarias -y previamente computadas- con la función `mig_to_send`.
5. En caso de producirse un error, se restaura la situación inicial.

En caso de proceder exitosamente se llama a `deputy_migrated`, en caso contrario se elimina el componente *deputy* con una llamada a `undeputy`. A continuación se analiza la implementación, véase como sigue:

`int mig_local_passto_remote(int, int)`

```
static int
mig_local_passto_remote(int whereto, int reason)
{
        struct task_struct *p = current;
        struct mosix_link *mlink;
        int error;
        int omigpages;
```

En el `task_struct *p` se guardará toda la información del proceso actual. Se define también la estructura `mosix_link` que servirá de nexo entre los nodos -más bien entre las diferentes partes del proceso: el *deputy* en el local y el *remote* en el nodo remoto-.

```
        if(!p->mosix.held_files && (error = mosix_rebuild_file_list()))
                return(error);

        lock_mosix();
        write_lock_irq(&tasklist_lock);
        p->mosix.remote_caps = current->cap_effective;
        task_lock(p);
        spin_lock(&runqueue_lock);
```

Han quedado bloqueados los cuatro recursos que no deben ser modificados mientras se trabaja con la sección crítica de esta operación: guardar los registros del proceso. Como puede verse se protege esta sección con un método de semáforos.

```
        if(p->task_dumpable)
                p->mosix.dflags |= DTDUMPABLE;
        else
                p->mosix.dflags &= ~DTDUMPABLE;
        if(p->mm->dumpable)
                p->mosix.dflags |= DDUMPABLE;
        else
                p->mosix.dflags &= ~DDUMPABLE;
        p->mosix.dflags |= (DDEPUTY | DSYNC);
```

He aquí la sección crítica, que comprueba si el proceso y la memoria necesaria son migrables. En tal caso, se marca esta opción con DUMPABLE. Hay que notar que no es hasta este momento que se verifica si el proceso puede ser migrado. Así si por ejemplo se trata de un proceso de emulación VM86 no es hasta este momento que openMosix se percata que no puede aplicar la política de migración que previamente ya ha computado.

Seguidamente se ponen los semáforos *en verde*.

```
        spin_unlock(&runqueue_lock);
        task_unlock(p);
        write_unlock_irq(&tasklist_lock);
        unlock_mosix();
```

Una vez hechas las comprobaciones puede procederse a hacer una copia de los registros para que conformen el *deputy*. Véase que se guardan *todos* los registros.

```
p->mosix.deputy_regs = ALL_REGISTERS;
p->mosix.pass_regs = 0;
```

Se comprueba que se puede establecer una conexión entre las dos divisiones del proceso. El enlace *mlink* es de tipo *mosix_link* -tal como se ha definido- y es la manera que tiene openMosix de crear la capa de enlace entre los nodos con los que está interactuando.

```
if (!(mlink = comm_open(whereto, 0, comm_connect_timo))) {
        error = -EDIST;
        goto failed;
}
```

El parámetro *whereto* es el ID del nodo con el que se quiere abrir la conexión. És útil porque *mlink* queda como si fuera un descriptor de archivo y no se tiene que estar usando la dirección IP continuamente.

En caso de producirse un error, se recogerá su tipo y se sale con *failed*. La salida con *failed* implica deshacer cualquier cambio para dejar el contexto del proceso tal y como debía estar antes de iniciar cualquier intento de conexión.

Tras esto se utiliza la conexión para saber si `mosix_struct` realmente posee connexión. Es una llamada a `comm_use`.

```
if (comm_use(p, mlink))
        panic("local_passto_remote: previous contact not null");
```

Se procede a hacer el recuento del número de páginas de memoria que deberán migrarse al nodo remoto.

```
if(!(omigpages = p->mosix.migpages))
        p->mosix.migpages = count_migrating_pages();
```

Se le indica al demonio de migración del nodo remoto la intención -por parte del nodo local- de realizar una petición de migración. En la misma operación se indica la razón de la operación.

```
if ((error = mig_send_request(reason, FROM_DEPUTY)))
        {
                p->mosix.migpages = omigpages;
                goto failed;
        }
```

Se realizará la migración efectiva de las páginas. En la primera línea se permite la migración hacia ese nodo. En la siguiente se utiliza `mig_to_send()` que es realmente quien realiza la migración -se verá con más detalle-. En caso de error se procede como ya se ha mencionado.

```
release_migrations(whereto);
if (mig_do_send()) {
        error = -EDIST;
        p->mosix.migpages = omigpages;
        goto failed;
}
p->mosix.migpages = omigpages;

deputy_startup();
return (0);
```

Estas últimas dos líneas dejan constancia que la migración se completó con éxito.

```
 failed:
        if(mlink)
                comm_close(NULL);
        undeputy(current);
        return (error);
}
```

En el caso de que alguna operación no haya termiando satisfactoriamente se sale con *failed*. Esto implica que se cierren todas la conexiones y se llame a `undeputy`, el cual elimina los apsos seguidos para crear el *deputy*. Resumiendo, todo vuelve al estado existente antes de iniciar la función.

**Migrar un proceso remoto a un nodo local**

En el caso que ahora se aborda el nodo local deberá alojar correctamente el proceso entrante -y remoto- para concederle los recursos que éste solicite: procesador, memoria, acceso a disco, *etc*. Deberán hacerse las comunicaciones pertinentes para que el proceso remoto migre al nodo local. Todo ello se implementa en la función `mig_remote_passto_local`.

Esta función se ejecuta también cuando se quiere que un proceso local que está siendo ejecutado en un nodo remoto sea ejecutado en el nodo local de nuevo. Para ello se vuelven a unir el *remote* y el *deputy* en la misma computadora y se verá como es la operación inversa al anterior procedimiento -como es ntaural y lógico-. Seguidamente se dará la implementación para el siguiente método:

1. preparar una nueva estructura para hacer caber las tablas de páginas llegadas,
2. comunicar a *remote* que se le requiere en el nodo local,
3. el *remote* deberá recibir la petición de comunicación y enviar el correspondiente asentimiento,
4. con el canal de comunicación activo, llamar a `mig_to_receive` para realizar la migración efectiva de migración.
5. Si se procede con éxito se cierra el nexo de comunicación internodo -el *mosix_link*- y se llama a `undeputy` para liberar la memoria ocupada.

`int mig_remote_passto_local(int, int)`

```
static int
mig_remote_passto_local(int whereto, int reason)
{
        struct task_struct *p = current;
        struct mig_request_h *mrp;
```

Se declaran dos estructuras:

- `task_struct *p` para manejar la memoria del proceso y
- `mig_request_h *mrp` usada para el envío de información requerida para establecer comunicación entre las partes.

```
        int error = 0;
        long orlim_as;
        long orlim_rss;
        long orlim_stack;
        unsigned int load_came_in = 0;

        mosix_deputy_rusage(0);
        if(obtain_mm())
            return(-EDIST);
```

Se ha creado una estructura para la memoria del proceso entrante y se procede con la asignación. Esta estructura es la que maneja toda la memoria del proceso, es muy importante para su vida en el nuevo nodo.

Seguidamente se tratarán de gaurdar los antiguos límites de memoria del proceso. Será muy necesario en el caso de que hubiera que restablecerlos a causa de algún error.

```
orlim_as = p->rlim[RLIMIT_AS].rlim_cur;
orlim_rss = p->rlim[RLIMIT_RSS].rlim_cur;
orlim_stack = p->rlim[RLIMIT_STACK].rlim_cur;
p->rlim[RLIMIT_AS].rlim_cur = RLIM_INFINITY;
p->rlim[RLIMIT_RSS].rlim_cur = RLIM_INFINITY;
p->rlim[RLIMIT_STACK].rlim_cur = RLIM_INFINITY;
```

Se enviará un mensaje al demonio de migración que se encuentra en el nodo remoto. Se le indica lo que se quiere hacer con el proceso (DEP_COME_BACK) y la razón por la que se quiere hacer.

```
if ((error = comm_send(DEP_COME_BACK, (void *) &reason, sizeof(reason),
                                                NULL, 0, 0))) {
        end_coming_in(error);
        current->mosix.pages_i_bring = 0;
        deputy_die_on_communication();
}
```

Si la llamada fallara quedaría indicado poniendo las páginas traídas a cero. También se dejaría constancia que el *deputy* obtuvo un error de comunicación.

Luego se recibirá -en el nodo local- la petición de alojamiento de la migración por parte del remoto -que habrá gestionado correctamente la llamada DEP_COME_BACK-. En caso de no recibirse, se irá a `fail`.

```
if ((error = mig_recv_request(&mrp)))
        goto fail;
```

Se indica la inminente recepción de un nuevo proceso,

```
load_came_in = 1;
```

El canal de comunicaciones `mrp` que se ha usado para el control de la comunicación ya no será para nada necesario. El canal de comunicaciones que se usará para el envío de información -páginas- nada tiene que ver con este canal de control del procedimiento.

```
comm_free((void *) mrp);
```

Se indica al demonio de migración que proceda con la migación. Se manda la petición MIG_REQUEST para pedir a *remote* que inicie el enviío de sus páginas. Si no se puede comunicar con el demonio se indicará marcando que la comunicación ha muerto.

```
if ((error = comm_send(MIG_REQUEST|REPLY, (void *)&error, sizeof(int),
                                                NULL, 0, 0))) {
        end_coming_in(error);
        current->mosix.pages_i_bring = 0;
        deputy_die_on_communication();
        /*NOTREACHED*/
}
```

Se llama a la función que lleva cabo el transporte de las páginas. Más tarde se verá como procede. Una vez ejecutada esta función ya se ha realizado la migración y con el siguiente código se indicará tanto lo que se quiere hacer como el tratamiento de errores. Como se ve a continuación:

```
if (!(error = mig_do_receive())) {
```

Si se ha conseguido la migración ya no será necesario el canal de comunicación con el demonio.

```
        comm_close(NULL);
```

Ya no existe la distribución de *deputy* y *remote*, así que se elimina la distinción.

```
        undeputy(p);
```

Se quita al *deputy* la información sobre el tiempo que lleva como tal -siendo *deputy*-, las señales, se elimina el *mosix_link* con el nodo donde se encuentra el proceso, se limpian los *dflags*, se elimina la lista de ficheros remotos, *etc.* Se termina indicando que la operación se realizó con éxito.

```
                mosix_clear_statistics();
#ifdef SHOW_MIGRATIONS
                if(SHOW_MIGRATIONS)
                        printk("Wooooooooo.....\n");
#endif /*SHOW_MIGRATIONS*/
                end_coming_in(0);
                current->mosix.pages_i_bring = 0;
                if(p->mosix.dflags & DDELAYHELD)
                        {
                                p->mosix.dflags &= ~DDELAYHELD;
                                mosix_rebuild_file_list();
                        }
                return(0);
        }
```

A continuación viene el control de errores.

```
 fail:
        p->rlim[RLIMIT_AS].rlim_cur = orlim_as;
        p->rlim[RLIMIT_RSS].rlim_cur = orlim_rss;
        p->rlim[RLIMIT_STACK].rlim_cur = orlim_stack;
        exit_mm(p);
        if(load_came_in)
                {
                        end_coming_in(error);
                        current->mosix.pages_i_bring = 0;
                }

        return (error);
}
```

Al igual que en el caso anterior, si se dan errores habrá que dejarlo todo como antes de proceder.

**Migrar un proceso remoto a un nodo remoto**

Ahora el esquema del método es:

1. crear un enlace tipo *mosix_link* hacia el demonio de migración del nuevo nodo remoto,
2. enviar una petición de migración y coger la dirección del nuevo remoto para responder,
3. enviar la petición DEP_PLEASE_MIGRATE al nodo remoto actual, con la razón y con la dirección destino del nuevo nodo remoto a la que *mosix_link* debe conectar,
4. si se procede con éxito se cerrará el antiguo enlace *mosix_link* y se usará el nuevo.
5. En caso de producirse errores se cortaría la nueva connexión o se llamaría a `deputy_die_on_communication` y se devovlerá el mensaje de error.

`int mig_remote_passto_remote(int, int)`

```
static int
mig_remote_passto_remote(int whereto, int reason)
{
```

Se crearán las estructuras que permitirán empaquetar la información tanto del proceso que debe re-migrarse como de la capa de enlace con el *deputy*.

```
struct task_struct *p = current;
struct mosix_link *newmlink = 0, *oldmlink = 0;
struct please_migrate_h pm;
int status;
int error;

p->mosix.pass_regs = 0;
```

Seguidamente se abrirá la nueva conexión con el nuevo nodo remoto donde se quiere hacer llegar el proceso.

```
if (!(newmlink = comm_open(whereto, 0, comm_connect_timo)))
        return (-EDIST);
```

Para las siguientes llamadas tipo *comm_** se deberá utilizar el nuevo enlace, por ser llamadas al nuevo remoto.

```
if (!(oldmlink = comm_use(p, newmlink)))
        panic("remote_passto_remote: no previous contact");
```

Se comprueba la conexión desdel nuevo remoto hacia el *deputy* para ver si la capa de enlace *newmlink* fuciona como debe. Si no lo hace, se sale por `failed`.

```
if ((error = mig_send_request(reason, DEPUTY_PROBE)))
        goto failed;
```

Y se copia el *remote* del viejo remoto a la estructura `pm`.

```
if ((error = comm_copydata((void *)&pm.ma, sizeof(pm.ma), 0)))
        goto failed;
```

A partir de ahora se volverá a usar el antiguo canal de comunicación.

```
comm_use(p, oldmlink);
```

Se añaden a la estructura `pm` el destino y la razón de migración para localizar el nuevo nodo remoto.

```
pm.reason = reason;
pm.to = whereto;
```

Se enviía la estructura `pm` al nuevo remoto.

```
mosix_deputy_rusage(0);
if ((error = deputy_request(DEP_PLEASE_MIGRATE, &pm, sizeof(pm),
                NULL, 0, 0, (void **)&status, -sizeof(status))))
        goto fatal;
if ((error = status))
        goto failed;
```

Se usará el nuevo canal de comunicación. El viejo puede destruirse puesto que las comprobaciones de que se peude trabajar con el nuevo nodo remoto han sido satisfactorias.

```
comm_use(p, newmlink);
comm_close(oldmlink);
```

Finalmente se sale por `return(0)` que indica el éxito de la operación.

```
return (0);
```

Si algo no ha ido como se ha descrito, se restaura el contexto y se deshacen los cambios.

```
 failed:
        comm_use(p, oldmlink);
        comm_close(newmlink);
        return (error);

 fatal:
        comm_use(p, oldmlink);
        comm_close(newmlink);
        deputy_die_on_communication();
        /* NOTREACHED */
}
```

**Información enviada en las migraciones**

Las funciones que se encargan de la migración llaman a otra función llamada `mig_do_send` o `mig_do_receive`. Ya se ha comentado que estas funciones son las que realmente realizan la migración, ahora habrá que ver qué es lo que realmente se realiza en las funciones para ver como se están migrando realmente los procesos. No va a ahondarse en todas las implementaciones de las subrutinas, puesto que no suponen ninguna novedad algorítmica. Todas las funciones siguen semejantes estrategias y, como se podrá comprobar, los mismos pasos lógicos.

Así por ejemplo `mig_do_send`, encargada de realizar la migración efectiva de la información que requiere ser transferida, procede como sigue

`int mig_do_send()`

```
int
mig_do_send(void)
{
        int credit;     /* # of clean demand-pages */
```

Esta variable será el valor de retorno y contiene el número de páginas solicitadas.

```
        if(current->mm->context.segments)
                clear_LDT();

        comm_migration_mode(1);
        neutralize_my_load(1); /* don't count me: I'm going to disappear */

        if(mig_send_mm_stats() || mig_send_mm_areas() ||
          (credit = mig_send_pages()) < 0 ||
          (current->used_math && mig_send_fp()) ||
          (current->mm->context.segments && mig_send_ldt()) ||
          mig_send_misc(credit))
```

1. `mig_send_mm_stats` envía datos relativos a los tamaños y direcciones de la memoria, del segmento de pila, del de datos y de otras variables.
2. `mig_send_mm_areas` envía cada segmento en función de las cotas definidas por la función anterior.
3. `mig_send_pages` retorna el número de páginas de memoria ocupadas.
4. En el caso que el proceso necesite del coprocesador matemático se comprueba que el nodo destino lo posea.

Y si alguna de las llamadas anteriores falla

```
                {
                        comm_send(MIG_NOT_COMING, NULL, 0, NULL, 0, 0);
```

```
                        comm_migration_mode(0);
                        neutralize_my_load(0);
                        changed_my_mind_and_staying();
                        return(-1);
                }
        /* "comm_migration_mode(0);" was done by "mig_send_misc" */
        if(current->mm->context.segments)
                load_LDT(current->mm);
        neutralize_my_load(0);
        return(0);
}
```

Como puede verse se divide la migración en diferentes partes según sea la naturaleza de los datos a enviar. Se irán viendo cada una de estas funciones con más detalle. En caso de que la migración falle se le indica al nodo donde se quería enviar el proceso que la migración falló enviándole un MIG_NO_COMING que indica que se procederá a abortar la migración que había sido iniciada.

Es útil para evitar que el nodo remoto tenga que estar haciendo comprobaciones sobre la integridad del proceso: será el nodo local quien sabrá qué ocurrió y así se lo indicará.

La última función deshace todo lo que se había hecho en el nodo local, el proceso que había sido elegido es deseleccionado. Se vuelve al estado anterior al que se estába cuando se elegió el proceso para ser migrado.

Antes de poder comprender la función `mig_do_receive` sería necesario comprender el funcionamiento de las funciones a las que llama `mig_do_send`. Se seguirá el orden lógico de la función, y se empieza por la implementación de la función `mig_send_mm_stats`.

`int mig_send_mm_stats()`

```
int
mig_send_mm_stats(void)
{
        struct mm_stats_h s;
```

Se compara la longitud de la información entre

- la que está buscando la estructura *mm* del kernel
- la de su propia estructura

```
        if(sizeof(struct mm_stats_h) !=
           offsetof(struct mm_struct, env_end) -
           offsetof(struct mm_struct, start_code) + sizeof(long))
                panic("mig_send_mm_stats");
```

Se da un mensaje de pánico si la longitud no fuera la misma, hecho que solo puede acontecerse cuando se cambie esa estructura en el kernel.

Se copian desde *mm-> start_code* todos los campos con destino la estructura `s` de estados de memoria. Los campos que se copian son los referenciados en el cuadro 6.1.

```
        memcpy((caddr_t)&s, (caddr_t)&current->mm->start_code, sizeof(s));
```

Se llamará a `comm_send`. Si diese algún error éste se desplazaría hasta la función que ha llamado esta función -o séase `mig_do_send`- y se abortaría.

```
        expel_progress = 1;
        return(comm_send(MIG_MM_STATS, &s, sizeof(s), NULL, 0, 0));
}
```

| | |
|---|---|
| unsigned long start_code; | donde empieza el código del proceso |
| unsigned long end_code | dirección de memoria del final del código |
| unsigned long start_data; | dirección del inicio de los datos estáticos |
| unsigned long end_data; | dirección del final de estos datos |
| unsigned long start_brk; | dirección del inicio de la memoria dinámica |
| unsigned long brk; | donde acaba esta memoria |
| unsigned long start_stack; | inicio de la pila |
| unsigned long arg_start; | inicio de los argumentos |
| unsigned long arg_end; | fin de los argumentos |
| unsigned long env_start; | inicio de las variables de ambiente |
| unsigned long env_end; | fin de estas variables |

Cuadro 6.1: openMosix a fondo: Datos de la estructura *mm_stats_h*

Se envía con el flag MIG_MM_STATS activado. Esto es muy importante a la hora de recibir esta información -como se verá en `mig_do_receive`- puesto que es lo que indica

- que el tipo de estructura que se usa es *mm_stats_h*
- y cómo se traduce en la información local de los procesos.

Se está enviando la estructura por lo tanto se está enviando la información antes expuesta.

`mig_send_mm_areas()`

```
int
mig_send_mm_areas(void)
{
        struct task_struct *p = current;
        register struct vm_area_struct *vma;
        struct file *fp;
        struct mmap_parameters_h m;

        m.origin = (p->mosix.dflags & DDEPUTY) ? PE : p->mosix.deppe;
        m.fixed = 1;
```

Definición de las estructuras necesarias:

- `struct task_struct *p = current` para el proceso que se trata.
- `register struct vm_area_struct *vma` para recorrer sus áreas virtuales.
- `struct file *fp` para poder controlar el mapeo de sus ficheros, si existe.
- `struct mmap_parameters_h m;` es una estructura propia de openMosix -las anteriores son del kernel-. Es muy similar a la estructura donde en la función anterior se guardaba toda la información.

Ahora se tratarán continuamente las áreas virtuales, así que ante el desconocimiento de tal término se aconseja al lector releer el capítulo sobre sistemas operativos.

```
        for(vma = p->mm->mmap ; vma != NULL ; vma = vma->vm_next)
```

Este bucle, que iniciándose en la primera *vma* recorre todas las que tenga ese proceso (en particular *vm_next*)-indica el siguiente área virtual hasta que no hay más que entonces es NULL.

```
                {
                        m.addr = vma->vm_start;
                        m.len = vma->vm_end - vma->vm_start;
                        m.flags = vma->vm_flags;
```

Se guarda la posición virtual en que se inicia este área, su longitud -de esta forma sabremos cuando empieza y cuando termina- y los flags -solo lectura, *etc.*- puesto que es importante conservarlos cuando se migra el proceso a otro nodo.

Cuando un área virtual se ha creado al mapear un fichero a memoria, este área lo indica apuntando al fichero con *vm_file*, por lo tanto esta comprobación es cierta si el área fue creada al mapear un fichero a memoria, en ese caso se guarda la información del fichero.

```
if((fp = vma->vm_file))
        {
                struct inode *ip = fp->f_dentry->d_inode;
```

Se guarda el *offset* (o desplazamiento).

```
                m.pgoff = vma->vm_pgoff;
```

Se separan los casos entre el que exiete el *deputy* en el nodo y entre el que esta parte se encuentra ya en otro nodo. Esta última situación se da cuando se migra un proceso ya migrado.

```
                if(p->mosix.dflags & DREMOTE)
                        {
                                m.fp = home_file(fp);
                                m.dp = ip->u.remote_i.dp;
                                m.uniq = ip->u.remote_i.unique;
                                m.isize = ip->i_size;
                                m.nopage = ip->u.remote_i.nopage;
                        }
                else
                        {
                                m.fp = vma->vm_file;
                                m.dp = m.fp->f_dentry;
                                m.uniq = ip->i_unique;
                                m.isize = ip->i_size;
                                m.nopage = vma->vm_ops->nopage;
                        }
        }
```

Así según el proceso haya estado en el nodo local o en un nodo remoto se habrá buscado la información en un lugar u en otro. Si el proceso no era local sino remoto, se habrá buscado la información sobre el inodo del fichero en la estructura especial de openMosix donde se guarda esta información; de otro modo se busca la información requerida en la estructura normal usada por el kernel.

Por supuesto en caso de que no haya sido mapeado el fichero, tenemos que indicarlo, para ello *m.fp =NULL* y *m.pgoff=0*, indicando que no existe el fichero y por lo tanto el *offset* es cero.

```
        else
                {
                        m.fp = NULL;
                        m.pgoff = 0;
                }
```

Aquí se lleva a cabo el envío de información efectivo. Se puede apreciar que la forma de enviar la información es idéntica al caso anterior: MIG_MM_AREAS indica que la información que se envía es sobre las *vma* e indica que tipo de estructura se está enviando.

```
        if(comm_send(MIG_MM_AREA, &m, sizeof(m), NULL, 0, 0))
                return(-1);
}
```

Como puede verse esta información se encuentra en el bucle, por lo tanto se envía la información de cada una de las áreas.

```
        expel_progress = 1;
        return(0);
}
```

`mig_send_pages()`

```
int
mig_send_pages(void)
{
        int credit;

        credit = run_over_dirty_pages(mig_send_page, 1);
        return(credit);
}
```

Retorna el valor `credit` que contiene el número de páginas con contenido que deberán ser enviadas.

`int mig_send_misc(int)` Esta función envía toda la información -y la información sobre la información- recolectada y que abarca cualquier aspecto relacionado con las páginas de memoria, registros y otras variables que el proceso necesitará para ser autónomo en un nodo remoto. he aquí la implementación de la esta última función.

```
int
mig_send_misc(int credit)
{
        struct mig_misc_h m;
        clock_t utime, stime;
        extern unsigned long do_it_virt(struct task_struct *, unsigned long);
        void *hd;
        int hdln;
        register struct task_struct *p = current;
        siginfo_t *forced_sigs;

        m.ptrace = p->ptrace;
        m.dflags = p->mosix.dflags & (DTRACESYS1|DTRACESYS2);
        memcpy((caddr_t)m.debugreg, (caddr_t)p->thread.debugreg,
                   sizeof(m.debugreg));
        m.nice = p->nice;
        m.caps = p->cap_effective;
        p->mosix.remote_caps = m.caps;
        m.it_prof_incr = p->it_prof_incr;
        m.it_virt_incr = p->it_virt_incr;
```

Se enviarán los registros.

```
        if(((p->mosix.dflags & DDEPUTY) && p->mosix.deputy_regs) ||
           ((p->mosix.dflags & DREMOTE) &&
                p->mosix.deputy_regs != ALL_REGISTERS)) {
                memcpy((caddr_t)&m.regs, (caddr_t)p->mosix.altregs, sizeof(m.regs));
        }       /* else do not bother - DEPUTY will bring */

        m.fs = p->thread.fs;
        m.gs = p->thread.gs;
```

Y los límites de memoria del proceso.

```
        m.rlim_cpu = p->rlim[RLIMIT_CPU];
        m.rlim_data = p->rlim[RLIMIT_DATA];
        m.rlim_stack = p->rlim[RLIMIT_STACK];
        m.rlim_rss = p->rlim[RLIMIT_RSS];
        m.rlim_as = p->rlim[RLIMIT_AS];
#ifdef CONFIG_MOSIX_DFSA
        m.rlim_nofile = p->rlim[RLIMIT_NOFILE];
        m.rlim_fsz = p->rlim[RLIMIT_FSIZE];
#endif /* CONFIG_MOSIX_DFSA */
        m.stay = (p->mosix.stay & DNOMIGRATE) != 0;
        if(p->mosix.dflags & DDEPUTY)
                {
                        m.deppe = PE;
                        m.mypid = p->pid;
                        memcpy(m.features, boot_cpu_data.x86_capability,
                                   sizeof(m.features));
                }
        else
                {
                        m.deppe = p->mosix.deppe;
                        m.mypid = p->mosix.mypid;
                        memcpy(m.features, p->mosix.features, sizeof(m.features));
                        m.passedtime = p->mosix.passedtime;
                }
        m.deputy_regs = p->mosix.deputy_regs;
        m.deccycle = p->mosix.deccycle;
        m.decay = p->mosix.decay;
        m.dpolicy = p->mosix.dpolicy;
        memcpy(m.depcost, deputy_here, sizeof(deputy_here));
        m.depspeed = cpuspeed;
        m.nmigs = p->mosix.nmigs + 1;
        m.info.disclosure = p->mosix.disclosure;
        m.info.uid = p->uid;
        m.info.gid = p->gid;
        m.info.pgrp = p->pgrp;
        m.info.session = p->session;
        memcpy(m.info.comm, p->comm, sizeof(m.info.comm));
        m.info.tgid = p->tgid;
        cli();
        m.it_virt_value = p->it_virt_value;
        m.it_prof_value = p->it_prof_value;
        p->it_prof_value = 0;
        p->it_virt_value = 0;
        if(p->mosix.dflags & DDEPUTY)
                m.passedtime = p->times.tms_utime + p->times.tms_stime;
        utime = p->times.tms_utime;
        stime = p->times.tms_stime;
        m.asig.sigs = p->mosix.asig;
        m.asig.nforced = p->mosix.nforced_sigs;
        forced_sigs = p->mosix.forced_sigs;
        m.pagecredit = credit;
        m.lastxcpu = p->mosix.last_sigxcpu;
        sti();
        if(comm_send(MIG_MISC, &m, sizeof(m), forced_sigs,
                                m.asig.nforced * sizeof(siginfo_t), 0))
```

```
                goto fail;
        comm_migration_mode(0);
        expel_progress = 1;
        if(comm_recv(&hd, &hdln) == (MIG_MISC|REPLY))
                return(0); /* commit point */

 fail:
        cli();
        p->it_prof_value = m.it_prof_value;
        p->it_virt_value = m.it_virt_value;
        /* maintain accurate correlation between process-times and timers: */
        utime = p->times.tms_utime - utime;
        stime = p->times.tms_stime - stime;
        sti();
        if(utime > 0)
                {
                        extern rwlock_t xtime_lock;

                        write_lock_irq(&xtime_lock);
                        do_it_virt(p, utime);
                        write_unlock_irq(&xtime_lock);
                }
        if(!(p->mosix.dflags & DDEPUTY) && /* (DEPUTY gets it in "depticks") */
           utime + stime > 0)
                absorb_deptime(utime + stime);
        return(-1);
}
```

`int mig_do_receive()` Esta función gestionará toda recepción de los flags definidos en openMosix. Esto conlleva y completo control de la situación puesto que cada tipo de definición resume el problema que haya podido darse o proseguir con el diálogo con la otra parte del proceso para asegurar una correcta comunicación..

```
int
mig_do_receive(void)
{
        struct mosix_task *m = &current->mosix;
        int type;
        void *head;
        int hlen;
        int (*mmap_func)(struct mmap_parameters_h *, int);
        unsigned int got_not_coming = 0;

        spin_lock_irq(&runqueue_lock);
        m->dflags |= DINCOMING;
        spin_unlock_irq(&runqueue_lock);
        current->used_math = 0;
        while(1) {
                switch(type = comm_recv(&head, &hlen)) {
                case MIG_MM_STATS:
                        mig_do_receive_mm_stats((struct mm_stats_h *)head);
                        break;
                case MIG_MM_AREA:
                        if(m->dflags & DREMOTE)
                                mmap_func = remote_mmap;
```

```
                        else
                                mmap_func = deputy_remmap;
                        if(mmap_func((struct mmap_parameters_h *)head, 1))
                        {
                                comm_free(head);
                                goto fail;
                        }
                        break;
                case MIG_PAGE:
                        if(mig_do_receive_page(*((unsigned long *)head)))
                        {
                                comm_free(head);
                                goto fail;
                        }
                        break;
                case MIG_FP:
                        mig_do_receive_fp((union i387_union *)head);
                        break;
                case MIG_XFP:
                        mig_do_receive_xfp((union i387_union *)head);
                        break;
                case MIG_LDT:
                        if(mig_do_receive_ldt())
                        {
                                comm_free(head);
                                goto fail;
                        }
                        break;
                case MIG_MISC:
                        mig_do_receive_misc((struct mig_misc_h *)head);
                        comm_free(head);
                        spin_lock_irq(&runqueue_lock);
                        m->dflags &= ~DINCOMING;
                        spin_unlock_irq(&runqueue_lock);
                        flush_tlb();    /* for all the new pages */
                        comm_send(MIG_MISC|REPLY, NULL, 0, NULL, 0, 0);
                        return(0);
                case MIG_NOT_COMING:
                        got_not_coming = 1;
                        goto fail;
                default:
                        if(m->dflags & DDEPUTY)
                                deputy_communication_failed();
                        comm_free(head);
                        goto fail;
                }
                comm_free(head);
                if((m->dflags & DREMOTE) && (mosadmin_mode_block || !NPE))
                        goto fail;
        }
fail:

        if(type >= 0)
                comm_flushdata(COMM_ALLDATA);
        spin_lock_irq(&runqueue_lock);
```

```
        m->dflags &= ~DINCOMING;
        spin_unlock_irq(&runqueue_lock);
        if((m->dflags & DDEPUTY) && !got_not_coming)
        {
                /* receiving all the debris can take time and
                   someone may need the memory meanwhile! */
                current->mosix.pages_i_bring = 0;
                do_munmap(current->mm, 0, PAGE_OFFSET, 1);
                while((type = comm_recv(&head, &hlen)) >= 0 &&
                      type != MIG_NOT_COMING)
                {
                        comm_free(head);
                        comm_flushdata(COMM_ALLDATA);
                        if(type == MIG_MISC)
                                /* send anything but MIG_MISC|REPLY: */
                                /* they will then send MIG_NOT_COMING! */
                                comm_send(DEP_SYNC, NULL, 0, NULL, 0, 0);
                }
        }
        return(-1);
}
```

### 6.7.4. *info.c*

```
struct uplist {
        struct uplist *next;
        unsigned short pe;
        short age;
};
#ifdef CONFIG_MOSIX_LOADLIMIT
extern unsigned long load_limit;
extern unsigned long cpu_limit;
extern unsigned long mosadmin_mode_loadlimit;
extern unsigned long mosadmin_mode_cpulimit;
#endif

static struct uplist uplist[MAXKNOWNUP];
static struct uplist *uphead, *upfree;
static int info_nup;            /* number of processes in uplist */

struct loadinfo loadinfo[INFO_WIN];

static int info_recv_message(struct infomsg *, int);
static int info_send_message(int, struct infomsg *);
static void update_uplist(struct loadinfo *);
static void update_window(struct loadinfo *);
static void inform(void);
static void not_responding(int);
static void age_uplist(void);
static int rand(int, int);

static char INFODSTR[] = "oM_infoD";

#define INFO_BUFSIZE    8192
```

```
int
mosix_info_daemon(void *nothing)
{
        struct task_struct *p = current;
        struct loadinfo *load;
        struct infomsg *msg;
        static char info_buf[INFO_BUFSIZE]; /* (large) buffer for info load */

        common_daemon_setup(INFODSTR, 1);
        lock_mosix();
        info_proc = current;
        unlock_mosix();

restart:
        wait_for_mosix_configuration(&info_daemon_active);
        comm_init_linkpool();
        if (!p->mosix.contact) {
                comm_use(p, comm_open(COMM_INFO, 0, 0UL));
                if (!p->mosix.contact)
                {
                        printk("%s: failed comm_open - exiting\n", INFODSTR);
                        comm_free_linkpool();
                        if(p->mosix.contact)
                                comm_close(NULL);
                        lock_mosix();
                        info_daemon_active = 0;
                        info_proc = NULL;
                        unlock_mosix();
                        do_exit(0);
                }
        }

        msg = (struct infomsg *) info_buf;
        load = (struct loadinfo *) &msg->load;

        while (1) {
                comm_wait();

                /* if openMosix was shut down - restart everything */
                if (!PE) {
                        comm_close(NULL);
                        comm_free_linkpool();
                        goto restart;
                }

                while (info_recv_message(msg, INFO_BUFSIZE))
                {
                        update_uplist(load);
                        update_window(load);
                        if (!mosadmin_mode_quiet) {
                                load_balance();
                                memory_balance();
                        }
                }
```

```
                if(sigismember(&p->pending.signal, SIGALRM))
                {
                        spin_lock_irq(&p->sigmask_lock);
                        flush_signals(p);
                        spin_unlock_irq(&p->sigmask_lock);

                        if(mosadmin_mode_quiet)
                                continue;

                        if(PE)
                                inform();
                        inc_decays();
                        comm_age_linkpool();
                        age_uplist();
                        loadinfo[0].mem = latest_free_mem;
                        loadinfo[0].rmem = nr_free_pages();
                        age_balancing();
                }
        }
}

static void
age_uplist(void)
{
        struct uplist *up, *prev, *cutpoint;
        int old = 10 * (NPE-1);
        /* (the more nodes around, the less often they are
         * likely to come up again, so we hold them longer) */

        if(old > 32767)
                old = 32767;
        spin_lock(&uplist_lock);
        for (prev = NULL, up = uphead; up; prev = up , up = up->next)
        if (++up->age >= old)
        {
                /* the list is sorted by age, so all the rest are too old! */
                if (prev)
                        prev->next = NULL;
                else
                        uphead = NULL;
                cutpoint = up;
                while(1)
                {
                        info_nup--;
                        if(up->next)
                                up = up->next;
                        else
                                break;
                }
                prev = upfree;
                upfree = cutpoint;
                up->next = prev;
                break;
        }
```

```
        spin_unlock(&uplist_lock);
}

static void
update_uplist(struct loadinfo *load)
{
        struct uplist *up, *prev;

        spin_lock(&uplist_lock);
        for (prev = NULL , up = uphead ; up ; prev = up , up = up->next)
        if (up->pe == load->pe)
        {
                up->age = 0;
                if (prev)       /* put it first */
                {
                        prev->next = up->next;
                        up->next = uphead;
                        uphead = up;
                }
                break;
        }
        if (!up) {
                if (upfree) {
                        up = upfree;
                        upfree = upfree->next;
                        info_nup++;
                } else {
                        for (prev = uphead ; prev->next->next ;
                                prev = prev->next)
                                ; /* nothing */
                        up = prev->next;
                        prev->next = NULL;
                }
                up->pe = load->pe;
                up->age = 0;
                up->next = uphead;
                uphead = up;
        }
        spin_unlock(&uplist_lock);
        schedule(); /* since a lot of time passed, let's see if there is anything to run in the me
}
static void
update_window(struct loadinfo *load)
{
        static int info_slot; /* pointer to next information to fill */
        unsigned int i;

        info_slot = (info_slot % (INFO_WIN - 1)) + 1;
        loadinfo[info_slot] = *load;

        for (i = 1 ; i < INFO_WIN ; i++)
        if (i != info_slot && loadinfo[i].pe == load->pe)
        {
                write_lock_bh(&loadinfo_lock);
                loadinfo[i].pe = 0;
```

```
                write_unlock_bh(&loadinfo_lock);
                break;
        }

}

#define speed_adjust(x)         ((x) * ((int64_t)STD_SPD) / loadinfo[0].speed)
void
info_update_costs(void)
{
        register unsigned int i;

        write_lock_bh(&loadinfo_lock);
        for(i = 0 ; i < MAX_MOSIX_TOPOLOGY ; i++)
        {
#ifdef CONFIG_MOSIX_TOPOLOGY
                loadinfo[0].costs[i].page = mosix_cost[i].PAGE_R;
                loadinfo[0].costs[i].syscall = mosix_cost[i].SYSCALL_R;
                loadinfo[0].costs[i].out = mosix_cost[i].COPYOUT_BASE_R;
                loadinfo[0].costs[i].outkb = mosix_cost[i].COPYOUT_PER_KB_R;
                loadinfo[0].costs[i].in = mosix_cost[i].COPYIN_BASE_R;
                loadinfo[0].costs[i].inkb = mosix_cost[i].COPYIN_PER_KB_R;
                loadinfo[0].costs[i].first = mosix_cost[i].first;
                loadinfo[0].costs[i].last = mosix_cost[i].last;
#else
                remote_here.page = mosix_cost[i].PAGE_R;
                remote_here.syscall = mosix_cost[i].SYSCALL_R;
                remote_here.out = mosix_cost[i].COPYOUT_BASE_R;
                remote_here.outkb = mosix_cost[i].COPYOUT_PER_KB_R;
                remote_here.in = mosix_cost[i].COPYIN_BASE_R;
                remote_here.inkb = mosix_cost[i].COPYIN_PER_KB_R;
#endif /* CONFIG_MOSIX_TOPOLOGY */
                remote_here_adjusted[i].page =
                                        speed_adjust(mosix_cost[i].PAGE_R);
                remote_here_adjusted[i].syscall =
                                        speed_adjust(mosix_cost[i].SYSCALL_R);
                remote_here_adjusted[i].out =
                                speed_adjust(mosix_cost[i].COPYOUT_BASE_R);
                remote_here_adjusted[i].outkb =
                                speed_adjust(mosix_cost[i].COPYOUT_PER_KB_R);
                remote_here_adjusted[i].in =
                                speed_adjust(mosix_cost[i].COPYIN_BASE_R);
                remote_here_adjusted[i].inkb =
                                speed_adjust(mosix_cost[i].COPYIN_PER_KB_R);
#ifdef CONFIG_MOSIX_TOPOLOGY
                remote_here_adjusted[i].first = mosix_cost[i].first;
                remote_here_adjusted[i].last = mosix_cost[i].last;
#endif /* CONFIG_MOSIX_TOPOLOGY */
        }
        write_unlock_bh(&loadinfo_lock);
}

void
info_update_mfscosts(void)
{
```

```
#ifdef CONFIG_MOSIX_TOPOLOGY
        memcpy(loadinfo[0].mfscosts, mfs_cost, sizeof(mfs_cost));
#endif /* CONFIG_MOSIX_TOPOLOGY */
}
#ifdef CONFIG_MOSIX_LOADLIMIT
void set_loadlimit(void)
{
  write_lock_bh(&loadinfo_lock);
  loadinfo[0].load_limit = load_limit;
  write_unlock_bh(&loadinfo_lock);
}
void set_loadlimitmode(void)
{
  write_lock_bh(&loadinfo_lock);
  loadinfo[0].llimitmode = mosadmin_mode_loadlimit;
  write_unlock_bh(&loadinfo_lock);
}
void set_cpulimitmode(void)
{
  write_lock_bh(&loadinfo_lock);
  loadinfo[0].cpulimitmode = mosadmin_mode_cpulimit;
  write_unlock_bh(&loadinfo_lock);
}
void set_cpulimit(void)
{
  write_lock_bh(&loadinfo_lock);
  loadinfo[0].cpu_limit = cpu_limit;
  write_unlock_bh(&loadinfo_lock);
}

#endif
void
set_my_cpuspeed(void)
{
        int s = cpuspeed;

        if(sizeof(loadinfo[0].speed) < 4 && s > 65535)
        {
                printk("Computer Too Fast!  Time to Update Standard-Speed.\n");
                s = 65535;
        }
        stable_export = (MF+2) * STD_SPD / (s * smp_num_cpus);
        if(stable_export == MF * STD_SPD / (s * smp_num_cpus))
                stable_export++;
        write_lock_bh(&loadinfo_lock);
        loadinfo[0].speed = s;
        write_unlock_bh(&loadinfo_lock);
        info_update_costs();
        info_update_mfscosts();
}

void
info_init(void)
{
        loadinfo[0].ncpus = smp_num_cpus;
```

```
        loadinfo[0].tmem = num_physpages;
        set_my_cpuspeed();
}

void
info_startup(void)
{
        unsigned int i;

        info_seed1 = PE;
        info_seed2 = PE*PE*PE*PE;
        write_lock_bh(&loadinfo_lock);
        loadinfo[0].pe = PE;
        for (i = 1 ; i < INFO_WIN ; i++) {
                loadinfo[i].pe = 0;
                loadinfo[i].load = 0xffffffff;
        }
        write_unlock_bh(&loadinfo_lock);

        spin_lock(&uplist_lock);
        upfree = uphead = NULL;
        info_nup = 0;

        memset(uplist, 0, sizeof(struct uplist) * MAXKNOWNUP);
        for (i = 0; i < MAXKNOWNUP; i++) {
                uplist[i].next = upfree;
                upfree = &uplist[i];
        }
        spin_unlock(&uplist_lock);
}

void
info_reconfig()
{
        unsigned int i;
        struct uplist *up, *prev;

        lock_mosix();   /* because "mos_to_net" does! */
        spin_lock(&uplist_lock);
recheck:
        prev = NULL;
        for (up = uphead ; up ; prev = up, up = up->next)
        if (!mos_to_net(up->pe, NULL)) {
                if (prev)
                        prev->next = up->next;
                else
                        uphead = up->next;
                up->next = upfree;
                upfree = up;
                info_nup--;
                goto recheck;
        }
        spin_unlock(&uplist_lock);
        write_lock_bh(&loadinfo_lock);
        for (i = 1; i < INFO_WIN; i++)
```

```
                if (loadinfo[i].pe && !mos_to_net(loadinfo[i].pe, NULL))
                        loadinfo[i].pe = 0;
        write_unlock_bh(&loadinfo_lock);
        unlock_mosix();
}

static int
info_recv_message(struct infomsg *info, int bufsize)
{
        static mosix_addr ra;           /* reply address */
        struct mosix_link *l = current->mosix.contact;
        struct loadinfo *load = &(info->load);
        int n;
        int sender;

        while (1) {
                n = comm_recvfrom(info, bufsize, l, &ra, 0);

                if (n == -EDIST)
                        continue;       /* message > bufsize */

                if (n < 0)
                        return (0);

                if (n < sizeof(*info)) {
                        continue;
                }

                if (info->version != MOSIX_BALANCE_VERSION) {
                        continue;
                }
                if (info->topology != MAX_MOSIX_TOPOLOGY) {
                        continue;
                }

                sender = load->pe ? : load->speed;
                if (sender > MAXPE || sender != net_to_mos(&ra)) {
                        continue;
                }

                if (sender == PE)
                {
                        printk("WARNING: Another computer is masquerading as same openMosix node a
                        continue;
                }


                if (load->pe)
                        return (1);

                /*
                 * Ugly convention: !load->pe ==> this is a GETLOAD request
                 */

                lock_mosix();
```

```
                write_lock_bh(&loadinfo_lock);
                loadinfo[0].status = my_mosix_status();
                loadinfo[0].free_slots = get_free_guest_slots();
                unlock_mosix();
                loadinfo[0].mem = latest_free_mem;
                loadinfo[0].rmem = nr_free_pages();
                loadinfo[0].util = acpuse;
#ifdef CONFIG_MOSIX_LOADLIMIT
                loadinfo[0].load_limit = load_limit;
                loadinfo[0].cpu_limit = cpu_limit;
                loadinfo[0].llimitmode = mosadmin_mode_loadlimit;
                loadinfo[0].cpulimitmode = mosadmin_mode_cpulimit;
#endif
#ifdef CONFIG_MOSIX_RESEARCH
                loadinfo[0].rio = io_read_rate;
                loadinfo[0].wio = io_write_rate;
#endif /* CONFIG_MOSIX_RESEARCH */
                *load = loadinfo[0];
                write_unlock_bh(&loadinfo_lock);


                comm_sendto(COMM_TOADDR, info, sizeof(*info), l, &ra);
        }
        return(0);
}

static int
info_send_message(int mos, struct infomsg *info)
{

        return(comm_sendto(mos, info, sizeof(*info), current->mosix.contact,
                                                                NULL) < 0);
}

int
load_to_mosix_info(struct loadinfo l, struct mosix_info *info, int touser)
{
        struct mosix_info tmp, *uaddr;
        int error;

        if(touser)
        {
                uaddr = info;
                info = &tmp;
        }
        else
                uaddr = NULL;   /* pacify angry stupid gcc */
        info->load = ((int64_t)l.load) * standard_speed / STD_SPD;
#ifdef CONFIG_MOSIX_LOADLIMIT
        info->load_limit = l.load_limit;
        info->llimitmode = l.llimitmode;
        info->loadlocal = l.loadlocal;
        info->loadremote = l.loadremote;
        info->cpulimitmode = l.cpulimitmode;
        info->cpu_limit = l.cpu_limit;
```

```
        info->cpulocal = l.cpulocal;
        info->cpuremote = l.cpuremote;
#endif
        info->speed = l.speed;
        info->ncpus = l.ncpus;
        info->mem = l.mem * PAGE_SIZE;
        info->rmem = l.rmem * PAGE_SIZE;
        info->tmem = l.tmem * PAGE_SIZE;
        info->util = l.util;

        info->status = l.status;
        if(touser && (error = copy_to_user((char *)uaddr, (char *)info,
                                                                sizeof(*info))))
                return(-EFAULT);
        return(0);
}

static int info_serialno = 0;
static int info_timo;
static int info_retry;
static int info_retry_cnt;

#define INFO_TIMO 50000         /* ms */
#define INFO_RETRY 20000        /* after 20 & 40 ms */

#define SIMULTANEOUS_QUERIES            10

void
mosinfo_update_gateways(void)
{
        /* the following may be further optimized in the future: */
        info_timo = INFO_TIMO * (mosadmin_gateways + 1);
#if INFO_RETRY * 3 > 65535
#error: "timtosend" below must be made "unsigned int".
#endif
        info_retry = INFO_RETRY * (mosadmin_gateways + 1);
        info_retry_cnt = (info_timo + info_retry - 1) / info_retry;
}

int
balance_get_infos(int first, int num, struct mosix_info *info, int touser)
{
        char *donebits, fewbits[20]; /* 20*8=160 seems to cover most cases */
        struct progress
        {
                unsigned short node;
                unsigned short timtosend;
                unsigned int timo;
                int serialno;
        } progress[SIMULTANEOUS_QUERIES];
        unsigned int hint = 0;
        unsigned int ntaken = 0;
        unsigned int ndone = 0;
        unsigned int inpot = 0;
        int node, from, n;
```

```
        unsigned int i;
        int error = 0;
        int timo;
        now_t before;
        mosix_link *mlink;
        struct loadinfo l;
        struct infomsg infomsg;

        if(num <= 8*sizeof(fewbits))
                donebits = fewbits;
        else if(!(donebits = (char *)kmalloc((num+7)/8, GFP_KERNEL)))
                return(-ENOMEM);
        memset(donebits, 0, (num+7)/8);
        if (!(mlink = comm_borrow_linkpool()))
                error = -EDIST;
        loop:
        while(!error && ndone < num)
        {
                while(inpot < SIMULTANEOUS_QUERIES && ntaken < num)
                {
                        while(donebits[hint/8] & (1 << (hint%8)))
                        if(++hint == num)
                                hint = 0;
                        donebits[hint/8] |= (1 << (hint%8));
                        ntaken++;
                        node = first + hint;
                        if (node == PE)
                        {
                                read_lock_bh(&loadinfo_lock);
                                l = loadinfo[0];
                                read_unlock_bh(&loadinfo_lock);
                                l.status = my_mosix_status();
#ifdef CONFIG_MOSIX_LOADLIMIT
                                l.load_limit = load_limit;
                                l.llimitmode = mosadmin_mode_loadlimit;
                                l.cpulimitmode = mosadmin_mode_cpulimit;
                                l.cpu_limit = cpu_limit;
#endif
                                l.util = acpuse;
#ifdef CONFIG_MOSIX_RESEARCH
                                l.rio = io_read_rate;
                                l.wio = io_write_rate;
#endif /* CONFIG_MOSIX_RESEARCH  */
                                ready_immediate:
                                ndone++;
                                if((error = load_to_mosix_info(l, &info[hint],
                                                                touser)))
                                        break;
                                continue;
                        }
                        if (!mos_to_net(node, NULL))
                        {
                                l.status = 0;
                                goto ready_immediate;
                        }
```

```
        progress[inpot].node = node;
        progress[inpot].timo = info_timo;
        progress[inpot].timtosend = 0;
        progress[inpot].serialno = info_serialno++;
        inpot++;
}
if(error || ndone == num)
        break;
timo = info_retry;
for(i = 0 ; i < inpot ; i++)
{
        if(progress[i].timo < timo)
                timo = progress[i].timo;
        if(progress[i].timo && progress[i].timtosend <= 0)
        {
                progress[i].timtosend = info_retry;
                infomsg.version = MOSIX_BALANCE_VERSION;
                infomsg.serialno = progress[i].serialno;
                infomsg.topology = MAX_MOSIX_TOPOLOGY;
                infomsg.load.pe = 0;    /* eg. GETLOAD */
                infomsg.load.speed = PE;
                if(comm_sendto(progress[i].node, &infomsg,
                        sizeof(infomsg), mlink, NULL) <= 0)
                {
                        node = progress[i].node;
                        progress[i] = progress[--inpot];
                        ndone++;
                        l.status = DS_MOSIX_DEF;
                        error = load_to_mosix_info(l,
                                &info[node - first], touser);
                        goto loop;
                }
        }
        if(progress[i].timtosend < timo)
                timo = progress[i].timtosend;
}
before = time_now();
n = comm_recvfrompe(&infomsg, sizeof(infomsg), mlink, &from,
                                                    info_retry);
if (n == sizeof(infomsg))
for (i = 0; i < inpot; i++) {
        if (from == progress[i].node &&
            infomsg.load.pe == progress[i].node &&
            infomsg.serialno == progress[i].serialno) {
                ndone++;
                node = progress[i].node;
                progress[i] = progress[--inpot];
                error = load_to_mosix_info(infomsg.load,
                                &info[node - first], touser);
                break;
        }
} else if (signal_pending(current))
        error = -EINTR;
if (error)
        break;
```

```
                before = time_since(before);
                if(before > timo)
                        before = timo;
                if(before)
                for(i = 0 ; i < inpot ; i++)
                {
                        progress[i].timo -= before;
                        if(progress[i].timtosend < before)
                                progress[i].timtosend = 0;
                        else
                                progress[i].timtosend -= before;
                }
                if(n <= 0)
                for(i = 0 ; i < inpot ; i++)
                if(progress[i].timo <= 0)       /* cannot realy be < */
                {
                        node = progress[i].node;
                        progress[i] = progress[--inpot];
                        ndone++;
                        l.status = DS_MOSIX_DEF;
                        if((error = load_to_mosix_info(l,
                                &info[node - first], touser)))
                                break;
                        i--;
                }
        }
        if (mlink)
                comm_return_linkpool(mlink);
        if(num > 8*sizeof(fewbits))
                kfree(donebits);
        return(error);
}
int
balance_ask_node(int node, struct infomsg *info)
{
        mosix_link *mlink = NULL;
        int from, n, error = 0;
        int serialno;
        int tries;
        now_t before;
        int timo;

        if (!(mlink = comm_borrow_linkpool()))
                error = -EDIST;
        serialno = info_serialno++;
        tries = info_retry_cnt;
        timo = info_retry;
        while (!error && tries--) {
                info->version = MOSIX_BALANCE_VERSION;
                info->serialno = serialno;
                info->topology = MAX_MOSIX_TOPOLOGY;
                info->load.pe = 0;      /* eg. GETLOAD */
                info->load.speed = PE;
                error = comm_sendto(node, info, sizeof(*info), mlink, NULL);
```

```
                if (error < sizeof(*info))
                {
                        if(error >= 0)
                                error = -EDIST;
                        goto out;
                }
                error = 0;
                before = time_now();
                n = comm_recvfrompe(info, sizeof(*info), mlink, &from, timo);
                if (n == sizeof(*info) && from == node &&
                        info->load.pe == node && info->serialno == serialno)
                        goto out;
                before = time_since(before);
                if(before < timo)
                {
                        timo -= before;
                        tries++;
                }
                else
                        timo = info_retry;
        }
        error = -EAGAIN;
out:
        if (mlink)
                comm_return_linkpool(mlink);
        return (error);
}

int
balance_get_info(int node, struct mosix_info *info)
{
        struct infomsg infomsg;
        int error;

        if (node == PE || node == 0)            /* local info */
        {
                read_lock_bh(&loadinfo_lock);
                infomsg.load = loadinfo[0];
                read_unlock_bh(&loadinfo_lock);
                infomsg.load.status = my_mosix_status();
#ifdef CONFIG_MOSIX_LOADLIMIT
                infomsg.load.load_limit = load_limit;
                infomsg.load.llimitmode = mosadmin_mode_loadlimit;
                infomsg.load.cpulimitmode = mosadmin_mode_cpulimit;
                infomsg.load.cpu_limit = cpu_limit;
#endif
                infomsg.load.util = acpuse;
#ifdef CONFIG_MOSIX_RESEARCH
                infomsg.load.rio  = io_read_rate;
                infomsg.load.wio  = io_write_rate;
#endif /* CONFIG_MOSIX_RESEARCH */

                load_to_mosix_info(infomsg.load, info, 0);
        }
        else if (!mos_to_net(node, NULL))
```

```
                info->status = 0;
        else if((error = balance_ask_node(node, &infomsg)))
        {
                info->status = DS_MOSIX_DEF;
                return(error);
        }
        else
                load_to_mosix_info(infomsg.load, info, 0);
        return(0);
}

int
balance_get_load(int node, struct loadinfo *l)
{
        struct infomsg infomsg;

        if (node == PE || node == 0)
        {
                *l = loadinfo[0];
                return(0);
        }
        else if (!mos_to_net(node, NULL))
                return(-1);
        else if(balance_ask_node(node, &infomsg))
                return(-1);
        *l = infomsg.load;
        return(0);
}

static void
inform()
{
        int to;
        int i;
        struct uplist *up;
        struct infomsg info;

        info.version = MOSIX_BALANCE_VERSION;
        info.topology = MAX_MOSIX_TOPOLOGY;
        info.serialno = 0;              /* meaning no serial number */
        write_lock_bh(&loadinfo_lock);
        loadinfo[0].free_slots = get_free_guest_slots();
        info.load = loadinfo[0];
        write_unlock_bh(&loadinfo_lock);
        info.load.load = export_load;

        /* first select any node, and send the load */
        lock_mosix();
        to = (NPE > 1) ? nth_node(rand(NPE-1, 1)) : 0;
        unlock_mosix();
        if(to && info_send_message(to, &info))
                not_responding(to);

        /* then select a node that seems to be up */
        spin_lock(&uplist_lock);
```

```
        if (info_nup)
        {
                for (up = uphead , i = rand(info_nup, 0) ; i-- ; up = up->next)
                        ; /* just stop at random element */
                to = (up->pe == to) ? 0 : up->pe;
        }
        else
                to = 0;
        spin_unlock(&uplist_lock);
        if (to && info_send_message(to, &info))
                not_responding(to);
}

void
not_responding(int pe)
{
        unsigned int i;
        struct uplist *up, *prev;

        spin_lock(&uplist_lock);
        prev = NULL;
        for (up = uphead ; up ; prev = up , up = up->next)
        if (up->pe == pe)
        {
                if (prev)
                        prev->next = up->next;
                else
                        uphead = up->next;
                up->next = upfree;
                upfree = up;
                info_nup--;
                break;
        }
        spin_unlock(&uplist_lock);
        write_lock_bh(&loadinfo_lock);
        for (i = 1; i < INFO_WIN; i++)
        if (loadinfo[i].pe == pe)
                loadinfo[i].pe = 0;
        write_unlock_bh(&loadinfo_lock);
}

void
this_machine_is_favourite(int which)
{
        register struct uplist *up;

        spin_lock(&uplist_lock);
        for(up = uphead ; up ; up = up->next)
                if(up->pe == which)
                        break;
        if(!up && upfree)
#ifdef CONFIG_MOSIX_CHEAT_MIGSELF
        if(which != PE)
#endif /* CONFIG_MOSIX_CHEAT_MIGSELF */
        {
```

```
                up = upfree;
                upfree = upfree->next;
                up->next = uphead;
                up->pe = which;
                up->age = 0;
                uphead = up;
                info_nup++;
        }
        spin_unlock(&uplist_lock);
}

static int
rand(int modulo, int regen)
{
        if(regen)
        {
                info_seed2++;
                /* alternating even/odd values: */
                info_seed1 = info_seed1*info_seed2 + 1;
                return((info_seed1 & 0x7fffffff) % modulo);
        }
        else
                return((((info_seed2+1)*info_seed1+1) & 0x7fffffff) % modulo);
}

/*
 * we are 99.99% going to migrate and let other processes be migrated,
 * but not before we adjust the local and remote loads to discourage
 * further migrations.
 */
void
release_migrations(int whereto)
{
        register struct mosix_task *m = &current->mosix;
        register int load;
        unsigned int i;
        int pages = m->migpages ? : count_migrating_pages();

        this_machine_is_favourite(whereto);

        /* Decrease the local load by the load caused by this process,
         * to avoid over-migration.
         */
        write_lock_bh(&loadinfo_lock);
        spin_lock_irq(&runqueue_lock);
        load = m->load * STD_SPD / 4 / cpuspeed;
        load /= smp_num_cpus;
          /* It is ON PURPOSE that 'acpuse' is not taken into account */
        if(loadinfo[0].load < load)   /* should not happen, but ... */
                load = loadinfo[0].load;
        load_left += load;
        spin_unlock_irq(&runqueue_lock);

        loadinfo[0].load -= load;
```

```
        /* increase the receiver's-load */
        for(i = 1 ; i < INFO_WIN ; i++)
        if(loadinfo[i].pe == whereto)
        {
                /* add slightly more than 1 process worth of load */
                loadinfo[i].load += MF * 102 * STD_SPD/
                        (loadinfo[i].speed * loadinfo[i].ncpus * 100);
                loadinfo[i].mem -= pages;
                break;
        }
        write_unlock_bh(&loadinfo_lock);
        m->pages_i_bring = -pages; /* discourage 'memory_badly_required' */
        unchoose_me();
}

void
info_someone_came_in(void)
{
        write_lock_bh(&loadinfo_lock);
        coming_in++;
        came_lately4 += 4;
        export_load += MF * STD_SPD / (smp_num_cpus * cpuspeed);
        write_unlock_bh(&loadinfo_lock);
}

void
end_coming_in(int error)
{
        write_lock_bh(&loadinfo_lock);
        coming_in--;
        if(error)
        {
                if((int)(came_lately4 -= 4) < 0)
                        came_lately4 = 0;
        }
        write_unlock_bh(&loadinfo_lock);
}
```

### 6.7.5. *comm.c*

En este fichero se implementa toda la política de comunicación, que incluye:

- capa de enlace entre *deputy* y *remote*.
- paso de mensajes.
- protocolo de comunicación para el envío del contexto desdel UHN al *remote*, para permitirle su ejecución remota.

La estructura más importante quizás sea la que define la capa de enlace

```
struct mosix_link {
        /* the first 2 elements are common to both types and must stay there */
        struct socket *sock;    /* socket para comunicaciones */
        int flags;              /* flags de estado */
        int dlen;               /* longitud de los datos pendientes */
        int peer;               /* mosix ID del nodo # of peer */
```

```
        char *hideptr;          /* puntero a los datos del proceso en curso */
        char *hidebuf;          /* puntero a la posici\'on del buffer para los  datos */
        int hidelen;            /* longitud de los datos en el buffer */
        char head[COMM_DEF_HEADSIZE];
};
```

### mosix_link *comm_open()

Abre un socket para las comunicaciones de openMosix. El tipo de conexión puede variar, se define para el entero pasado por parámetro `mos` la siguiente relación:

- mos > 0, para conectar con el daemon de migración `mig-daemon` del nodo con IP = mos.
- mos = COMM_TOADDR, para conectar con la dirección dada en el campo *maddr->saddr* .
- mos = COMM_ACCEPT, configura un socket y lo prepara para ceptar una comunicación.
- mos = COMM_MIGD, configura un socket y lo prepara para `mig-daemon`.
- mos = COMM_INFOD, configura un socket para `info-daemon`.
- mos = COMM_LOOSE, para conectar con múltiples daemons.

Véase la implementación:

```
mosix_link *
comm_open(int mos, mosix_addr *maddr, unsigned long timo)
{
        int error = -EDIST;
        struct socket *sock = NULL;
        mosix_link *mlink = NULL;
        struct sockaddr sa;
        int connect = 1, bind = 1, need_comm = 1, listen = 1, peer = 0;
        struct sockaddr *saddr = &(maddr->saddr);
        DECLARE_WAITQUEUE(wait, current);

        /*   tabla de los flags para cada caso
         * COMM_INFO:   connect = 0, need_comm = 0, listen = 0, bind = 1
         * COMM_MIGD:   connect = 0, need_comm = 1, listen = 1, bind = 1
         * COMM_ACCEPT: connect = 0, need_comm = 1, listen = 1, bind = 1
         * COMM_TOADDR: connect = 1, need_comm = 1, listen = 0, bind = 1
         * COMM_LOOSE:  connect = 0, need_comm = 0, listen = 0, bind = 0
         * default:     connect = 1, need_comm = 1, listen = 1, bind = 1
         */
        switch (mos) {
        case COMM_LOOSE:
                bind = 0;
                /* fall through */
        case COMM_INFO:
                listen = 0;
                /* fall through */
        case COMM_MIGD:
                need_comm = 0;
                /* fall through */
        case COMM_ACCEPT:
                connect = 0;
                if(!saddr)
```

```
                saddr = &sa;
        break;
case COMM_TOADDR:
        if(saddr->sa_family != AF_INET)
                return(NULL);
        break;
default:
        peer = mos;
        saddr = &sa;
        break;
}
DISABLE_EVENTS();
/* fill in socket address and allocate a socket */
if (!(sock = comm_set_address(mos, saddr, 1)))
        goto failed;

if (need_comm) {                /* si se requiere mosix_link */
        mlink = kmalloc(sizeof(mosix_link), GFP_KERNEL);
        if (!mlink) {
        no_memory:
                goto failed;
        }
        mlink->flags = COMM_FULLLINK;
        comm_setup_link(mlink, peer);
} else {
        mlink = kmalloc(sizeof(struct socket *) + sizeof(int),
                                                        GFP_KERNEL);
        if (!mlink)
                goto no_memory;
        mlink->flags = COMM_INFOLINK;
}
mlink->sock = sock;

if ((error = comm_setup_socket(sock, mos)))
        goto failed;

if (!connect) {     /* configuracion para mantenerse a la escucha de comunicaciones */
        if(bind)
        {
                error = sock->ops->bind(sock, saddr, sizeof(*saddr));
                if (error) {
                        if(error == -EADDRINUSE && mos == COMM_MIGD)
                          printk("Migration port (%d) Already in use\n", ntohs(MIG_DAEMON
                        goto failed;
                }
        }
        if (listen) {                   /* se solicita una comunicacion? */
                error = sock->ops->listen(sock, SOMAXCONN);
                if (error) {
                        goto failed;
                }
        }

        if (mos == COMM_ACCEPT) {
                mlink->flags |= COMM_WAITACCEPT;
```

```
                        if ((error = comm_getname(sock, saddr))) {
                                goto failed;
                        }
                }
        } else {                /* configuracion para iniciar una comunicacion */
                if (!timo)
                        timo = MAX_SCHEDULE_TIMEOUT;
                error = sock->ops->connect(sock, saddr, sizeof(*saddr),
                                                        O_NONBLOCK);
                add_wait_queue(sock->sk->sleep, &wait);
                while (sock->state != SS_CONNECTED) {
                        set_current_state(TASK_INTERRUPTIBLE);
                        error = sock->ops->connect(sock, saddr, sizeof(*saddr),
                                                        O_NONBLOCK);
                        if (error != -EALREADY || (error=sock_error(sock->sk)))
                                break;

                        timo = schedule_timeout(timo);
                        if (timo <= 0) {
                                error = -EAGAIN;
                                break;
                        }
                }
                remove_wait_queue(sock->sk->sleep, &wait);
                set_current_state(TASK_RUNNING);

                if (error) {
                        goto failed;
                }
                if (sock->sk->err) {
                        error = sock_error(sock->sk);
                        goto failed;
                }
                /* socket del cliente ready */
        }
        ENABLE_EVENTS();

        return (mlink);

failed:
        ENABLE_EVENTS();
        if (sock)
                sock_release(sock);
        if (mlink)
                kfree(mlink);
        return (0);
}
```

`mosix_link *comm_close()`

Esta función cierra el socket abierto para una comunicacion establecida con anterioridad.

```
void
comm_close(mosix_link *mlink)
{
```

```
        unsigned int ours = 0;

        if (!mlink) {  /* si la funcion se llama con NULL */
                ours = 1;
                mlink = current->mosix.contact;
        }

        comm_shutdown(mlink);  /* termina la comunicacion con este canal */
        sock_release(mlink->sock);

        if (ours)
                current->mosix.contact = NULL;
                  /* apunta el contacto del proceso en curso a NULL */

        kfree(mlink);  /*libera la capa de enlace */
}
```

### 6.7.6. *config.c*

En este fichero se implementan las rutinas de comprobación de la coherencia del sistema openMosix, ya sea en el tipo de direcciones Ip para los nodos, los ID utilizados para cada uno de ellos y, sobretodo y lo que seguidamente se detalla, el *shutdown* de un nodo.

`static int config_shutdown(void)`

Cuando se indica a linux que el nodo debe apagarse, openmosix deberá finalizar el servicio. Apagar un nodo de modo ilegal puede conllevar la pérdida irreparable de los procesos que en él hubieran migrado.

```
static int
config_shutdown(void)
{
        struct mosixnet *svmton;
        int svnmton, svpe, svnpe, svmaxpe;
        int error;
        struct task_struct *t1, *t2;

        /* temporarily change configuration */
        lock_mosix();
        svpe = PE;
        svnpe = NPE;
        svmaxpe = MAXPE;
        svmton = mosnet;
        svnmton = nmosnet;
        mosnet = NULL;
        nmosnet = 0;
        NPE = MAXPE = 0;
        unlock_mosix();

        /* retornamos los procesos migrados y expedimos los ajenos */
        if ((error = config_validate()))
                goto fail;
        lock_mosix();
        PE = 0;
        unlock_mosix();
        info_reconfig();
```

```
        /* detenemos los daemons */
        lock_mosix();
        if((t1 = info_proc))
                get_task_struct(t1);
        if((t2 = mig_proc))
                get_task_struct(t2);
        unlock_mosix();
        if(t1)
        {
                send_sig(SIGALRM, t1, 1);
                free_task_struct(t1);
        }
        if(t2)
        {
                send_sig(SIGALRM, t2, 1);
                free_task_struct(t2);
        }

        lock_mosix();
        while ((mig_daemon_active || info_daemon_active) &&
                                                !signal_pending(current))
        {
                unlock_mosix();
                set_current_state(TASK_INTERRUPTIBLE);
                schedule_timeout(HZ/10);
                lock_mosix();
        }
        unlock_mosix();
        set_current_state(TASK_RUNNING);
        if (signal_pending(current))
        {
                error = -ERESTART;
                goto fail;
        }

        printk(KERN_NOTICE "openMosix configuration disabled\n");
        unregister_reboot_notifier(&mosix_notifier);

        if (svnmton) {
                kfree(svmton);
                kfree(mosnetstat);
        }

        lock_mosix();
        pe_ready = 0;
        unlock_mosix();
#ifdef CONFIG_MOSIX_FS
        mfs_change_pe();
#endif /* CONFIG_MOSIX_FS */

        return (0);

fail:
        lock_mosix();
```

```
        PE = svpe;
        NPE = svnpe;
        MAXPE = svmaxpe;
        mosnet = svmton;
        nmosnet = svnmton;
        unlock_mosix();
        wake_up(&wait_for_mosix_config);
        return (error);
}
```

### 6.7.7. *load.c*

Este fichero está ocupado casi en su totalidad por la función que se detalla. Otra función existente es la que resetea la carga del nodo, inicializando su valor al mínimo.

Véase como se calcula la carga de un nodo en función de la ocupación de sus recursos.

`void mosix_calc_load(unsigned long unused)`

```
void
mosix_calc_load(unsigned long unused)
{
        struct task_struct *p;
        register struct mosix_task *m;
        register int ladd, cpu, ticks;
        unsigned long flags;
        unsigned long newload;
#ifdef CONFIG_MOSIX_LOADLIMIT
#if 0
        static int ii=0, iii=0;
#endif
        unsigned long loadremote, loadlocal, cpulocal, cpuremote;
        int ii;
#endif
        int new_exploads;
        unsigned new_cpuse;
        unsigned new_came;
        static unsigned upper_load;    /* load estimado */
        static unsigned accload;       /* load acumulado */
        static int display_counter = 0;
#ifdef CONFIG_MOSIX_RESEARCH
        unsigned int new_io_read;
        unsigned int new_io_write;
        unsigned int major;
        unsigned int disk;
#endif /* CONFIG_MOSIX_RESEARCH */

        ticks = load_ticks;
        cpu = cpuse;
        ladd = load_adder;
        cpuse = load_adder = load_ticks = 0;

        ladd = ladd * ((long long)(MF * STD_SPD)) /
                (ticks * cpuspeed * smp_num_cpus);
        if(ladd * 128 > accload)        /* poco aumento de carga */
                accload = accload * DECAY + ladd * 128 * NEWDATA;
```

```
        else                            /* rapido descenso de carga */
                accload = ladd * 128;
        if(ladd >= upper_load)          /* rapido aumento de carga */
                upper_load = ladd;
        else                            /* very slowly down */
                upper_load = (upper_load * 7 + ladd) / 8;
        new_cpuse = (acpuse * 3 + cpu * MF / ticks + 3) / 4;
        newload = (accload+64) / 128;
        new_expload = (upper_load + stable_export +
                came_lately4 * MF * STD_SPD /
                        (4 * cpuspeed * smp_num_cpus)) *
                        MF * smp_num_cpus / new_cpuse;
        if(newload < load_left)
                newload = 0;
        else
                newload -= load_left;
        newload = newload * MF * smp_num_cpus / new_cpuse;
        new_came = came_lately4 * DECAY + coming_in * 4 * NEWDATA;
        spin_lock_irqsave(&runqueue_lock, flags);
#ifdef CONFIG_MOSIX_LOADLIMIT
        loadlocal = loadremote = cpulocal = cpuremote = 0;
#if 0
        ii = (ii+1) & 0xf;
#endif
#endif
        for_each_task(p)
        {
                m = &p->mosix;
                if(m->runstart)
                {
                        m->ran += ticks + 1 - m->runstart;
                        m->runstart = 1;
                }
                m->load = m->load * DECAY + m->ran * MF * 4*NEWDATA/ticks;
#ifdef CONFIG_MOSIX_LOADLIMIT
                if (m->dflags & DREMOTE)
                  {
                    if (m->last_per_cpu_utime[0])
                      {
                        /* this means that at least one time we were already here */
                        for (ii = 0;  ii < smp_num_cpus; ii++)
                          {
                            cpuremote += p->per_cpu_utime[ii] -
                              m->last_per_cpu_utime[ii];
                            cpuremote += p->per_cpu_stime[ii] -
                              m->last_per_cpu_stime[ii];
                          }
                      }
                    loadremote += m->load;
                  }
                else
                  {
                    if (m->last_per_cpu_utime[0])
                      {
                        for (ii = 0 ; ii < smp_num_cpus; ii++)
```

```
                        {
                          cpulocal += p->per_cpu_utime[ii] -
                            m->last_per_cpu_utime[ii];
                          cpulocal += p->per_cpu_stime[ii] -
                            m->last_per_cpu_stime[ii];
                        }
                     }
                   loadlocal += m->load;
                 }
                 for (ii = 0;  ii < smp_num_cpus; ii++)
                   {
                     m->last_per_cpu_utime[ii] = p->per_cpu_utime[ii];
                     m->last_per_cpu_stime[ii] = p->per_cpu_stime[ii];
                   }

        #if 0
                if (!ii)
                  printk("PID: %d m->load: %d m->ran: %d ticks: %d ranstart:%d MF:%d\n",
                         p->pid, m->load, m->ran, ticks, m->runstart, MF);
#endif
#endif
                m->ran = 0;
                m->page_allocs >>= 1;   /* decay in time */
        }
        spin_unlock_irqrestore(&runqueue_lock, flags);

        if(Tvis)
                printk("\0337\033[22;55HL=%d,E=%d,R=%d,U=%d  \0338",
                        (int)newload, new_exploa d, mosix_running,
                        new_cpuse);
        if(Tload) {
                if (!(display_counter = (display_counter + 1) & 0xf))
                        printk("\naccload upper_load\tload_adder\tload_ticks\n");
                printk("%7d\t%10d\t%10d\t%d\n",
                        accload, upper_load, ladd, ticks);
        }
        write_lock(&loadinfo_lock);
        loadinfo[0].load = newload;
#ifdef CONFIG_MOSIX_LOADLIMIT
        loadinfo[0].loadremote = loadremote;
        loadinfo[0].loadlocal = loadlocal;
        loadinfo[0].cpulocal = cpulocal*MF / ticks;
        loadinfo[0].cpuremote = cpuremote*MF / ticks;
#if 0
        if (!ii)
          {
            iii++;
            printk("loadlocal: %lu loadremote: %lu load: %lu\n", loadlocal, loadremote,
                   newload);
          }
#endif
#endif
        export_load = new_exploa d;
        acpuse = new_cpuse;
        came_lately4 = new_came;
```

```
        load_left = 0;

        write_unlock(&loadinfo_lock);

        if((p = (struct task_struct *)info_proc))
                send_sig(SIGALRM, p, 1);
}
```

### 6.7.8. *remote.c*

El *remote* permanece en un nodo remoto atento constantemente a la comunicación que pueda solicitar *deputy*. En esta función, la principal, pueden verse todos los tipos de peticiones que reconoce el elemento migrado del proceso.

`int remote_wait()`

```
int
remote_wait(int expect, void **head, int *hlen)
{
        int type, error = 0;
        register struct task_struct *p = current;
        struct syscall_ret_h *rp;
        unsigned int has_ret_value = 0;
        int ret_value = 0;      /* value set only to pacify compiler */

        while(1)
        {
                if(URGENT_REMOTE_CONDITIONS(p) &&
                                        !(p->mosix.dflags & DSENTURGENT))
                        inform_deputy_of_urgent();
                if((type = comm_recv(head, hlen)) < 0)  /* an error */
                        return(type);
                if(expect == DEP_USERMODE)
                /* after migration from REMOTE to REMOTE, certain "replies"
                 * can arrive as a result from the original call.
                 * Fortunately, there are not too many of them:
                 */
                        switch(type & ~USERMODE)
                        {
                        case REM_SYSCALL|REPLY:
                                rp = *head;
                                absorb_deptime(rp->deputytime);
                                remote_unpack_read_cache_data(rp);
                                has_ret_value = 1;
                                ret_value = rp->ret;
                                /* fall through */
                        case REM_SYSCALL_TRACE|REPLY:
                                if(!(type & USERMODE))
                                {
                                        comm_free(*head);
                                        *head = NULL;
                                        continue;
                                }
                        }
                if(type & USERMODE)
```

```
{
        p->mosix.dflags &= ~DPSYNC;
        if(expect == DEP_USERMODE)
        {
                if(has_ret_value)
                        mos_to_regs(&p->mosix)->eax = ret_value;
                return(0);
        }
        if(type != (expect | USERMODE))
        {
                printk("REMOTE-%d: Unexected USERMODE while waiting for 0x%x\n",
                        p->pid, expect);
                mosix_panic("Unexpected USERMODE");
                return(-EINVAL);
        }
}
if((type & ~USERMODE) == expect)
        return(0);
switch(type & ~USERMODE)
{
case DEP_SYNC:
        comm_free(*head);
        break;
case DEP_NOTHING:
        comm_free(*head);
        error = comm_send(DEP_NOTHING|REPLY, NULL, 0,
                          NULL, 0, 0);
        break;
case DEP_MMAP:
        error = remote_mmap((struct mmap_parameters_h *)*head, 0);
        break;
case DEP_BRK:
        error = remote_brk((struct brk_parameters_h *)*head);
        break;
case DEP_MUNMAP:
        error = remote_munmap((struct munmap_parameters_h *)*head);
        break;
case DEP_MPROTECT:
        error = remote_mprotect((struct mprotect_parameters_h *)*head);
        break;
case DEP_LISTHOLD:
        comm_free(*head);
        error = remote_report_files();
        break;
case DEP_SETUPFRAME:
        error = remote_setup_frame((struct setupframe_parameters_h *)*head);
        break;
case DEP_NICE:
        error = remote_nice((long *)*head);
        break;
case DEP_INFO:
        error = remote_updinfo((struct disclosure_h *)*head);
        break;
case DEP_CAPS:
        error = remote_caps((kernel_cap_t *)*head);
```

```
                break;
        case DEP_OPCOSTS:
                error = remote_depcosts(*head);
                break;
        case DEP_RESTORESIGCONTEXT:
                error = remote_restore_sigcontext((struct sigcontext **)*head);
                break;
        case DEP_PREQUEST:
                error = remote_prequest((struct prequest_h *)*head);
                break;
        case DEP_COPY_FROM_USER:
                error = remote_copy_from_user((struct user_copy_h *)*head);
                break;
        case DEP_COPY_TO_USER:
                error = remote_copy_to_user((struct user_copy_h *)*head);
                break;
        case DEP_DATA_TO_USER:
                error = remote_data_to_user((struct user_copy_h *)*head);
                break;
        case DEP_CLEAR_USER:
                error = remote_clear_user((struct user_copy_h *)*head);
                break;
        case DEP_STRNCPY_FROM_USER:
                error = remote_strncpy_from_user((struct user_copy_h *)*head);
                break;
        case DEP_STRNLEN_USER:
                error = remote_strnlen_user((struct strnlen_user_h *)*head);
                break;
        case DEP_VERIFY_WRITE:
                error = remote_verify_write((struct user_copy_h *)*head);
                break;
        case DEP_CSUM_COPY_FROM_USER:
                error = remote_csum_copy_from_user((struct user_csum_copy_h *)*head);
                break;
        case DEP_CACHE_READ_DATA:
                error = remote_unpack_read_cache_data(NULL);
                /* NO REPLY! */
                break;
        case DEP_RLIMIT:
                error = remote_rlimit((struct rlimit_h *)*head);
                break;
        case DEP_TAKEURGENT:
                comm_free(*head);
                error = remote_urgent();
                break;
        case DEP_RUSAGE:
                error = remote_rusage((int *)*head);
                break;
        case DEP_PERSONALITY:
                error = remote_personality((unsigned long *)*head);
                break;
        case DEP_EXECVE_COUNTS:
                error = remote_execve_counts((struct execve_counts_h *)*head);
                break;
        case DEP_BRING_STRINGS:
```

```
                error = remote_bring_strings((struct execve_bring_strings_h *)*head);
                break;
        case DEP_SETUP_ARGS:
                error = remote_setup_args((struct execve_setup_args_h *)*head);
                break;
        case DEP_EXEC_MMAP:
                error = remote_exec_mmap();
                break;
        case DEP_INIT_AOUT_MM:
                error = remote_init_aout_mm((struct exec *)*head);
                break;
        case DEP_ELF_SETUP:
                error = remote_elf_setup((struct execve_elf_setup_h *)*head);
                break;
        case DEP_FIX_ELF_AOUT:
                error = remote_fix_elf_aout((struct execve_fix_elf_aout_h *)*head);
                break;
        case DEP_DUMP_THREAD:
                /* comm_free(*head); is not needed (NULL) */
                error = remote_dump_thread();
                break;
        case DEP_LIST_VMAS:
                /* comm_free(*head); is not needed (NULL) */
                error = remote_list_vmas();
                break;
        case DEP_PLEASE_FORK:
                error = remote_fork((struct fork_h *)*head);
                break;
        case DEP_BRING_ME_REGS:
                error = remote_bring_me_regs((unsigned long *)*head);
                break;
        case DEP_DUMP_FPU:
                /* comm_free(*head); is not needed (NULL) */
                error = remote_dump_fpu();
                break;
        case DEP_COME_BACK:
                if (!remote_come_back(*head) || bootexpel)
                        remote_disappear();
                break;
        case DEP_PLEASE_MIGRATE:
                error = remote_goto_remote(*head);
                break;
        case DEP_CONSIDER:
                error = remote_consider((int *)*head);
                break;
        case DEP_UPDATE_DECAY:
                error = remote_setdecay((struct decay_h *)*head);
                break;
        case DEP_UPDATE_LOCK:
                error = remote_set_lock((int *)*head);
                break;
        case DEP_UPDATE_MIGFILTER:
                error = remote_set_migfilter((int *)*head);
                break;
        case DEP_UPDATE_MIGPOLICY:
```

```
                        error = remote_set_migpolicy((int *)*head);
                        break;
                case DEP_UPDATE_MIGNODES:
                        error = remote_set_mignodes((int *)*head);
                        break;
                case DEP_PSINFO:
                        /* comm_free(*head); is not needed (NULL) */
                        error = remote_psinfo();
                        break;
#ifdef CONFIG_MOSIX_DFSA
                case DEP_DFSA_CLEAR:
                        remote_clear_dfsa();
                        error = comm_send(DEP_DFSA_CLEAR|REPLY, NULL, 0,
                                        NULL, 0, 0);
                        break;
                case DEP_DFSA_CHANGES:
                        error = remote_receive_dfsachanges((int *)*head);
                        break;
                case DEP_READ_YOURSELF:
                        error = remote_read_yourself(
                                (struct read_yourself_h *)*head);
                        break;
#endif /* CONFIG_MOSIX_DFSA */
                default:
                        printk("openMosix: remote type %x unexpected\n", type);

                        if(type != MIG_REQUEST)
                                mosix_panic("deputy_wait");
                        return(-EINVAL);
                }
                if(error)
                {
                        return(error);
                }
        }
}
```

## 6.8. ./include/*

The include subdirectory contains most of the include files needed to build the kernel code. It too has further subdirectories including one for every architecture supported. The include/asm subdirectory is a soft link to the real include directory needed for this architecture, for example include/asm-i386. To change architectures you need to edit the kernel makefile and rerun the Linux kernel configuration program.

```
patching file include/asm-generic/smplock.h
patching file include/asm-i386/a.out.h
patching file include/asm-i386/checksum.h
patching file include/asm-i386/cpufeature.h
patching file include/asm-i386/elf.h
patching file include/asm-i386/fcntl.h
patching file include/asm-i386/i387.h
patching file include/asm-i386/processor.h
patching file include/asm-i386/rwsem.h
patching file include/asm-i386/semaphore.h
patching file include/asm-i386/siginfo.h
patching file include/asm-i386/smplock.h
patching file include/asm-i386/spinlock.h
patching file include/asm-i386/stat.h
patching file include/asm-i386/uaccess.h
patching file include/hpc/balance.h
patching file include/hpc/comm.h
patching file include/hpc/debug.h
patching file include/hpc/defs.h
patching file include/hpc/dfsa.h
patching file include/hpc/dscosts.h
patching file include/hpc/hpcctl.h
patching file include/hpc/hpctask.h
patching file include/hpc/hpcversion.h
patching file include/hpc/mfscosts.h
patching file include/hpc/protocol.h
patching file include/hpc/request.h
patching file include/hpc/routines.h
patching file include/linux/binfmts.h
patching file include/linux/capability.h
patching file include/linux/completion.h
patching file include/linux/dcache.h
patching file include/linux/dfsa_interface.h
patching file include/linux/elf.h
patching file include/linux/errno.h
patching file include/linux/file.h
patching file include/linux/fs.h
patching file include/linux/fs_struct.h
patching file include/linux/hpcctl.h
patching file include/linux/hpc.h
patching file include/linux/iobuf.h
patching file include/linux/keyboard.h
patching file include/linux/linkage.h
patching file include/linux/mfs_fs_i.h
patching file include/linux/mfs.h
patching file include/linux/mfs_socket.h
patching file include/linux/mman.h
patching file include/linux/mm.h
patching file include/linux/mount.h
patching file include/linux/net.h
patching file include/linux/personality.h
patching file include/linux/pipe_fs_i.h
patching file include/linux/proc_fs.h
patching file include/linux/remote_fs_i.h
patching file include/linux/sched.h
patching file include/linux/smp_lock.h
patching file include/linux/spinlock.h
patching file include/linux/wait.h
patching file include/net/sock.h
```

### 6.8.1. *hpc.h*

```
#ifndef _LINUX_MOSIX_H
#define _LINUX_MOSIX_H

#ifdef CONFIG_MOSIX

#ifdef __KERNEL__

#include <linux/sched.h>
#include <linux/user.h>
#include <linux/elf.h>
#include <hpc/defs.h>
#include <hpc/request.h>

/* operations on DEPUTY's data-base of REMOTE files: */

extern int mosix_register_a_file(struct file *, int);
extern void mosix_undo_last_file_registration(struct file *, int);
extern int mosix_rebuild_file_list(void);
extern void mosix_update_remote_files(void);

struct vmalist
{
        unsigned int vmstart;
        unsigned int vmend;
        unsigned short vmflags;
        short   maydump;
};

struct vmamaps
{
        unsigned long vmstart, vmend, vmflags, vmpgoff;
        struct file *fp;
};

extern void init_mosix(void);
extern void wake_up_mosix(struct task_struct *);
extern int mosix_wakeable(struct task_struct *);
extern int mosix_go_home(int);
extern int mosix_go_home_for_reason(int, int);
extern int mosix_send_back_home(struct task_struct *);
extern int mosix_need_while_asleep(void);
extern void mosix_run_while_asleep(void);
extern void mosix_pre_usermode_functions(void);
extern int stay_me_and_my_clones(uint32_t);
extern void unstay_mm(struct mm_struct *);
extern void mosix_inform_remote_of_nice(void);
extern void mosix_snap_load(int);
extern void mosix_clear_statistics(void);
extern int mosix_fork_init_fields(struct task_struct *);
extern void mosix_fork_free_fields(struct task_struct *);
extern void mosix_exit(void);
extern void mosix_very_exit(void);
extern void mosix_obtain_registers(unsigned long);
extern void mosix_bring_monkey_users_back(struct inode *);
```

```
extern void mosix_no_longer_monkey(struct inode *);
extern void mosix_check_for_freedom_to_move(void);
extern int mosix_pre_clone(void);
extern void mosix_post_clone(void);
extern void mosix_remote_syscall_trace(void);
extern u64 mosix_remote_tsc(void);
extern int mosix_forkmigrate(void);
extern int mosix_deputy_fork(struct task_struct *, int, unsigned long);
extern void mosix_exit_mm(struct task_struct *);
extern void mosix_exec_mmap(struct mm_struct *);
extern void mosix_deputy_rusage(int);
extern int mosix_deputy_personality(unsigned long);
extern void mosix_deputy_count_args(char **, char**, int *, int *);
extern int mosix_deputy_bring_strings(struct linux_binprm *, char *, char **, char **);
extern int mosix_deputy_setup_args(int, unsigned long *);
extern int mosix_deputy_exec_mmap(char *);
extern int mosix_deputy_dump_thread(struct user *);
extern void mosix_deputy_init_aout_mm(struct exec *);
extern unsigned long mosix_deputy_elf_setup(char *, int, int, struct elfhdr *, unsigned long, unsi
extern void mosix_deputy_fix_elf_aout_interp(unsigned, unsigned, unsigned);
extern int mosix_deputy_list_vmas(struct vmalist **, unsigned long *, unsigned long *);
extern long mosix_deputy_brk(unsigned long, unsigned long);

extern void mosix_notify_urgent(struct socket *);
extern void mosix_notify_receive(struct socket *);

extern void mosix_proc_init(void);
extern int mosix_proc_get_remote_array(char *, int, int);
extern int mosix_proc_get_node_array(char *, int, int);

extern void mosix_decay_exec(void);
extern void mosix_add_to_whereto(struct task_struct *, int);
extern void mosix_do_add_to_whereto(struct task_struct *, int);

struct sockaddr;
extern int reserved_mosix_address(struct sockaddr *);
extern void comm_report_violation(char *, struct sockaddr *);

void init_guest_user_struct(void);
extern int get_free_guest_slots(void);
extern int count_guests(void);
#ifdef CONFIG_MOSIX_FS
int mfs_client_statfs(int, mfs_handle_t, struct statfs *);
#endif /* CONFIG_MOSIX_FS */
/* argument of mosix_deputy_setup_args: */
#define SETUP_ARGS_PLAIN        0
#define SETUP_ARGS_AS_AOUT      1
#define SETUP_ARGS_AS_ELF       2
#define SETUP_ARGS_NOTYET       3

extern int mosix_deputy_restore_sigcontext(struct sigcontext *, int *);
extern void mosix_deputy_setup_frame(unsigned long, struct k_sigaction *, siginfo_t, sigset_t *);
extern unsigned long mosix_deputy_mmap(struct file *, unsigned long, int, unsigned long, unsigned
extern int deputy_munmap(unsigned long, size_t);
extern int deputy_mprotect(unsigned long, size_t, unsigned long);
```

```
extern void mosix_deputy_rlimit(int, struct rlimit);
extern int mosix_deputy_dump_fpu(struct user_i387_struct *);
extern int mosix_sync_caps(kernel_cap_t);

#define ALL_REGISTERS   0x7fff  /* as many as in struct pt_regs */
#define BIT_OF_REGISTER(_which) \
        (1 << (((int)&(((struct pt_regs *)0)->_which)) / sizeof(int)))

/*
 * meaning of signals on REMOTE
 */

#define FATAL_SIGSEGV           SIGINT
#define REMOTE_FILE_RELEASED    SIGQUIT

/* other signals that can occur on REMOTE:
 * SIGKILL, SIGSEGV, SIGPROF, SIGVTALRM, SIGFPE, SIGBUS, SIGIOT, SIGILL,
 * SIGXCPU, SIGXFSZ, SIGPROF, SIGTRQAP, SIGPWR (a tricky one).
 * SIGTERM requires the process to migrate back.
 */

#define MOSIX_PRIORITY          (100)
#define MOSIX_ASLEEP_PRIORITY   (MOSIX_PRIORITY * 3 / 10)
#define MOSIX_DEPUTY_PRIORITY   (MOSIX_PRIORITY * 1 / 10)
#define MOSIX_RESPOND_PRIORITY  (MOSIX_PRIORITY * 6 / 10)

#define PROC_MOSIX_USE_START    1024    /* coordinate with proc/fs/base.c */

#if defined(CONFIG_MOSIX_DIAG) && defined(CONFIG_SMP)
#define KERNEL_LOCKED   do{if(current->lock_depth == -1)panic("not locked %d of " __FILE__, __LINE_
#define MOSIX_LOCKED    do{if(current->mosix.lock_depth == -1)panic("not locked %d of " __FILE__, _
#else
#define KERNEL_LOCKED   do {} while(0)
#define MOSIX_LOCKED    do {} while(0)
#endif /* CONFIG_MOSIX_DIAG */

#define mos_to_contact(m) ((struct socket *)((m)->contact))
#define mos_to_waitp(m) ((wait_queue_head_t *)(&(m)->wait_dist))
#define mos_to_regs(m)  ((struct pt_regs *)((m)->altregs))

struct proc_dir_entry;
extern struct proc_dir_entry *proc_mosix;
typedef ssize_t (proc_mosix_pid_writer)(struct file *, struct task_struct *, const char *, size_t);
extern proc_mosix_pid_writer proc_mosix_pid_set_migrate;
extern proc_mosix_pid_writer proc_mosix_pid_set_goto;
extern proc_mosix_pid_writer proc_mosix_pid_set_lock;
extern proc_mosix_pid_writer proc_mosix_pid_set_sigmig;
extern proc_mosix_pid_writer proc_mosix_pid_set_disclosure;
extern proc_mosix_pid_writer proc_mosix_pid_set_migfilter;
extern proc_mosix_pid_writer proc_mosix_pid_set_migpolicy;
extern proc_mosix_pid_writer proc_mosix_pid_set_mignodes;
extern proc_mosix_pid_writer proc_mosix_pid_set_miggroup;
#ifdef CONFIG_MOSIX_FS
extern proc_mosix_pid_writer proc_mosix_pid_set_selected;
#endif /* CONFIG_MOSIX_FS */
```

```
#ifdef CONFIG_MOSIX_UDB
extern void udbinit(void);
extern int udb_booting;
extern int nmi_debugger;
#endif /* CONFIG_MOSIX_UDB */

#endif /*__KERNEL__*/

#endif /* CONFIG_MOSIX */

#endif
```

## 6.9. ./init/*

```
This directory contains the initialization code for the kernel and it
is a very good place to start looking at how the kernel works.

patching file init/main.c
```

### 6.9.1. *main.c*

Este fichero es el que inicia en cualquier sistema linux el proceso con pid 1, el proceso *init*. Se definen las primeras variables de arranque y seguidamente se invoca a tres funciones importantes.

1. `extern void prepare_namespace()` . Prepara el *namespace*, es decir, decide qué y donde montar, cargar los *ramdisks*, *etc.* Su implementación puede encontrarse en */init/do_mounts.c* y no sufre modificaciones por el parche de openMosix.
2. `static int init(void * unused)` . Ejecuta los primeros procesos del sistema. Esto incluye también a la función `init_mosix()` cuya implementación se encuentra en *./hpc/init.c* .
3. `init_mosix()` . Véanse comentarios en la implementación.

`extern void prepare_namespace()`

```
void prepare_namespace(void)
{
        int is_floppy = MAJOR(ROOT_DEV) == FLOPPY_MAJOR;
#ifdef CONFIG_ALL_PPC
        extern void arch_discover_root(void);
        arch_discover_root();
#endif /* CONFIG_ALL_PPC */
#ifdef CONFIG_BLK_DEV_INITRD
        if (!initrd_start)
                mount_initrd = 0;
        real_root_dev = ROOT_DEV;
#endif
        sys_mkdir("/dev", 0700);
        sys_mkdir("/root", 0700);
        sys_mknod("/dev/console", S_IFCHR|0600, MKDEV(TTYAUX_MAJOR, 1));
#ifdef CONFIG_DEVFS_FS
        sys_mount("devfs", "/dev", "devfs", 0, NULL);
        do_devfs = 1;
#endif

        create_dev("/dev/root", ROOT_DEV, NULL);
        if (mount_initrd) {
                if (initrd_load() && ROOT_DEV != MKDEV(RAMDISK_MAJOR, 0)) {
                        handle_initrd();
                        goto out;
                }
        } else if (is_floppy && rd_doload && rd_load_disk(0))
                ROOT_DEV = MKDEV(RAMDISK_MAJOR, 0);
        mount_root();
out:
        sys_umount("/dev", 0);
        sys_mount(".", "/", NULL, MS_MOVE, NULL);
        sys_chroot(".");
        mount_devfs_fs ();
}
```

`static int init(void * unused)`

```
static int init(void * unused)
{
        lock_kernel();
        do_basic_setup();

        prepare_namespace();

        free_initmem();
#ifdef CONFIG_MOSIX
        init_mosix();
#endif /* CONFIG_MOSIX */
        unlock_kernel();

        if (open("/dev/console", O_RDWR, 0) < 0)
                printk("Warning: unable to open an initial console.\n");

        (void) dup(0);
        (void) dup(0);

        if (execute_command)
                execve(execute_command,argv_init,envp_init);
        execve("/sbin/init",argv_init,envp_init);
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/bin/sh",argv_init,envp_init);
        panic("No init found.  Try passing init= option to kernel.");
}
```

### init_mosix()

```
void
init_mosix(void)
{
        extern int x86_udelay_tsc;
        cpuspeed = ((int64_t)loops_per_jiffy) * STD_SPD / STD_LOOPS; /*
velocidad del nodo */
```

La constante de openMosix *STD_SPD* debe su nombre a *standard speed* y se define en *./include/hpc/defs.h* como 15000.

```
        if(!x86_udelay_tsc)
                cpuspeed *= 2;
```

La constante `x86_udelay_tsc` se define en *./lib/delay.c* como el entero de valor 0 -cero-. Como puede leerse, si el *delay* entre dos lecturas al registro TSC es cero, se dobla el valor de velocidad tomado para el nodo donde se inicialice el servicio openMosix.

El TimeStamp Counter (TSC) es un valor que se toma de un registro que poseen todos los nuevos procesadores. Este registro se incrementa cada ciclo de reloj. Leyendo 2 veces el registro y dividiendo la diferencia de ciclos obtenida entre el intervalo de tiempo transcurrido, se obtiene la frecuencia de reloj del procesador.

```
        info_init();
```

Esta función está implementada en *./hpc/info.c*. Define los valores para:

- número de procesadores

- tamaño de memoria
- invoca a `set_my_cpuspeed()` para definir los costes tanto de carga como para el sistema de ficheros mfs. Se encuentra implementada también en *./hpc/info.c* .

```
        proc_update_costs();

#ifdef CONFIG_MOSIX_FS
        init_mfs();             /* si el kernel soporta MFS, se inicia */
#endif /* CONFIG_MOSIX_FS */
#ifdef CONFIG_MOSIX_DFSA
        dfsa_init();            /* si el kernel soporta DFSA, se inicia */
#endif /* CONFIG_MOSIX_DFSA */
        kernel_thread(mosix_mig_daemon, NULL, 0);  /* se inicia el daemon de  migracion */
                                                   /* como hebra del kenel             */
        info_startup();
```

Esta función también está implementada en *./hpc/info.c* . Toma una ventana de información del cluster -i.e. varios nodos, pueden no ser todos- y evalá su ponderación.

```
        mosix_load_init();          /* actualiza la carga instantanea del nodo */
        mosinfo_update_gateways();  /* actualiza costes de envios entre nodos */
        kernel_thread(mosix_info_daemon, NULL, 0);   /* daemon de informacion del sistema */
        kernel_thread(mosix_mem_daemon, NULL, 0);    /* daemon para la memoria */
}
```

## 6.10. ./ipc/*

This directory contains the kernels inter-process communications code.

patching file ipc/shm.c

### 6.10.1. *shm.c*

Los cambios realizados en este fichero atienden a los problemas de openMosix en sistemas con el dispositivo virtual montado en */dev/shfs* en sistemas de fichero reiser.

la solución hasta ahora pasaba por desmontar tal unidad, puesto que según el mismo Moshe Bar solo el 99 % de los usuarios de linux realmente la necesitan. no obstante, y pensando en las minorías, se ha arreglado este bug.

Para más detalle puede consultarse el fichero.

## 6.11. ./kernel/*

```
The main kernel code. Again, the architecture specific kernel code is
in arch/*/kernel.

patching file kernel/acct.c
patching file kernel/capability.c
patching file kernel/exec_domain.c
patching file kernel/exit.c
patching file kernel/fork.c
patching file kernel/module.c
patching file kernel/panic.c
patching file kernel/ptrace.c
patching file kernel/sched.c
patching file kernel/signal.c
patching file kernel/sys.c
patching file kernel/sysctl.c
patching file kernel/timer.c
patching file kernel/user.c
```

## 6.12. ./lib/*

This directory contains the kernel's library code.

```
patching file lib/rwsem.c
patching file lib/rwsem-spinlock.c
```

### 6.12.1. *rwsem.c*

Este fichero contiene las funciones para la gestión de los bloqueos de los semáforos de lectura y/o escritura *-contention handling functions for R/W semaphores-*.

Una de las estructuras pasadas por parámetro es `sem` de tipo `rw_semaphore`. Esta estructura se define en Linux dentro del fighero *./include/asm-i386*, así

```
struct rw_semaphore {
        signed long             count;
#define RWSEM_UNLOCKED_VALUE            0x00000000
#define RWSEM_ACTIVE_BIAS               0x00000001
#define RWSEM_ACTIVE_MASK               0x0000ffff
#define RWSEM_WAITING_BIAS              (-0x00010000)
#define RWSEM_ACTIVE_READ_BIAS          RWSEM_ACTIVE_BIAS
#define RWSEM_ACTIVE_WRITE_BIAS         (RWSEM_WAITING_BIAS + RWSEM_ACTIVE_BIAS)
        spinlock_t              wait_lock;
        struct list_head        wait_list;
#if RWSEM_DEBUG /* to be removed shortly */
        int                     debug;
#endif
};
```

Otra de las estructuras pasadas por parámetro es `rwsem_waiter`, donde se incluyen los campos necesarios para identificar al semáforo ya sea de lectura o escritura, el proceso que lo invoca y unos *flags* para opciones de control. Véase su implementación:

```
struct rwsem_waiter {
        struct list_head        list;
        struct task_struct      *task;
        unsigned int            flags;
#define RWSEM_WAITING_FOR_READ  0x00000001
#define RWSEM_WAITING_FOR_WRITE 0x00000002
};
```

Las modificaciones que realiza openMosix sobre el fichero *rwsem.c* se centran en un pequeño cambio a la funcion `rw_semaphore *rwsem_down_failed_common()`. Esta función espera para que se conceda el bloqueo de un semaforo[5]. Se añade una línia que invoca a la función `adjust_task_mosix_context()`. Esta función está definida en *./hpc/service.c* e implementada como sigue:

```
void
adjust_task_mosix_context(struct task_struct **tp)
{
        struct task_struct *t = *tp;

        if(t->mosix.dflags & DINSCHED)
                *tp = MOSIX_CONTEXT(t);
}
```

Esta implementación consigue modificar el `task_struct` que identifica al proceso en curso para que sea compatible con el entorno de operación de openMosix. MOSIX_CONTEXT se define en *./include/linux/wait.h* de forma que aplica al `task_struct` pasado como parámetro la máscara KERNEL_ADDRESS_BIT de la manera que sigue:

```
#define KERNEL_ADDRESS_BIT      0x80000000
#define MOSIX_CONTEXT(p)        ((struct task_struct *)\
                                (((unsigned long) (p)) & ~KERNEL_ADDRESS_BIT))
```

[5] La práctica demuestra como normalmente las peticiones de bloqueo nunca son servidas con inmediatez, por las propias operaciones que realizan lso procesos con los recursos de que se apropian.

Véase la función:

`static inline struct rw_semaphore *rwsem_down_failed_common()`

```
static inline struct rw_semaphore *rwsem_down_failed_common(struct rw_semaphore *sem,
                                                            struct rwsem_waiter *waiter,
                                                            signed long adjustment)
{
        struct task_struct *tsk = current;  /* se apunta 'tsk' al proceso en curso */
        signed long count;

        set_task_state(tsk,TASK_UNINTERRUPTIBLE);
                        /* se marca al proceso como ininterrumpible */
        spin_lock(&sem->wait_lock);   /* se pone la peticion en la lista de espera   */
        waiter->task = tsk;
#ifdef CONFIG_MOSIX
        adjust_task_mosix_context(&waiter->task);   /* se aplica la modificaion del contexto */
#endif /* CONFIG_MOSIX */

        list_add_tail(&waiter->list,&sem->wait_list);

        /* note that we're now waiting on the lock, but no longer actively read-locking */
        count = rwsem_atomic_update(adjustment,sem);

        /* if there are no longer active locks, wake the front queued process(es) up
         * - it might even be this process, since the waker takes a more active part
         */
        if (!(count & RWSEM_ACTIVE_MASK))
                sem = __rwsem_do_wake(sem);

        spin_unlock(&sem->wait_lock);

        for (;;) { /*bucle a la espera de la concesion de bloqueo */
                if (!waiter->flags)
                        break;
                schedule();
                set_task_state(tsk, TASK_UNINTERRUPTIBLE);
        }

        tsk->state = TASK_RUNNING;

        return sem;
}
```

### 6.12.2. *rwsem-spinlock.c*

Este fichero sigue implementando *handlers* como los del fichero anterior, pero para bloqueos implementados con las funciones spinlock.

Las modificaciones se realizan en la función encargada de controlar el bloqueo exclusivo de escritura sobre el recurso que solicite el proceso. La implementación es muy parecida a la comentada anteriormente. Véase:

`void __down_write()`

```
void __down_write(struct rw_semaphore *sem)
```

```
{
        struct rwsem_waiter waiter;
        struct task_struct *tsk;

        rwsemtrace(sem,"Entering __down_write");

        spin_lock(&sem->wait_lock);

        if (sem->activity==0 && list_empty(&sem->wait_list)) {
                /* bloqueo concedido */
                sem->activity = -1;
                spin_unlock(&sem->wait_lock);
                goto out;
        }

        tsk = current;
        set_task_state(tsk,TASK_UNINTERRUPTIBLE);

        waiter.task = tsk;
#ifdef CONFIG_MOSIX
        adjust_task_mosix_context(&waiter.task);
#endif /* CONFIG_MOSIX */
        waiter.flags = RWSEM_WAITING_FOR_WRITE;

        list_add_tail(&waiter.list,&sem->wait_list);

        spin_unlock(&sem->wait_lock);

        for (;;) {
                if (!waiter.flags)
                        break;
                schedule();
                set_task_state(tsk, TASK_UNINTERRUPTIBLE);
        }

        tsk->state = TASK_RUNNING;

 out:
        rwsemtrace(sem,"Leaving __down_write");
}
```

## 6.13. ./mm/*

This directory contains all of the memory management code.

```
patching file mm/filemap.c
patching file mm/memory.c
patching file mm/mlock.c
patching file mm/mmap.c
patching file mm/mprotect.c
patching file mm/mremap.c
patching file mm/oom_kill.c
patching file mm/page_alloc.c
patching file mm/shmem.c
patching file mm/vmscan.c
```

## 6.14. ./net/*

The kernel's networking code.

```
patching file net/802/cl2llc.c
patching file net/core/scm.c
patching file net/core/skbuff.c
patching file net/core/sock.c
patching file net/ipv4/af_inet.c
patching file net/ipv4/tcp_input.c
patching file net/sunrpc/sched.c
patching file net/sunrpc/svc.c
```

## 6.15. EL API DE OPENMOSIX

*The real danger is not that computers will begin to think like men,*
*but that men will begin to think like computers.*

*Sydney J. Harris*

El API de openMosix es la manera mediante la cual los programadores pueden controlar el comportamiento de un cluster. Sirve como herramienta de diagnóstico para monitorizar los aspectos más íntimos del procedimiento que se lleva a cabo en el seno del nodo y, por extensión, del cluster. De esta interfaz se obtienen los datos para `mosmon` por ejemplo, para monitorizar la carga -*load*- o las relaciones de memoria libre y ocupada.

No hará falta ser un experto para poder comprender esta interfaz, puesto que es de una simplicidad sorprendente: lo justo para poder saber en cualquier momento, cualquier aspecto relativo a la algorítmica que controla el cluster: migración, balenceo, *etc.*

De esta manera se aconseja al lector ver como se opera en este directorio para buscar las *razones* de migración, por ejemplo, o qué procesos no locales se están ejecutando en su nodo -aspectos que hasta ahora habían permanecido ocultos al ser invisivbles al comando `ps`-. Se pueden llegar a solucionar problemas o simplemente comprender mejor a openMosix.

En este apartado se describirá el API de openMosix, al que se puede acceder como a un conjunto de ficheros[6] y escribir o leer de ellos dependiendo de su utilización. Todos los ficheros se encuentran en */proc/hpc/*. Dentro de este directorio se encuentran los siguientes subdirectorios:

- **admin:** variables que controlan la administración del nodo. Ver cuadro 6.2 .
- **decay:** control de estadísticas. Ver cuadro 6.3 .
- **info:** información de variables relativas al nodo. Ver cuadro 6.4 .
- **nodes:** información del estado y carga de cada uno de los nodos del cluster. Las variables son las mismas que para el nodo local, en *info*.
- **remote:** ruta que sirve para migrar procesos hacia otros nodos. Ver cuadro 6.5 .

En los cuadros se ha hablado de estadísticas referentes tanto a memoria com a procesador. Éstas contienen la siguiente información:

Estadísticas de Memoria

- tamaño total
- tamaño residente
- memoria compartida
- páginas de texto
- páginas de librerías
- páginas de datos
- páginas *sucias*

Estadísticas de Procesador

- tiempo de usuario en el nodo
- tiempo de los descendientes, no se muestra
- *nice* -prioridad UNIX- del proceso

[6] Siguiendo con la filosofía Linux.

| Fichero | W/R | Función | Valor de activación |
|---|---|---|---|
| block | W | bloqueo para la llegada de procesos | 1 |
| bring | W | devuelve los procesos migrados | 1 |
| config | R/W | configuración del sistema | - |
| decayinterval | W/R | intervalo de control estadístico | - |
| dfsalinks | W | activación de rutas DFSA | *ruta* |
| expel | W | expele procesos ajenos, activa *block* | 1 |
| fastdecay | W | tasa de decaída estadística rápida | *defecto:* 926 |
| gateway | R | número de gateways en red | *defecto:* 0 |
| lstay | W/R | no migrarán procesos locales | 1 |
| mfscosts | W/R | costes de MFS | **?** |
| mfskill | W | esconde árboles complejos a MFS | *ruta* |
| mospe | R/W | identificador de nodo (nodo ID) | */etc/openmosixmap* |
| nomfs | R/W | deshabilita la opción de MFS | 1 |
| overhead | R/W | sobrecargas sobre nodos y red | - |
| quiet | R/W | el nodo deja de diseminar información a otros | 1 |
| slowdecay | R/W | procesos con tasa de caída lenta | *defecto:* 976 |
| speed | R/W | velocidad del nodo | *valor* |
| sspeed | R/W | velocidad estandar, par areferencia | 10000 |
| stay | R/W | los procesos no migrarán | 1 |
| version | R | versión de openMosix funcionando | *variable* |

Cuadro 6.2: El API de openMosix: */proc/hpc/admin/*

| Fichero | W/R | Función | Valor de activación |
|---|---|---|---|
| clear | W | resetea valor estadísticos del nodo | 1 |
| cpujob | W | advierte que el proceso es cpu-intensivo | 1 |
| exec | W | protege de `execve` la política de decay | 1 |
| execonce | W | como el anterior, pero solo para el primer `execve` | 1 |
| fast | W | estadísticas de decaimiento rápidas | 1 |
| inherit | W | heredadas de un proceso a sus hijos | 1 |
| iojob | W | proceso io-intensivo | 1 |
| own | R | factor de *decay* que utilizaría el proceso | - |
| slow | W | estadísticas de decaimiento lentas | 1 |

Cuadro 6.3: El API de openMosix: */proc/hpc/decay/*

| Fichero | W/R | Función | Valor de activación |
|---|---|---|---|
| load | R | carga del nodo | *variable* |
| mem | R | memoria utilizada por el sistema | *variable* |
| rmem | R | memoria del sistema en bruto | *variable* |
| speed | R | velocidad del nodo o procesador del nodo | *variable* |
| status | R | estado del nodo | *variable* |
| tmem | R | memoria total del sistema -utilizada y sin utilizar- | *variable* |
| util | R | factor de unificación del sistema | *variable* |
| cpus | R | número de procesadores del sistema | *variable* |

Cuadro 6.4: El API de openMosix: */proc/hpc/info/*

| Fichero | W/R | Función | Valor de activación |
|---|---|---|---|
| from | R | nodo de procedencia | *nodo ID* |
| goto | W | obligación de migrar hacia ese nodo | *nodo ID* |
| statm | R | estadísticas de memoria | - |
| stats | R | estadísticas de procesador | - |

Cuadro 6.5: El API de openMosix: */proc/hpc/remote/*

| Fichero | Función |
|---|---|
| cantmove | motivo por el que no ha migrado el proceso |
| goto | nodo al que se desea migrar el proceso |
| lock | bloqueo del proceso en este nodo |
| migrate | obliga al proceso a migrar |
| nmigs | número de veces totales que el proceso ha migrado |
| sigmig | el solicita una señal tras migrar |
| where | nodo destino |

Cuadro 6.6: El API de openMosix: */proc/PID/*

- *vsize*, tamaño virtual
- *rss*, número de páginas residentes
- *nswap*, número de páginas en *swap*
- *cnswap*, páginas en *swap* de los descendientes

Por otro lado existe en */proc/PID/* una estructura ampliada relativa a la información para cada proceso. Entre esta información -propia de linux- ahora podrán encontrarse aspectos relacionados con openMosix, como muestra el cuadro 6.6 .

# Capítulo 7

# Tutoriales útiles para casos especiales

## 7.1. Nodos sin discos

*Never trust a computer you can't throw out a window.*
*Steve Wozniak*

Previamente se dará un repaso para tener las nociones básicas sobre el ensamblaje del hardware necesario para la construcción de este tipo de computadoras minimalistas. De esta manera si los nodos ya son funcionales[1], podrás pasarte los primeros apartados. A continuación se exponen esquematicamente los diálogos de comunicación entre nuestras computadoras, a fin de hacer comprender mejor el funcionamiento y poder detectar disfunciones con mayor precisión en el caso de no obtener una rápida puesta en marcha del sistema.

Aquí cubriremos todo el proceso de construcción de un cluster con una política de máquinas centrales -*servidores*- que servirán los servicios necesarios a un conjunto de nodos que poseerán los componentes hardware mínimos para funcionar -*clientes*-: desde la construcción de nodos minimalistas hasta su puesta en marcha con linux. Empecemos pues con lo tangible, las piezas.

### 7.1.1. Componentes hardware requeridos

Como ya se ha indicado, en este tipo de clusters tendremos dos tipos de computadoras: los *servidores* y los *clientes*.

Los **servidores** serán máquinas completas, entendiendo como tal un ordenador manejable localmente a través de su teclado, ratón y con la salida estándar direccionada a un monitor. Los **clientes** serán nodos con los mínimos componentes para funcionar. Éstos son, para cada uno:

- fuente de alimentación -a su vez podrá alimentar varios nodos si sabemos cómo realizar las conexiones-.
- placa madre -*motherboard*- con soporte de arranque por red.
- procesador -*cpu*-.
- memoria -entendiendo memoria principal de acceso aleatorio RAM-.
- y tarjeta de red con soporte de arranque por RPL (*Remote Program load*).

PRECAUCIONES: Deberemos prestar máxima atención a las compatibilidades entre las piezas que hayamos adquirido, puesto que resulta sumamente desagradable descubrir tras días de pruebas que todas ellas funcionan correctamente por separado, pero no en conjunto. Los problemas de temperatura, sobretodo de los procesadores, también son de máxima importancia, puesto que quemarlos no entra dentro de los objetivos a cubrir. Otra manera de reducir el riesgo de averías es trabajar siempre desconectado de la red eléctrica mientras trabajemos con las tarjetas y demás componentes.

### 7.1.2. Componentes hardware prescindibles

Para enfatizar aún más sobre lo que NO es necesario cabe mencionar, como componentes estándar, los siguientes:

- monitor, pantalla y targeta gráfica.
- teclado.
- ratón -*mouse*-.
- discos duros.
- diqueteras -*floppy*-.
- cualquier tipo de circuito integrado en placa madre.

[1] Entendiéndose que tras el correcto ensamblaje podemos acceder a la BIOS de la placa madre.

### 7.1.3. Ventajas e inconvenientes

**Ventajas**

- configuración más sencilla, organizada y barata.
- mayor seguridad en el sistema -pérdida de datos, intrusiones-.
- menores costes de mantenimiento del maquinario.
- menores riesgos por averías en discos, al disponer de menor número de ellos.

**Inconvenientes**

- rompemos la descentralización y las ventajas que ello supone en un cluster.
- mayor tráfico en la red, puesto que añadiremos al tráfico de procesos del cluster el acceso a sistemas de ficheros remotos.
- menor robustez de la red: por estar más cargada es más susceptible a fallos y congestiones.

### 7.1.4. Croquis de la arquitectura

Los **servidores** proveerán las cuatro necesidades básicas para que un cliente pueda arrancar y funcionar adecuadamente en red.

1. Arranque (protocolo RPL). Desde el momento de su puesta en marcha el computador deberá ser capaz de dejar a parte el reconocimiento de dispositivos locales e intentar hacerse con los servicios de arranque a través de su nic.
2. Identificación (DHCP). Los clientes no tienen forma de mantener sus configuraciones anteriores -si hubieran existido- en ningun medio físico, puesto que no disponen de memoria secundaria -discos duros, ...-. Los servidores deben ocuparse de asignar la identidad a cada cliente cada vez que arranquen.
3. Descarga de la imagen del kernel (TFTP). Igualmente los clientes no pueden almacenar el kernel con el que deben arrancar. Éste debe mantenerse en el servidor y ser trasferido cada vez que el cliente lo solicite.
4. Sistema de archivos remoto (NFS). Una vez que los clientes dispongan de identidad y hayan arrancado su kernel les faltará un lugar donde volcar sus resultados y donde obtener los ejecutables que sus operaciones requieran. Necesitaremos configurar NFSroot

Consecuentemente los **clientes** a su vez seguirán un proceso de arranque distinto al común. Véase este nuevo método.

1. Arranque desde tarjeta de red (RPL). La única información útil de que disponen los clientes es su dirección MAC -o dirección ethernet- que está en el *firmware* de su tarjeta de red -viene especificada por el fabricante-. Ésta es única en el mundo y es lo primero que el servidor recibirá de un cliente y le permitirá reconocerlo, o no.
2. Identificación como miembro de la red (DHCP).
3. Descarga del kernel (TFTP). Para el arranque en las computadoras clientes la imagen del kernel tendrá que sufrir ciertas modificaciones que le permitan cargarse en memoria y poder ser arranacadas desde ahí. El resultado de dichos cambios en un *bzImage* común dan lugar a una *tagged images* del mismo. La transmisión se realiza por TFTP, todo ello se explicará más tarde.
4. Sistema de ficheros remoto (NFS). Para que el *host* sea útil debe poseer un directorio raíz donde montar su sistema de ficheros y así pasar a ser operable como cualquier sistema linux.
5. Integración en el cluster. Para terminar, tendremos que cargar los *scripts* necesarios para poder integrar cada cliente al cluster openMosix.

### 7.1.5. Diálogo de comunicación

Llegados a este punto se acuerda tener una idea del proceso que debe llevar a cabo cada tipo de computadora. Seguidamente se presenta todo ello en un pequeño diálogo -muy intuitivo- para esclarecer posibles dudas de procedimiento. Se indica igualmente el protocolo que lo implementa.

El primer paso es arrancar algún servidor, puesto que por deficnición sin ellos ningún cliente podría arrancar ningún servicio. Una vez hecho esto podrá ponerse en marcha un cliente. Su BIOS deberá percatarse de que debe iniciar desde la tarjeta de red -así lo habremos configurado, se explicará más tarde- y enviará por *broadcast*[2] su dirección MAC a la espera que alguien -un servidor- la reconozca y sepa qué configuración asignarle.

RPL

- **Cliente (C):** ¿alguien en esta red reconoce mi MAC?
- **Servidor (S):** yo mismo. Paso a darte capacidad para formar parte de esta red.

DHCP

- **C:** perfecto S, ¿quién soy?
- **S:** eres la IP x.x.x.x, tu servidor soy yo y tu fichero de arranque es `vmlinuz.<etiqueta>`

TFTP

- **C:** por favor S, dame `vmlinuz.<etiqueta>` para poder arrancar linux
- **S:** coge el fichero de mi directorio `/tftpdir/vmlinuz.<etiqueta>`
- **C:** correcto, empiezo mi arranque...

NFS

- **C:** arranque completo. Déjame montar mi sistema de ficheros raíz (con NFS)
- **S:** lo tienes en `/tftpboot/C/`
- **C:** déjame montar mis demás directorios (`/usr`, `/home`, etc...)
- **S:** los tienes ahí también
- **C:** ¡gracias S, ahora soy totalmente operable!

Como se verá a la hora de dar detalles sobre la forma de configurarlo, los nombres de ficheros y directorios pueden ser escogidos arbitrariamente.

### 7.1.6. Servicios requeridos

El diálogo anterior requiere de los servicios ya introducidos para poder llevarse a cabo.

- el servidor debe ser capaz de responder a peticiones de arranque por **RPL** de los clientes.
- el servidor y el cliente deberán comprender el protocolo **DHCP**.
- ídem para el protocolo **TFTP**.
- el servidor deberá arrancar el demonio servidor de **NFS**. Las computadoras clientes deben ser capaces de montar unidades por NFS, es una opción del kernel.

[2] Es la dirección de difusión (255.255.255.255), se envía a todas las direcciones del segmento de red.

Como se ha señalado e impone el número de servicios a configurar, se divide en cuatro partes la temática de esta sección. En una primera fase se comentarán sus especificaciones para tener una visión de las posibilidades que ofrecen y tras ello se verán algunas herramientas necesarias para su configuración.

#### I - El servidor y el cliente del protocolo RPL

El protocolo RPL ha sido diseñado para el arranque desde tarjetas de red -nic- y dota al computador de la posibilidad de adquirir la posibilidad de hacer petición por DHCP, servicio que no posee debido a su falta de capacidad de almacenaje.

Instalar un nic con este soporte permite interrumpir a la BIOS del computador -en este caso un nodo cliente- con la interrupción INT 19H para gestionar el arranque desde este dispositivo. Los requerimientos, como ya se ha dicho, es que la placa madre del computador permita arranque por red y que el nic permita arranque RPL, algo común en el maquinario actual.

La estructura de datos en que se basa RPL es una pequeña imagen -rom- que deberá ser transferida al nic. Esta rom puede ubicarse en 2 localizaciones:

- **Integrarse directamente en el hardware del nic.** Aunque esta posibilidad solo viene contemplada *de serie* por las tarjetas más caras puesto que requiere un chip *bootp rom* específico para cada modelo. Este chip ha de colocarse sobre el socket -que suelen llevar todas las tarjetas- y tiene una capacidad de unos 32KB.

- La otra posibilidad -que es la más económica y la que aquí se detalla- es **montar un servidor de roms** para que los clientes obtengan la suya desde él. Esta posibilidad ofrece ventajas tanto a nivel de flexibilidad -no será necesario el chip. Las imágenes suelen ocupar 16KB.

#### II - El servidor y el cliente del protocolo DHCP

DHCP es un superconjunto de las operaciones que puede realizar BOOTP, que a su vez es una mejora sobre el antiguo protocolo de arranque RARP. En este caso utilizaremos DHCP para conseguir la dirección IP.

El funcionamiento de BOOTP -y por extensión DHCP- es sencillo. Es un protocolo de Internet que desde hace años -concretamente desde el 1985- está especificado en la RFC951[3] de la *The Internet Engineering Task Force* -perteneciente a la *Internet Society*-. Este protocolo tiene como principales características:

- se realizan intercambios de un solo paquete con el mismo formato tanto para peticiones como respuestas. Este paquete o datagrama es IP/UDP y se utiliza *timeout* para retransmitir mientras no se reciba respuesta a una petición.

- Existe un código de opción que indica si el paquete es BOOTREQUEST (por el puerto 68) o BOOTREPLY (por el puerto 67). Las peticiones BOOTREQUEST contienen la máquina que las solicitó si ésta es conocida.

- Opcionalmente, se puede conceder el de la BOOTREPLY del nombre de un fichero que se debe descargar.

- Los servidores se comportan como *gateway* permitiendo incluso peticiones BOOTP entre varias redes.

El contenido de los campos de un paquete en el protocolo BOOTP es interesante para ver sus opciones. Además puede servirnos para detectar, mediante aplicaciones *sniffers*, fallos en tarjetas. Lo vemos en el `Cuadro 7.1`.

Como puede verse, los campos son muy significativos para el tipo de operación que se pretenda realizar. Estos campos están incluidos en un datagrama IP/UDP, lo que representa una ventaja respecto a RARP, que solamente manipulaba operaciones en bruto sobre la red -sobre ethernet- no permitiendo opciones como el encaminamiento entre redes, ni la utilización de otro tipo de tecnologías distintas que no soportasen protocolo ARP.

El funcionamiento es muy sencillo. El cliente rellena un paquete con todos los campos que conoce y con el `opcode` de petición, y lo difunde a la dirección 255.255.255.255 de broadcast.

[3] `http://www.ietf.org/rfc/rfc0951.txt?number=951`

| CAMPO | BYTES | DESCRIPCIÓN |
|---|---|---|
| op | 1 | opcode, 1=BOOTREQUEST – 2=BOOTREPLY |
| htype | 1 | tipo de hardware, 1=10Mbps ethernet |
| hlen | 1 | longitud de la MAC (normalmente 6) |
| hops | 1 | utilizado por los *gateways* para contar los saltos |
| xid | 4 | ID (identificador) de la transacción |
| secs | 2 | rellenado por el cliente con el riempo que lleva solicitando respuestas |
| - | 2 | reservados |
| ciaddr | 4 | dirección IP del cliente, en el caso de conocerse |
| yiaddr | 4 | dirección de un cliente, lo rellena el servidor tras consultar su configuración |
| siaddr | 4 | dirección del servidor |
| giaddr | 4 | opcional, para atravesar varias redes |
| chaddr | 16 | dirección hardware del cliente |
| sname | 64 | nombre del servidor (opcional) |
| file | 128 | ruta del archivo arranque |
| Vend | 64 | etiqueta específica del fabricante |

Cuadro 7.1: Nodos *diskless*: Campos de un paquete del protocolo BOOTP

Luego contesta la máquina servidora rellenando el paquete de manera que el cliente recibe el paquete y lo procesa para colocar como praámetros de su pila IP los proporcionados por el servidor. Este caso se da, como se ha comentado repetidamente, siempre que el servidor esté configurado para responder a la dirección MAC del cliente que genera la petición.

El servidor de DHCP se inicia como un demonio -*daemon*- externo al *inetd*. Gracias a DHCP queda especificado todo lo referente a la identificación IP de aquellas máquinas que intenten entrar en la red. Se podrá asignar un rango dinámico de direcciones, también se puede hacer que todos descarguen la misma imagen del kernel sin asignarla directamente a ningún host en particular -útil en un cluster homogéneo-. En el apéndice relativo a *Salidas de comandos y ficheros* se ve un ejemplo del fjchero *dhcpd.conf* .

En cualquier caso y como resulta obvio, será necesario asignar una IP a cada *host* -sea nodo cliente o servidor- para que forme parte de la red del cluster y poder especificar el entorno de configuración de cada máquina, así como asignar los ficheros de configuración de openMosix.

#### iii- El servidor y el cliente del protocolo TFTP

Como su nombre indica -*Trivial FTP*- este protocolo es un FTP especial: mucho más simple que éste. Para empezar, la capa de transporte utiliza UDP en lugar de TCP y transferencia por bloques para el envío de los ficheros, lo que hace más sencilla la transferencia (y más en una red ethernet). El otro motivo por el que se utiliza UDP y un mecanismo tan simple de control de paquetes es porque se necesita que el programa y lo mínimo de pila IP ocupen poco en memoria para que este pueda grabarse en ROMs, que inherentemente disponen de poca capacidad (32KBytes por lo general).

El servidor de TFTP -controlado por el demonio *tftpd*- se suele arrancar desde *inetd*. Su configuración se centrará en el servidor, puesto que el cliente lo adopta sin que sea necesario que lo hagamos explícito en ninguna parte.La configuración puede hacerse:

- **Simple.** Hay que tener cuidado del entorno en el que se encuentra el servidor, ya que el protocolo tftp es inherentemente inseguro y al no exigir autentificación, los clientes pueden solicitar el fichero */etc/passwd* o similar sin ningún problema, lo que supone una inseguridad considerable.

- **Seguro.** Basa su seguridad en una llamada a *chroot()*. De este modo, en la ejecución de la rutina de aceptación de peticiones, el directorio exportado se convierte en directorio raíz. Consecuentemente el acceso a otros ficheros es más difícil.

TFTP es un servicio inseguro, ya que el propio protocolo es simple e inseguro, por lo que es recomendable que el servidor que posea este servicio esté aislado de cualquier red que no garantice medidas serias de seguridad. En casi contrario, cualquiera podría sustituir los ficheros que descargan los clientes e incluir en ellos alguna rutina no deseada.

La activación de este servicio es tan fácil como una línea en el fichero */etc/inetd.conf*. Los parámetros pueden variar sensiblemente dependiendo de la aplicación que usemos para controlar el demonio. En principio debería no haber problemas con:

```
tftp dgram udp wait root /usr/sbin/in.tftpd in.tftpd /tftpboot
```

IV- NFS

NFS es el sistema de almacenamiento ingeniado por *Sun Microsystems* y que utiliza RPC. Es un modelo de servidores sin estado, es decir, los servidores NFS no guardan en nigún momento los archivos a los que se están accediendo[4].

El funcionamiento se basa en dos secciones: cliente y servidor. El cliente monta el sistema de ficheros exportado por el servidor y a partir de este momento accede a los ficheros remotos como si fuesen propios. Este sistema es utilizado desde hace tiempo en casi todos los sistemas Unix como método de compartir ficheros en red. NFSroot designa el método que sigue el kernel cuando en lugar de tomar el clásico sistema de ficheros *ext2* o *reiserfs* para montar /, importa un directorio NFS de un servidor.

El sistema de archivos que debemos exportar en el servidor debe contener todos los archivos necesarios para que la distribución pueda funcionar. Este factor es muy variable, dentro de cada distribución, según activemos unos servicios u otros.

En principio cabe pensar en qué directorios deben ser necesariamente de lectura y escritura o solamente de lectura. Una buena forma de ahorrar espacio en el servidor sería exportando para todos los clientes los mismos directorios para solo lectura y particularizando para cada uno los de escritura y lectura. De cara a evitar mayores consecuencias en los posibles errores que puedan cometerse en la puesta en marcha del servicio, un buen consejo es no exportar directamente ningún directorio del servidor, sino uno propio de un cliente puesto en el directorio del servidor que usemos para disponer los directorios exportables -normalmente `/tftpboot/`-.

Aproximadamente se requieren unos 30MB de información específica para cada nodo -si estos disponen de un sistema linux mínimo-, mientras que el resto puede ser compartida.

### 7.1.7. Configuración de los servicios requeridos

Llegados a este punto tenemos un conocimiento adecuado de lo que pueden ofrecernos los cuatro servicios que utilizaremos. Sigamos la lectura para lograr hacer nuestras configuraciones, y aún más, entenderlas.

Reiteremos una vez más un par de puntos importantes:

- A la computadora *completa* que proporciona los servicios la referenciaremos como *servidor* y hará las funciones de servidor RPL, DHCP, TFTP y de NFS. Será la encargada de gestionar el arranque y de exportar la totalidad del sistema de ficheros para los nodos *diskless -clientes-*.
- Para todo ello deberemos estar seguros de haber incluido soporte para NFSroot en la configuración del kernel del cliente. Los demás servicios no tienen nada a configurar en la parte del cliente -puesto que no tendría donde almacenarla-.

En este capítulo aprenderemos a configurar los servicos que anteriormente se han expuesto, así pues se seguirá la misma estructurarión cuaternaria: rpl, dhcp, tftp y nfs.

I- Configuración servidor RPL. *rpld.conf*

Lo primero será conseguir un demonio para el protocolo RPL. Una implementación funcional y estable puede bajarse de un rincón de la web de la Universidad de Cambridge[5].

[4] lo que puede ser un problema en el caso de querer controlar el bloqueo de un fichero en concreto, o una ventaja en el caso de que caiga la máquina y se recupere rápidamente.

[5] http://gimel.esc.cam.ac.uk/james/rpld/

Una vez compilado e instalado -y siendo root- podremos ejecutar el comando

```
rpld
```

Este demonio intentará acceder al fichero */etc/rpld.conf*. Este fichero tiene una sintaxis muy concreta -a la vez que potente- y por ello se recomienda leer las páginas *man* posteadas en la misma dirección. Su lectura permitirá al administrador hacer un examen exhaustivo de todas las posibilidades que pueda brindarnos este servicio.

Existe un ejemplo de este tipo de fichero en el apéndice *Salidas de comandos y ficheros*. Basta con anotar la dirección MAC del cliente que realizará la petición e indicar la imagen rom que se corresponde al modelo de chipset de su nic[6]. Los demás parámetros se refieren al tamaño de los bloques que se transferirán; los valores indicados en el fichero de ejemplo no deben dar problemas.

N. al LECTOR. La documentación necesaria para generar la rom adecuada a cada modelo de nic se encuentra en la siguiente sección: **ROMs para arranque sin discos**.

#### II- Configuración servidor DHCP. *dhcpd.conf*

El fichero que el demonio *dhcpd* leerá para cargar la configuración será el */etc/dhcpd.conf* . Este demonio puede lanzarse con el comando

```
/etc/init.d/dhcpd start
```

La ruta del demonio puede variar según la distribución. En el apéndice *Salidas de comandos y ficheros* hay un ejemplo de este tipo de fichero, para comprenderlo se aconseja ejecutar `man dhcpd` en cualquier consola. A continuación se detallan los parámetros que pueden ser más interesantes. Para la **red**:

- *default/max-lease-time* tiempos en milisegundos para las operaciones de reconocimiento.
- *option domain-name* define el nombre para la red.
- *option domain-name-server* dirección del servidor.
- *option subnet-mask* máscara de subred.
- *range dynamic-bootp* rango de direcciones, en este caso son 255 computadoras (de la 0 a la 254).
- *option routers* aquí indicaremos la dirección de un nodo que haga de enrutador, si existe.

Para cada **nodo**:

- *host* indicará el nombre del nodo.
- *hardware-ethernet* quizás haga falta decir que cada `X` se refiere a un dígito hexadecimal correspondiente a la dirección MAC de la targeta en concreto.
- *fixed-address* asigna al nodo la dirección IP que el servidor hará llegar al cliente con la MAC anterior.
- ***filename*** referencia a la *tagged image* (imagen de kernel modificada) para arrancar.
- *option root-path* ruta para / del nodo.

El parámetro *filename* hace referencia a una imagen del kernel distinta a la salida del comando `make bzImage` utilizado para realizar la imagen del kernel linux. Para conocer las modificaciones necesarias a una iamgen para obtener una *tagged image* consúltese **ROMs para arranque sin discos** en la próxima sección.

#### III- Configuración del servicio TFTP *inetd.conf*

[6] La generación y manejo de estas roms será fácil una vez se conozca el tema de primera mano. La opción más recomendable viene de parte del proyecto Etherboot. Para más información *ROMs para arranque sin discos*.

Como ya se ha mencionado, el protocolo TFTP es inseguro *per se*, es conveniente asegurar nuestro sistema con herramientas como `tcp-wrapper` y algún tipo de cortafuego -*firewall*- si queremos bajar las probabilidades de que llegue a darse una intrusión[7].

Una vez hagamos hecho las modificaciones ya dadas al fichero */etc/inetd.conf*, para relanzar el demonio *inetd*[8] deberá ejecutarse:

```
/etc/init.d/inetd reload
```

Otra opción para relanzarlo es la señal HUP. Para ello deberemos conocer el PID del demonio y ejecutar:

```
kill -HUP <inetd_pid>
```

Para restablecer la seguridad que TFTP restará a nuestro sistema se debe hacer el máximo esfuerzo para: evitar recibir ataques de *ipspoofing*, proteger mediante permisos correctos el sistema, realizar un control sobre los *logs* creados y dar el mínimo de permisos posibles al servicio *tftpd*, por eso ejecutamos el demonio como el usuario `nobody`. El *chroot* es imprescindible así como el control de `tcp-wrapper` mediante los ficheros.

```
/etc/host.allow
/etc/host.deny
/etc/syslog.conf
/etc/inetd.conf
```

En la línea que hay que añadir -o descomentar- en el *inetd.conf* aparece como último parámetro el directorio raíz que sirve para encontrar los ficheros a importar. En este caso se ha considerado el directorio */tftpboot/* del servidor. En este directorio sería conveniente colocar todas las imágenes de los clientes[9], para dotar al sistema de cierto criterio y coherencia. Cabe recordar que **es el fichero *dhcpd.conf* el que marca qué imagen debe recoger cada cliente**.

#### IV- Servidor NFS

Hasta este punto los clientes tienen que poder empezar a arrancar su kernel, no obstante no podrán iniciar el proceso con PID 1 , el INIT, debido a que no deben ser capaces de poder montar ninguna unidad. Aquí es donde entra el único servicio que resta: el NFS. Esta sección está separada, según sea el cometido:

- la exportación -por parte del servidor- de una serie de directorios
- o la importación del directorio / por parte del cliente.

#### IV a- Exportación de directorios con NFS

El servidor debe tener arrancado el demonio de *servidor de nfs* para poder proveer este servicio. Este demonio comunmente -en la mayor parte de las distribuciones linux- viene con el nombre `nfssserver`; para arrancarlo puede procederse como en cualquier demonio:

```
/etc/init.d/nfsserver start
```

El fichero de configuración de NFS que exporta los directorios necesarios suele estar en */etc/exports*. Tiene su propio manual (`man exports`) que se deberá leer para incluir los directorios con una sintaxis correcta y añadir algo de seguridad al servicio NFS. En el caso de dos clientes -`metrakilate` y `mCi`- de arranque sin discos, este fichero de configuración tendría que ser algo parecido a lo siguiente:

[7] Se pueden cerrar los puertos de tftp -normalmente el 69 UDP- salvo para los clientes autorizados de este servicio, i.e. las direcciones IP concedidas en el apartado anterior.

[8] Este demonio suele iniciarse al arranque de la computadora.

[9] Las imágenes de kernel linux serán de tipo `bzImage` de momento. Ésta estructura no sirve para ser transferida por red para cargarse y ejecutarse desde memoria, así que los ficheros que deberán constar en *dhcpd.conf* deberán ser conversiones a *tagged images*. Para más información *ROMs para arranque sin discos*.

```
/tftpboot/metrakilate/  192.168.1.2/255.255.255.0(rw,no_root_squash,sync)
/tftpboot/mCi/          192.168.1.3/255.255.255.0(rw,no_root_squash,sync)
/media/dvd/             192.168.1.0/255.255.255.0(ro,no_root_squash,sync)
```

En este ejemplo se indica donde se encuentra la raíz para cada una de las dos máquinas, más un directorio -la unidad local de dvd- que será exportada a cualquier nodo con IP `192.168.1.X` (siendo `X` un valor entre 1 y 255).

El comando `exportfs -a` hace que se proceda a exportar los directorios que se lean de este fichero en el caso de que se añada alguna entrada cuando el servicio está activado. Otra posibilidad es guardar los cambios al fichero y enviar una señal `tHUP` o llamar al *script* con la opción *reload*.

#### IV B- Raíz en los nodos clientes

Dentro de */tftpboot/<directorio_raiz_de_un_nodo>* se debe encontrar un sistema réplica de ficheros a la distribución que queramos utilizar. La edción de ficheros para configuraciones añadidas o la ejecución de aplicaciones podrá hacerse normalmente como si se trabajara con unidades locales, ya que la capa de VFS nos hace transparente el proceso.

Así pues una solución es comenzar una instalación de linux cambiando la ruta de instalación a */tftpboot/<directorio_raiz_de_un_nodo>*, de manera se aseguran que todos los archivos necesarios se encuentran en el lugar requerido.

Otra posibilidad es hacer una instalación normal en un disco conectado a la placa madre del nodo cliente para asegurar que todo funciona correctamente. Una vez se ha comprobado que el sistema operativo es capaz de montar unidades por NFS y tiene capacidad de entrar en una red, puede procederse a empaquetar su directorio / desde la máquina servidora[10] para luego desempaquetarlo en el */tftpboot/<directorio_raiz_de_un_nodo>* elegido.

La alternativa más dura será montar un linux -existente en algun disco- en el directorio mencionado del servidor. Esta alternativa pasa por copiar en el directorio lo siguiente:

- */bin, /sbin, /lib*. Estos directorios pueden ser copiados directamente del directorio raíz con `cp -a /bin /tftpboot/<directorio_comun_a_los_nodos>/` y luego hacerlos compartirdos entre todos los clientes sin discos mediante enlaces duros[11].
- */var, /tmp, /etc*. Deben ser copiados en cada cliente y adaptados a las necesidades de cada uno de ellos, más abajo se da una breve descripción de lo que debe ser cambiado en */etc* para que funcione.
- */proc, /root, /home*. Deben existir y tener los permisos adecuados según sea su propietario.
- */dev*. Debe ser único de cada cliente. Para crearlo se pueden utilizar dos opciones, `devfs` o una copia del sistema y luego la ejecución de MAKEDEV (script) y añadir un dispositivo */nfsroot* al kernel que utilicemos.

Existen dos puntos que complican sensiblemente el proceso. Se detallan seguidamente.

1. configuración propia de cada nodo.
2. los dispositivos */dev* .

En el caso de */etc* es conveniente quitar del entorno *rc.d* todos aquellos scripts de arranque que no sean necesarios para cada nodo. Hay que configurar todos los apartados específicos como pueden ser el */etc/network/interfaces/* y/o adecuar el *openmosix.map* -que debiere ser indistinto para cualquier máquina del cluster-.

En general deben ser cambiados todos los parámetros propios de cada nodo, es relevante señalar la importancia de los parámetros de red. El problema de los dispositivos se puede resolver de dos maneras. La primera es utilizar el viejo sistema de `minor` y `major number` que hasta ahora se han utilizado en linux. Para ello puede hacerse una copia con:

```
cp -a /dev /tftpboot/<directorio_raiz_de_un_nodo>
cd /tftpboot/<directorio_raiz_de_un_nodo>
```

[10] Habiendo montado el sistema de ficheros del cliente -/- en el servidor -*</directorio_cualquiera>*-, con lo que ahora el servidor pasa a ser cliente que importa el directorio y el nodo el exportador de su / .

[11] `man ln`

```
./MAKEDEV
```

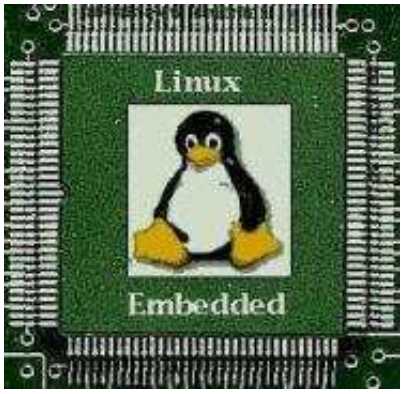
Esto consigue que se generen los enlaces necesarios a este directorio -propio del nodo- en lugar de a */dev/xxxx*, que es el directorio del servidor. Además hay que hacer un nuevo dispositivo de `major 0` y `minor 255`, y exportarlo al kernel para que se puedan recibir comandos de nfsroot en él. Esto se puede conseguir con:

```
mknod /dev/nfsroot b 0 255
rdev bzImage /dev/nfsroot
```

Aquí `bzImage` tiene que ser la imagen del kernel del nodo *diskless* en cuestión. Generalmente se encuentra en */usr/src/linux-version/arch/i386/boot* tras su compilación.

Llegados aquí, finalmente, tus clientes *disk-less* deberían estar funcionando sin nigún problema.

## 7.2. ROMs para arranque sin discos



*There is only one satisfying way to boot a computer.*

*J. H. Goldfuss*

Llegados a estas páginas habrá de comprenderse que al carecer de discos, los nodos *diskless* deben disponer de lo necesario para su arranque en otra computadora, y en el tipo de sistemas que aquí se detallan esta información estará organizada en un -o varios- nodos servidores.

Esta sección no cubre el procedimiento de configuración de los servicios, que debe haberse visto durante la lectura de la anterior. Aquí se trata la necesidad de aportar al cliente ciertos paquetes de información -en forma de roms- para que sea capaz de cumplir con su parte en el diálogo de comunicación que finalizará con su arranque. Cabe insistir una vez más que el cliente en ningún momento se percata de que no tiene un sistema operativo instalado en un disco local, la transparencia a este nivel la otorga el propio linux.

Haciendo memoria. El cliente -como cualquier otra computadora- arrancará su BIOS al ser alimentado. Tras la detección de memoria y también su tarjeta de red se producirá la interrupción de BIOS `INT 19H` que dirigirá su esfuerzo en encontrar información que le permita arrancar a través del nic instalado. En este momento es cuando el servidor debe estar atento para prestar la **primera rom**, correspondiente al servicio RPL[12].

La **segunda rom** es necesaria durante la puesta en marcha del tercer servcio, el TFTP. Se utilizará este protocolo para descargar y arrancar un kernel desdel servidor, pero este kernel debe estar modificado -no sirve directamente un fichero `bzImage`, resultado de una compilación con `make bzImage`-.

## Arranque del cliente desde chip EPROM

Los pasos necesarios para este proceso son los siguientes:

- comprobar que la tarjeta de red dispone de socket para inserirle un chip; será el señal que confirma que soporta arranque por RPL.
- en el caso de no querer situar la rom de RPL en un servidor será necesario adquirir un chip *bootp rom* compatible con la tarjeta. Lo proporciona el fabricante.
- generar una imagen ROM válida con el modelo de tarjeta.
- quemar dicha imagen en el chip -si usamos esta opción- o situarla en el servidor y configurar el servicio.
- generar una *tagged image* del kenel para poder arrancarla con la ROM, situarla en el servidor y configurar el servicio.

Lo que tiene que entenderse en el tercer punto es que la rom contiene diferencias para cada tarjeta, o mejor dicho, para cada *chipset* de tarjeta de red. Así pues es aconsejable realizar pruebas con esta misma imagen grabada en un disquete, para no tener que sustituir diversas veces el contenido de nuestro chip (algo sin duda más engorroso que borrar y grabar un disquete).

## Arranque del cliente desde disquete

El arranque realmente no difiere en exceso, puesto que podemos verlo simplemente como una diferente ubicación de las instrucciones necesarias -en disquete en vez de en chip o en el servidor- para arrancar nuestro nodo cliente.

En efecto, cabe elogiar el trabajo de la gente del proyecto **Etherboot** por poner a nuestro alcance un sistema tan práctico para hacer nuestras primeras configuraciones: poner la imagen en un disquete.

La ventaja que esto aporta es básicamente que es un proceso realizable en la mayoría de las computadoras, siempre que dispongan de conexión a internet y de disquetera. Con este sistema podemos:

- hacer nuestras configuraciones desde disquete y preparar una rom totalmente operativa para volcarla en memoria EPROM[13].

[12] Si el nic dispone de socket para chip EPROM se podrá colocar esta rom en él, alibiando al servidor del servicio RPL.

[13] Si trabajamos con imagenes rom en el servidor podremos prescindir de disquetes, puesto que entorpecería y ralentizaría el proceso. para cargar la nueva configuración solo sería necesario matar al demonio `rpld` y volverlo a iniciar

- o dejar nuestros nodos clientes con la disquetera y arrancarlos siempre desde disquete.

La opción más elegante es sin duda la primera, puesto que si realmente el cometido que nos ha hecho llegar aquí han sido los nodos minimalistas, ¿qué menos que prescindir de las disqueteras, su cable de alimentación y el de datos?

## Generando imagen para RPL

Como no podía ser de otra manera la comunidad del software libre pone a nuestra disposición un sistema en extremo asequible e intuitivo para la generación de las susodichas imágenes rom.

Existen dos propuestas, ambas se encuentran en SourceForge:

- Etherboot[14], que se sirve de drivers insertados en la imagen.
- Netboot[15], que dispone los drivers en un paquete.

Explicaremos con más detalle cada una de estas propuestas, analizando también el benecifio que en cada caso puedan aportar a nuestro sistema.

### I - ETHERBOOT

El proyecto Etherboot dispone de un apartado sobre documentación (en inglés, como es de suponer). Los procedimientos explicados a continuación pueden ampliarse en su web, aunque en esencia puede resumirse todo ello de la forma que sigue.

¿Cómo puedo obtener una imagen ROM?

- Marty Connor ha trabajado en una versión en linea para poder bajarnos la imagen para nuestro modelo específico de *chipset*. Puedes encontrarlo en su proyecto ROM-o-matic for Etherboot[16].
- o podemos compilar nosotros mismos la imagen.

El mecanismo que genera la imagen es el mismo, es decir, romomatic simplemente llama a la aplicación etherboot, con las opciones marcadas, a través de un *script*. Por esa razón -y pensando en simplificar siempre las cosas- aquí se aboradrá el caso desde *romomatic*. En caso de querer hacer las cosas por propia cuenta habrá que enterarse bien de los parámetros de la aplicación[17].

En *rom-o-matic.net* existen varias versiones, es recomendable utilizar la versión estable -*production release*-. Un hiperenlace lleva a la página donde hay que introducir:

1. las especificaciones del chipset,
2. el medio desde donde la imagen será leída -chip, disquete, imagen en servidor-,
3. y configuraciones opcionales -*temporizadores, etc.*-.

Finalmente pulsar `Get ROM` para terminar bajando la imagen. Las imágenes preparadas para disquete hay que copiarlas con el comando

```
cat eb-5.0.8-tu_nic.lzdsk >/dev/fd0
```

donde `/dev/fd0` refiere a nuestra disquetera y `eb-5.0.8-tu_nic.lzdsk` es la eom.

En este proceso pueden surgirte las siguiente dudas, que como no, tienen su respuesta.

- ¿qué modelo de *chipset* tiene mi tarjeta?
- ¿está soportado por Etherboot?

[14] `http://www.etherboot.org/`
[15] `http://netboot.sourceforge.net/`
[16] http://rom-o-matic.net/
[17] http://etherboot.sourceforge.net/doc/html/userman-3.html

El modelo de *chipset* viene especificado por un número que puede conseguirse de diferentes maneras. Si no aciertas a conseguirlo con la documentación que la tarjeta pueda llevar sería adecuado hacer uso de los comandos, como superusuario (*root*):

```
lspci
cat /proc/bus/pci/devices
```

Puedes encontrar un ejemplo de su salida en el apéndice referente a *Salidas de comandos y ficheros*. Sobre si está soportado, y para conseguir siempre resultados acorde con la versión vigente de Etherboot, debe consultarse el hiperenlace de la página de generación de imagenes de ROM-o-matic sobre los PCI IDs soportados.

### II- NETBOOT

¿Quieres redactar este apartado? ¡Envíamelo! mikel@akamc2.net . GRACIAS

## Generando la rom para *tagged image*

La información contenida en una imagen de kernel *normal* -tipo `bzImage`- no es suficiente para que pueda cargarse por red. Pero esto no supone ningún problema, el usuario dispone de las herramientas `mknbi` que permiten *taggear* estos ficheros para hacerlos capaces de ello.

Así pues el resultado de aplicar las modificaciones de `mknbi-linux` al fichero `bzImage` resultado de la compilación de las fuentes de linux en el nodo que queramos arrancar por red, se llama *tagged image*. A efectos de usuario no supone ninguna diferencia con una imagen sin modificar.

Los paquetes necesarios para proceder de esta manera contienen unas aplicaciones llamadas`mknbi-<so>` donde *so* refiere al sistema operativo que queramos arrancar. Nos centraremos en la utilidad `mknbi-linux`.

¿Qué necesita `mknbi` para funcionar? Evidentemente los parámetros obligados son el nombre de la imagen a modificar y el nombre de la iamgen de salida. Una posible llamada podría ser

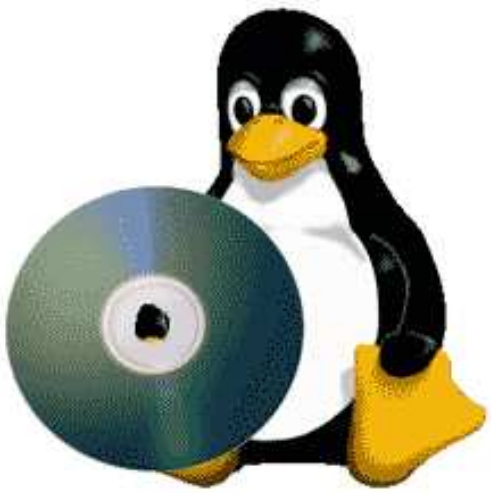
```
mknbi-linux --format=nbi --ip=rom --output=/tftpboot/vmlinuz-metrakilate
--rootdir=/tftpboot/metrakilate /usr/src/linux/arch/i386/boot/bzImage
```

Una vez más se enfatiza en el hecho que para un pleno conocimiento de la herramientas será más útil consultar los manuales que incorpora, o la documentación que se dispone en su web.

Como se ve es sencillo de utilizar. Con un poco de soltura también se pueden hacer menús que gestionen la carga de uno u otro kernel con pantallas de menús tipo *ncurses* mediante `mknbi-mgl` y un lenguaje de programación semejante a Pascal, lo que es una opción bastante profesional para entornos de producción.

Sobre la aplicación cabe decir que es necesario pasarle la ruta `nfsroot` -correspondiente al parámetro `rootdir`- que indica de donde importar la raíz /. En el caso de no pasarse este parámetro existe una cosntante dentro de */usr/src/linux/fs/nfs/nfsroot.c* que indica de donde será cargado el raíz. Por defecto es */tftpboot/<direccion_IP>*.

## 7.3. *Live Linux CD!* Funcionando desde cdrom



*Being busy does not always mean real work.*
*The object of all work is production or accomplishment*
*and to either of these ends there must be forethought, system, planning, intelligence*
*and honest purpose, as well as perspiration.*
*Seeming to do is not doing.*

*Thomas A. Edison*

Un *Live Linux cdrom* es un disco cdrom que contiene un sistema operativo Linux completo y por consiguiente un sistema de ficheros. Los *Live Linux CDROM* arrancan directamente desde un dispositivo lector cdrom. Simplemente tendremos que configurar la BIOS de la computadora para que intente arrancar primero desde cdrom.

Generalmente el orden para el arranque en la BIOS es: disquetera, disco duro y CDROM. Hay otras posibilidades, como puede ser el arranque por red, tal y como se explica también en el segundo apartado de este mismo capítulo (*Nodos sin discos*).

Para acceder a las opciones de la BIOS de nuestra computadora simplemente tendremos que pulsar la tecla DEL (o SUPR según nuestro teclado indique) en el momento en que se proceda al chequeo de la memoria, breves instantes después de poner en marcha cualquier PC.

En la red podemos encontrar proyectos bien consolidados que consiguen dar una alta funcionalidad a sistemas a partir de un linux embebido en un cdrom. Entre estas posibilidades contamos actualmente con el disco *SuSE Live-eval*, que proporciona una demo para la distribución alemana SuSE, o la distribución Knoppix[18]. Google proporciona, como no, una referencia sobre las distribuciones *live* más extendidas en *Google Live Linux* [19].

Así pues, ahora que se ha puesto en conocimiento del lector la posibilidad de arrancar un sistema linux almacenado en cdrom, puede surgir la primera pregunta: ¿nos será posible guardar nuestros documentos o cambios en este sistema? Evidentemente NO en el propio cdrom, pero como veremos existen multitud de alternativas que linux nos brinda con un fácil manejo.

En primer lugar cabe señalar que ejecutar un sistema operativo desde cdrom será tanto más ameno cuanto más rápido sea el dispositivo lector. Actualmente se han llegado a las 72x[20] en dispositivos que alcanzan a leer varias pistas simultáneamente (y es que esto de paralelizar los procedimientos cada vez es más una realidad) y esto se traduce en casi 11 mega-octetos por segundo. No obstante, cualquier reproductor de medianas prestaciones seguirá dándonos unos resultados más que aceptables.

En respuesta a si esta solución de arranque aporta algún tipo de beneficio extra, podemos referirnos básicamente al ahorro en material informático. Si sólo necesitamos el cdrom para poder realizar las funciones que nuestras necesidades requieren (navegar por la red, trabajar con un sistema de archivos remoto, *etc*...) nuestro terminal podrá carecer de disco duro y disquetera. Esto también implica, a su vez, un gran ahorro en tareas de mantenimiento de los equipos, puesto que cualquier actualización puede quemarse en nuevos cds (¡o borrarlos y reescribirlos!) y ningún usuario podrá modificar el sistema que hayamos generado.

Es importante pues que el sistema mínimo para arrancar desde cdrom contenga el hardware necesario para un arranque sin discos además del lector de discos compactos.

En resumidas cuentas, este sistema de arranque nos puede proporcionar ventajas en varios frentes:

- crear un CD de instalación.
- tener el sistema operativo en cdrom y utilizar los discos duros para otros fines.
- tener un método de fácil actualización de sistemas, usando discos regrabables (CD-RW).

La idea, mirando al futuro de los sistemas informáticos, es poder contar con una computadora sin discos duros: utilizar el cdrom para el sistema operativo, un disco en memoria RAM para */tmp* y un sistema de ficheros remoto NFS para lo demás.

[18] http://www.knopper.net/knoppix
[19] http://directory.google.com/Top/Computers/Software/Operating_Systems/Linux/Distributions/Live_CD/?tc=1
[20] http://www.kenwoodtech.com/72x_atapi.html

### 7.3.1. Consideraciones previas

Es posible llegar a poner nuestra distribución favorita de manera que arranque desde cdrom, como veremos no supone un gran esfuerzo. Esto puede aportarnos las ventajas de configurar el kernel a nuesta manera, así como los módulos cargables y el entorno gráfico.

Para nuestro documento, y teniendo en cuenta que estamos sobrellevando todo esto mirando a la clusterización con openMosix, podremos llegar a poner en este CD un kernel con el soporte adecuado para él, y así poder incluir a nuestro cluster cualquier PC que tengamos conectado a la red, simplemente indicándole que arranque desde el cdrom.

Como puede deducirse a partir de lo visto hasta ahora, y para dejar claro que este sistema no va en detrimento de ningún recurso (sino al contrario) ni hace perder funcionalidades a los nodos, se indica que desde esta distribución podremos montar las particiones que tengamos en los discos locales de cada nodo, y así pues seguir con nuestras tareas mientras hemos convertido el PC en miembro de un cluster openMosix. Evidentemente podremos escribir en estas particiones montadas siempre que tengamos permiso para hacerlo.

En este documento se asume que el lector tiene conocimiento de utilidades como *cdrecord* para quemar cdroms o *mkisofs* para generar un sistema de ficheros ISO. No obstante, con los parámetros aquí utilizados debería ser suficiente para llegar a nuestra meta.

### 7.3.2. Dispositivos *ramdisk* en linux

Como ya hemos explicado, será necesario tener los conocimientos precisos para poder trabajar con */tmp* en RAM para poder disponer de un *lugar* donde nuestro sistema sea capaz de almacenar sus archivos temporales. Crear estos dispositivos y trabajar con ellos en linux es trivial. A continuación se intentará dar una visión global de este método volátil de almacenaje.

Antes de proceder, comprueba que estás actuando como administrador (root) de tu sistema.

#### Qué es un Ramdisk

Consideraremos un dispositivo en RAM cualquier porción de memoria que hayamos dispuesto para usar como una partición más en nuestro sistema de ficheros. Podremos verlos como discos duros virtuales.

¿Por qué puede interesarnos trabajar con estos dispositivos? De todos debe ser sabido que la memoria de nuestra computadora está dispuesta mediante una jerarquía en cuanto a la velocidad de acceso (parámetro directamente proporcional a su coste por octeto). Así pues, el acceso a memoria principal (RAM) será varias veces más rápido que el acceso a memoria secundaria (discos duros) y varias veces más lento que cualquier memoria cache. Si disponemos los ficheros que más habitualmente usaremos en dispositivos RAM -*ramdisks*-, podemos llegar a aumentar considerablemente el potencial de nuestra computadora -método ampliamente utilizado en grandes servidores web-.
Evidentemente y para el caso concreto que nos atañe, este dispositivo nos servirá como una unidad de almacenaje temporal para que nuestro sistema pueda trabajar localmente con sus ficheros temporales (valga la redundancia).

#### Como usar un Ramdisk

Su uso y configuración es altamente sencillo con linux: formatear tal dispositivo y montarlo en nuestro árbol de ficheros. Para ver todos los dispositivos ram con los que cuenta nuestro sistema solo tenemos que ejecutar:

```
ls -la /dev/ram*
```

No obstante, estos dispositivos se encuentran sin formato. Como se ha indicado, deberemos formatearlos y disponerlos en el árbol. Así el proceso con el primer dispositivo RAM */dev/ram0* sería:

- crear un directorio donde montar el dispositivo

```
mkdir -p /tmp/ramdisk0
```

- generar su sistema de archivos (i.e. formatearlo)

```
mkfs -t <ext2, ext3,...> /dev/ram0
```

- montar este sistema de ficheros en el árbol de directorios

```
mount /dev/ram0 /tmp/ramdisk0
```

Ahora podremos trabajar con él de forma usual; podremos percatarnos que los movimientos de datos entre este tipo de dispositivos son más rápidos. Es importante remarcar que **estos discos estan en memoria y por tanto al rearrancar la computadora PERDEREMOS los datos que les hubieramos grabado**.

Para comprobar que el montaje de las unidades ha sido correcto, podemos consultar el fichero */etc/mtab* o ejecutar el comando `df` para ver la relación de espacios libres y usados. Ambas salidas puedes encontrarlas en el apéndice *Salidas de comandos y ficheros*.

#### Cambiar el tamaño de los Ramdisks

Si hemos intentado generar un sistema de ficheros ReiserFS quizás nos hayamos topado con la desagadable sorpresa de no disponer de suficiente espacio para poder generar el *jounal* requerido. Los ramdisks vienen con un tamaño máximo, parámetro que evidentemente se puede modificar.

Hay 2 maneras de hacerlo. Veámoslas:

- editar el fichero */usr/src/linux/block/rd.c* y cambiar el valor de la variable
  `int rd_size = 4096; /*Size of the ramdisks */`
  por el valor el quilo-octetos deseado. Esta opción implica recompilar el kernel y rearrancar la máquina.
- la solución fácil es editar */etc/lilo.conf* y annexar una nueva linea `ramdisk=<N>` en la partición de arranque de linux para obtener un tamaño de *N* quilo-octetos. Luego deberemos ejecutar el comando `lilo` para actualizar los cambios.

#### Montar directorios en particiones Ramdisk

Para terminar se da un método fácil con el que podremos montar alguno de nuestros directorios en memoria. Esto es precisamente lo que utilizaremos para montar el */tmp* en ramdisk en el proceso de arranque, para que podamos escribir en él con nuestro *live linux cdrom*.

Las instrucciones son éstas:

- guardar una copia del directorio que queramos montar en memoria

```
mv /tmp /tmp_real
```

- crear un directorio con el nombre que linux espera

```
mkdir /tmp
```

- escribir los siguientes comandos para crear el FS y montarlo

```
/sbin/mkfs -t ext2 /dev/ram2
mount /dev/ram2 /tmp
```

- hacer la copia de contenidos desde disco a memoria

```
tar -C /tmp_real -c . | tar -C /tmp -x
```

- para comprobar que la cosa funciona podemos ver el fichero */etc/mtab* .

Estas lineas podremos utilizarlas igualmente en *scripts* para que nuestro linux haga estas operaciones, por ejemplo, al arranque. Esta técnica nos permitirá poder arrancar desde rom y normalmente se escriben en los ficheros del directorio */etc/rc.d/*.

Los contenidos en cada distribución linux puede cambiar levemente. Es questión de investigar un poco para saber el orden en como se cargan. Quizás el fichero que más pueda ayudarnos en esta tarea sea, dentro de tal directorio, *rc.sysinit*.

### 7.3.3. Modificaciones a linux

Ahora ya se han expuesto las herramientas necesarias para poder generar un cdrom arrancable, a falta de un método ordenado. Éste consistirá en crear una partición de repuesto en el disco duro para hacer pruebas, antes de desechar inutilmente cdroms que no den el resultado esperado. Se podrá arrancar desde la partición correspondiente aunque con la peculiaridad de montarse como de sólo lectura (*read only*) con el fin que funcione como una memoria rom. Una vez testeada la prueba final, podremos pasar la partición a cdrom.

Todo ello se irá viendo en esta sección; el esquema sobre el que basaremos la metodología podría quedar así:

- conectar un disco duro y una grabadora de cdroms a la computadora que luego desearemos arrancar.
- particionar el disco de manera que dejemos al menos una partición (`test`) del tamaño del cdrom que vamos a usar (usualmente 700MB) y otra (`principal`) para realizar una instalación *normal* de Linux, desde donde generaremos la imagen arrancable y quemaremos el cdrom.
- se requerirán pues dos instalaciones de nuestro linux: una en la partición que contendrá lo que luego volcaremos al cdrom, y otra con soporte para quemar cdroms. Opcionalmente pueden copiarse en `test` los directorios que permitan arrancarlo, desde la instalación hecha en la partición principal; esto nos ahorrará hacer dos instalaciones.
- en `principal` montaremos `test`. En este ejemplo se usará la ruta absoluta */test*, teniendo en cuenta que `principal` está montada en / .
- en el caso de que el kernel nos venga sin estas opciones (rara vez ocurre) arrancaremos `test` y recompilaremos el kernel con soporte de *isofs* (el sistema de ficheros iso9660 es el estándar en cdroms) y soporte para cdrom, incluidos y NO como módulos.
- */tmp* se montará en un *ramdisk* y apuntaremos */var* hacia */tmp/var*, teniendo en cuenta que estamos hablando de la partición montada en */test*. Estos dos directorios son los que linux requiere como de lectura y escritura para mantener su coherencia. Configuraremos los ficheros necesarios para ajustarse a esta nueva situación del sistema.
- para el arranque del kernel hace falta echar mano de un disquete arrancable y configurarlo para utilizar el cdrom como dispositivo `root (/)`. Ya se verá más detalladamente.
- finalmente quemaremos un cdrom con *cdrecord*.

#### Cambios en la estructura de directorios

Para empezar con los cambios en la partición `test` deberemos tener claro lo que hay que hacer, esto es, ser capaces de entender el proceso y no tanto de aplicar ciertas modificaciones a ciertos ficheros obviando lo que ello pueda significar. Este último método sin duda acortaría notablemente la duración del capítulo, no obstante y como Vd. querido lector ya sabrá, las estrucutras de directorios y nombres de los ficheros que contienen pueden sufrir variaciones entre distribuciones (y entre versiones de las mismas).

Suponiendo que se ha comprendido el funcionamiento de los *ramdisk* y que se tiene cierta soltura con `lilo` para arrancar sistemas linux, empezaremos a moldear los ficheros para que sean capaces de mantener funcionando nuestro equipo aun estando, o almenos en parte, en un dispositivo rom.

Quede claro pues que se modificará la estructura de archivos de la partición `test` para que pueda arrancarse con los permisos necesarios en una unidad de sólo lectura, como es un cdrom. Las rutas que se dan suponen que se está trabajando desde `test`. El esquema de las modificaciones que le aplicaremos es el siguiente:

1. Apuntar */var* a */tmp/var* . El objetivo del enlace puede no existir aún, pero se generará en el arranque gracias a un *script*. De momento lo veremos como un enlace roto.

   ```
   ln -s /tmp/var /var
   ```

   */var* será otro directorio en el que será necesario poder escribir. Se construye esta estructura -apuntándolo hacia */tmp/var*- aunque evidentemente podríamos generar otro *ramdisk* y montar var en él. Esto son decisiones de diseño.

2. Configurar el *runlevel* a 1. Bastará editar el fichero */etc/inittab* y cambiar la linea

   `id:<N>:initdefault:` donde *N* será seguramente 3 ó 5

   por

   `id:1:initdefault:`

   Dando un repaso rápido a los *runlevels*, podemos verlos como diferentes niveles de arranque del linux en nuestro equipo. El modo `1` es el modo monousuario (ingl. *single user mode*), el modo `3` es el modo multiusuario completo con red (ingl. *full multiuser with network*) y el `5` es el modo multiusuario completo con red y gestor de escritorio (ingl. *full multiuser with network and xdm*). Hay 6 niveles pero en los arranques normales de linux se utilizan básicamente el 3 y el 5. Trasladar linux a *runlevel* 1 comportará que no se requiera login ni contraseña para ejecutar el `shell`.

   **¿Por qué es necesario trabajar con runlevel 1?** Un sistema multiusuario necesita poder guardar información sobre los errores del servidor X-window de cada usuario que lo utilice, guardar información de los comandos de la consola y toda una retahíla de información que debe poder escribirse. En sistemas *live*, donde se depende de un dispositivo mucho más lento que un disco duro, sin memoria virtual (*swap*) y donde necesitamos la memoria primaria para poder cargar directorios para escritura, es mucho más cómodo trabajar en monousuario.

3. Borrar (o comentar con # al inicio) en el archivo */etc/fstab* cualquier referencia a dispositivos de memoria virtual *swap*. Estas lineas, en discos duros IDE, se muestran como algo parecido a:

   ```
   /dev/hd<X>    swap      swap     pri=42     0 0
   ```

   El otro cambio que se efectuará en *fstab* es el dispositivo para montar / . En principio será la partición donde esté localizada `test` pero al trasladar los cambios al cdrom deberemos indicar aquí `hda, hdb, hdc` o `hdd` según donde tengamos conectado el lector. Esto sería, en un caso concreto, pasar de:

   ```
   /dev/hdb3    /    reiserfs   defaults     1 1
   /dev/hdc     /    iso9660    defaults     1 1
   ```

   Puede ser una buena idea escribir ambas líneas y comentar la segunda, de cara a recordar que más tarde deberemos pasar el comentario a la primera linea para poder arrancar correctamente. Puede observarse que el sistema de ficheros se adaptará al nuevo dispositivo rom, así que independientemente del formato de nuestra unidad en disco duro (ext2, ext3, xfs, reiserfs...) el cdrom se quemará con el estándar iso9660. Ya se ha advertido anteriormente que es importante tener soporte para él en el mismo kernel.

4. Aumentar el tamaño de los *ramdisks*, añadiendo en el fichero *lilo.conf*

   ```
   ramdisk = 35000
   ```

   dentro de la entrada que se corresponda con el arranque de la partición `test`. No olvidemos ejecutar el comando `lilo` para actualizar los cambios.

   Este parámetro solo tendrá efecto para el arranque de la partición desde disco duro, más tarde se detallará dónde debe incluirse esta linea para que tenga efecto en el arranque *live*.

5. Enlazar *mtab*. Este fichero se actualiza cada vez que se monta o desmonta un dispositivo, por lo tanto debemos poder modificarlo aún estando en el lugar donde linux lo buscará (en */etc*). ¿Cómo hacer esto? Convirtiendo *mtab* en un enlace simbólico hacia */proc/mounts*, fichero al cual empezaremos actualizándole las fechas de acceso

   ```
   touch /proc/mounts
   rm -rf /etc/mtab
   ln -s /proc/mounts /etc/mtab
   ```

   En algunos *scripts* aparecen las lineas

```
# Clear mtab
>/etc/mtab
```

para reinicializar el fichero en cuestión. Si este cambio se aplica tras haber montado nuestros dispositivos en memoria, puede darse el caso de que nuestro equipo no los vea, algo que se traduce en una inconsistencia mediana o grave. Lo mejor será comentarlo (añadiendo # al inicio de la linea).

Es importante trabajar desde `test` para generar los enlaces, puesto que si creamos los enlaces con rutas desde `principal` (como */test/var* o *./tmp/var*) la partición `test` no los reconocería correctamente puesto que ella, al arrancar, tendrá */var* montado en */tmp/var* y no en *./tmp/var* (partiendo de */etc* o de */test*).

6. Convertir la unidad en acceso de sólo lectura es fácil puesto que en estos *scripts* de iniciación de remonta / para hacerla de lectura y escritura. Las lineas que se corresponden a esta operación deben ser algo parecido a

```
# Remount the root filesystem read-write.
state=‘awk ’/(ˆ\/dev\/root| \/ )/ { print $4 }’ /proc/mounts‘
[ "$state" != "rw" ] && \
  action $"Remounting root FS in read-write mode: " mount -n -o remount,rw /
```

que debería sustituirse por

```
# Remount the root filesystem read-write.
state=‘awk ’/(ˆ\/dev\/root| \/ )/ { print $4 }’ /proc/mounts‘
[ "$state" != "rw" ] && \
  action $"Remounting root FS in read-only mode: " mount -n -o remount,ro /
```

igualmente debería anularse cualquier referencia a memoria virtual (*swap*). Como puede apreciarse en el siguiente ejemplo, la linea se ha comentado para que no sea ejecutada.

```
# Start up swapping.
# action $"Activating swap partitions: " swapon -a -e
```

Por regla general en estos *scripts* no hay ninguna referencia más a dispositivo de lectura y escritura. No obstante estaría bien tomar paciencia y repasarlos de arriba a abajo. Generalmente estan muy bien comentados -en inglés- ahorrándonos muchos dolores de cabeza.

7. Finalmente tendrá que generarse el *ramdisk* y su árbol de directorios para que pueda guardar todas las entradas en */tmp* y */var* que el sistema requiera[21]. Podemos editar */etc/rc.sysinit* o cualquier script que venga con nuestra distribución y al que se invoque en el arranque.

   Las siguientes lineas deberán incluirse almenos tras configurar el PATH, i.e. allá donde linux irá a buscar los ficheros binarios `mkdir` y `mkfs` (entre otros) necesarios apra crear directorios y sistemas de archivo, respectivamente). Esta linea puede aparecer como

   ```
   # Set the path
   PATH=/bin:/sbin:/usr/bin:/usr/sbin
   export PATH
   ```

   Una buena forma de asegurarnos de no cambiar los permisos del *ramdisk* y mantener la coherencia de *mtab* es incluir estas lineas directamente a continuación de remontar / como sólo lectura.

   *a*) crearemos el dispositivo en memoria y le daremos formato

   ```
   action "Making ram0 " /sbin/mkfs.ext3  /dev/ram0
   action "Mounting ram0" mount /dev/ram0 /tmp -o defaults,rw
   ```

[21] estos requerimientos los aportan las inicializaciones de los propios *scripts* de inicio.

*b*) crearemos el árbol de directorios

```
action "Making /tmp/etc" mkdir /tmp/etc
action "Making /tmp/var" mkdir /tmp/var
action "Making /tmp/var/log" mkdir -p /tmp/var/log
action "Making /tmp/var/run" mkdir -p /tmp/var/run
action "Making /tmp/var/lock" mkdir -p /tmp/var/lock
action "Making /var/lock/subsys" mkdir -p /var/lock/subsys
action "Making /tmp/var/lib" mkdir -p /tmp/var/lib
action "Making /tmp/var/spool" mkdir -p /tmp/var/spool
action "Making /tmp/var/spool/mqueue" mkdir -p /tmp/var/spool/mqueue
```

*c*) cambiaremos los permisos para poder escribir en ellos

```
action "Chmod 777 /tmp " chmod 777 /tmp
action "Chmod +t /tmp" chmod +t /tmp
```

*d*) haremos un nuevo dispositivo *ramdisk* y le copiaremos el contenido de */dev*

```
echo "mkdir /tmp/dev" mkdir -p /tmp/dev
echo "Making ram1 " /sbin/mkfs.ext2  /dev/ram1
echo "Mounting ram1" mount /dev/ram1 /tmp/dev
echo "cp -a /dev/* /tmp/dev/" cp -a /dev/* /tmp/dev/ > /dev/null
echo "umounting /tmp/dev" umount /tmp/dev
echo "remounting /tmp/dev to /dev" mount /dev/ram1 /dev
```

Tras esto `test` debería arrancar sin mayores problemas. Es normal que se nos quede colgada en algunos de los muchos ensayos que vamos a hacer hasta llegar a este resultado. No obstante debe avanzarse siguiendo una lógica y no suponiendo que *tenemos algo que no lo hace funcionar*. Algunos *scripts* necesitarán llegar a cierta estructura de árbol (en */var* o */tmp*) y en caso de que no lo hayamos generado en el dispositivo en RAM nos devolverá un error. Simplemente añadiendo y/o modificando las líneas del punto *b)* para creación de directorios podremos arreglar esto.

¿Por qué no se ha hablado de */proc*? Este directorio contiene información sobre lso procesos que corren en nuestro equipo, pero ya por defecto se monta en memoria. Por esta razón -y no por otra- no deberemos encargarnos de él.

#### Arrancando un kernel con una imagen de disquete

Para proceder con estos, los últimos pasos para obtener el cdrom, es mejor reiniciar el sistema y trabajar desde `principal` (siempre teniendo `test` en */test*).

Para generar disquetes arrancables existen varios métodos. Quizá el más simple y efectivo sea obtenerlo directamente del cdrom de alguna distribución. Generalmente aparecen con nombres como *boot.img* o *bootdisk* y siempre tienen un tamaño de 1,4 mega-octetos. De forma alternativa podremos generarlo nosotros mismos.

- hacer una copia del kernel

  `cp /boot/vmlinuz-<version_del_kernel> /tmp/vmlinuz`

- hacer la copia arrancable del cdrom en */dev/<dispositivo_cdrom>*

  `rdev /tmp/vmlinuz /dev/<dispositivo_cdrom>`

  `ramsize /tmp/vmlinuz 20000`

- formatear un disquete y copiar el kernel

  `mkfs -t ext2 /dev/fd0`

  `dd if=/tmp/vmlinuz of=/dev/fd0`

En el caso de que se disponga de la imagen de un disquete arrancable, deberemos modificar ciertos parámetros. Para hacerlo deberemos montar esa imagen:

```
mount <fichero_imagen> <directorio_de_montaje> -o loop -t vfat
```

En <*directorio_de_montaje*> debería aparecernos almenos: una imagen binaria del kernel de linux (`vmlinuz`), algunos ficheros binarios para los menús de pantalla y los ficheros que consideraremos importantes, `ldlinux.sys` y `syslinux.cfg`. Deberemos editar el segundo para personalizarlo a nuestro sistema. Básicamente deberá contener la etiqueta del arranque y el nombre de la imagen, por ejemplo:

```
default linux

    label linux
    kernel vmlinuz
    root=/dev/<dispositivo_cdrom>
    ramdisk=35000
```

Si estamos familiarizados con LiLO no nos será difícil configurar este fichero. La otra cosa importante es sustituir `vmlinuz` por la imagen que arranque `test` (y su fichero *initrd* en caso de que exista; ambos los encontraremos en */test/boot/*). En el apéndice *Salidas de comandos y ficheros* podrás ver un caso concreto de este archivo[22]. Ahora desmontaremos la imagen.

```
umount <directorio_de_montaje>
```

Y copiaremos esta imagen de disquete arrancable al directorio */test*. El kernel embebido aquí será el que arrancará el sistema.

Un testeo interesante antes de quemar el cdrom es probar si la imagen de disquete que hemos creado funciona. La manera rápida y eficaz de hacerlo es copiar dicha imagen con el mismo comando dado para copiar el kernel aunque indicando el fichero adecuado a *if*, así `dd if=bootdisk of=/dev/fd0` . Podemos arrancar la computadora y si este arranque parece funcionar (¡no tendrá sistema de ficheros!), probaremos con el cdrom.

AHORRANDO UNA SEGUNDA INSTALACIÓN. COPIA DE CONTENIDOS

Una variante de este esquema podría darse cuando no hayamos cumplido un proceso de instalación en `test`. Tendremos que tener en cuenta que podremos aprovechar la mayor parte de los directorios de la partición `principal` (siempre y cuanto quepan en el directorio */test*). En el árbol de directorios, excepto */usr*, el contenido de los demás directorios no será crítico. En */test* podremos disponer la estructura copiando los directorios desde `principal` y arreglando los ficheros necesarios en */etc* .

Para generar los directorios en `test`, por ahora vacía, podríamos ejecutar:

```
cd /test
mkdir root
mkdir mnt
mkdir proc
mkdir tmp
mkdir home
mkdir misc
mkdir opt
mkdir dev
```

E iniciamos la copia de contenidos con `rsync`. Alternativamente podríamos usar el comando `cp -a`.

```
rsync -a /dev/* dev
mkdir lib
rsync -a /lib/* lib
mkdir bin
rsync -a /bin/* bin
```

[22] cabe señalar que pararlas pruebas en la partición `test` deberemos indicar a la etiqueta `root=` la partición correcta.

```
mkdir sbin
rsync -a /sbin/* sbin
mkdir usr
mkdir etc
rsync -a /etc/* etc
mkdir boot
rsync -a /boot/* boot
```

### 7.3.4. Creando el cdrom

Antes de empezar con el proceso de quemado es necesario recordar que hay que modificar el fichero */test/etc/fstab* para indicarle a linux que el dispositivo root ahora no será la partición `test` sino el cdrom (con iso9660).

Para disponer en un cdrom todos los contenidos que hasta ahora hemos modificado seguiremos los siguientes pasos:

- crear la imagen iso9660 llamada boot.iso (situados en el directorio donde queramos generarla)

  ```
  mkisofs -R -b boot.img -c boot.catalog -o boot.iso /test
  ```

- verificar la imagen iso, montándola en algun directorio

  ```
  mount boot.iso <directorio_de_montaje> -o loop -t iso9660
  ```

- quemar el CD

  ```
  cdrecord -v speed=<velocidad> dev=<ruta_del_grabador> boot.iso
  ```

- verificar el arranque desde cdrom, re-arrancando la computadora.

### 7.3.5. Últimos detalles

Puede ser interesante, desde el punto de vista práctico, mostrar una pantalla antes de empezar con el arranque del kernel, para poder elegir las opciones con las que va a hacerlo. Esto es modo gráfico, tamaño de los ramdisks, *etc...*

Por el momento el espacio disponible en un disquete no nos permite poder embeber en él varios kernels de tamaño tradicional. No obstante si se trabaja con sistemas con pocos requerimientos se podrán arrancar varias imágenes `vmlinuz`. El fichero *syslinux.cfg* se maneja como el *lilo.conf* así que no debería darnos más problemas. En el apéndice *Salidas de comandos y ficheros* se adjunta un ejemplo.

Para mostrar un menú podemos añadir, a la cabecera de *syslinux.cfg*, la linea

```
display boot.msg
```

donde *boot.msg* será la pantalla a mostrar. Este fichero podemos editarlo con un editor de texto llano, aunque contiene algunos bytes en binario que deberemos respetar. Igualmente podremos llamar otros menús a partir de éste, con las teclas de función. Estas configuraciones también irán a la cabecera de *syslinux.cfg*[23].

Las posibilidades que brinda este cargador pueden embellecerse usando los 16 colores que cualquier targeta VGA soporta. A continuación se da un ejemplo de un menú que mostraría por pantalla, en cada color, el número hexadecimal que lo codifica.

```
^L
Ejemplo de colores para el mensaje de arranque desde un disquete botable:^007
         ^001 azul claro=1 ^007
         ^002 verde=2 ^007
         ^003 azul claro=3 ^007
         ^004 rojo oscuro=4 ^007
         ^005 purpura=5 ^007
         ^006 marron=6 ^007
```

[23] en el apéndice *Salidas de comandos y ficheros* podeos ver un ejemplo del fichero completo.

```
^007 gris=7 ^007
^008 verde oscuro=8 ^007
^009 azul=9 ^007
^00a verde claro=A ^007
^00b azul claro=B ^007
^00c rojo=C ^007
^00d lila=D ^007
^00e amarillo=E ^007
^00f negrita=F ^007
```

La estructura de estos ficheros es:

- `^L`

  limpia la pantalla. Es binario así que algun editor puede mostrarlo distinto a como aquí se muestra (con `emacs`). Para asegurar que se está tomando el carácter correcto lo mejor es hacerse con un fichero ya existente en cualquier liveCd y copiarlo.

- los colores se codifican con la secuencia

  `^00x`

  donde `x` es un dígito hexadecimal (del 1 al F); y los saltos de linea son

  `^007`

El carácter con el que empiezan los código hexadecimales no es el acento circunflejo, así que si probamos de escribir el texto anterior tal cual no funcionará. Como se ha descrito, lo mejor es tomar ficheros ya existentes y copiar dichos códigos.

Para terminar con este apartado cabe decir que SuSE no ha reparado en esfuerzos hasta llegar a conseguir mostrar una inmejorable imagen de gecko (el camaleón verde que les sirve de logotipo) en nuestro sistema. Su `gfxboot` permite dispone las opciones como un menú a todo color. Para más detalles mejor consultar su *syslinux.cfg* .

## 7.4. Referencias


- *'First Attempt at Creating a Bootable Live Filesystem on a CDROM'* de Mark Nielsen
  http://www.linuxgazette.com/issue54/nielsen.html
- *'How to use a Ramdisk for Linux'* de Mark Nielsen
  http://www.tcu-inc.com/mark/articles/Ramdisk.html
- *'Diskless Nodes HOW-TO document for Linux'* de Robert Nemkin, Al Dev, Markus Gutschke, Ken Yap y Gero Kuhlmann
  http://www.tldp.org/HOWTO/Diskless-HOWTO.html
- *'The Linux Bootdisk HOWTO'* de Tom Fawcett
  http://www.tldp.org/HOWTO/Bootdisk-HOWTO/index.html
- *'Creating Installation Cds from various Linux Distributions'* de Mark Nielsen y Krassimir Petrov
  http://www.tcu-inc.com/mark/articles/cdburn.html
- *'The Linux BootPrompt-HowTo'* de Paul Gortmaker
  http://www.ibiblio.org/mdw/HOWTO/BootPrompt-HOWTO.html
- *'CD-Writing HOWTO'* de Winfried Trümper
  http://www.tldp.org/HOWTO/CD-Writing-HOWTO.html
- *'LILO mini-HOWTO'* de Miroslav "Misko"Skoric
  http://www.tldp.org/HOWTO/mini/LILO.html
- Como siempre, las fuentes de linux vienen con abundante y útil documentación
  /usr/src/linux/Documentation/ramdisk.txt
- Lista con muchas referencias sobre protocolos y aplicaciones
  http://www.fokus.gmd.de/research/cc/glone/employees/joerg.schilling/private/cdr.html

# Capítulo 8

# Apéndices

## 8.1. APÉNDICE A: Aplicaciones funcionando, o no

Este apéndice contendrá una lista de aplicaciones que han sido probadas y otra que mostrará las NO pueden funcionar, en openMosix.

## Programas que funcionan

- **Utiliades MPEG.** Utilizan *pipes* de entrada y salida (i/o) intensivamente, hecho que hace que funcionen excepcionalmente bien en clusters de pequeño y mediano tamaño. La codificación y decodificación se realizan en el nodo raíz pero los filtros migrados en los nodos incrementan el rendimiento en la compresión de ficheros de alta calidad (que requieren un mayor procesamiento).
- **bladeenc.** Esta utilidad sirve para ripear rápidamente nuestros ficheros de audio mp3.
- **Povray.** Podemos dividir nuestros frames del trabajo de renderización en múltiples procesos desde un shell script o utilizar la versión paralela (PVM) para que se haga automáticamente.
- **MPI.** Entre MPI y openMosix parece existir una totalidad compatibilidad.
- **FLAC.** Se trata de un codificador de audio *lossless* (sin pérdidas). `http://flac.sourceforge.net/`

Podremos encontrar información más actualizada en las referencias Wiki del HOWTO de Kris Buytaert[1].

## Programas que NO funcionan

Las aplicaciones que utilizan memoria compartida funcionan en un linux estándar, pero no migran. Tampoco podrán migrar pogramas que hagan uso de recursos que no puedan migrar, como por ejemplo pasa con todas las aplicaciones que puedan apoiarse en Java green thread JVM's.

- **Programas Java que utilizan *threads* nativos.** Estos programas no migrarán porque utilizan memoria compartida. Los *Green Threads JVM's* permiten migración pero cada thread en un proceso separado.
- **Aplicaciones que utilizan threads.** El hecho de que no migren los threads no es una limitación de openMosix sinó de Linux. Contrariamente a las plataformas como Solaris donde los threads son procesos *ligeros* con su propio espacio de memoria, en Linux no poseen dicho espacio. Si hacemos un `ps` en Linux podremos ver cada thread ya que cada uno será una tarea en el scheduler. No obstante cada una de estas tareas no puede funcionar por ella misma ya que necesita el espacio de direcciones donde fue lanzada. De esta manera queda pues como hecho imposible poder migrar *pthread* a otro nodo sin poderlo conectar son su espacio de direcciones.
- **Phyton** con el threading activado.
- **MySQL** utiliza memoria compartida.
- **Apache** utiliza memoria compartida.
- **Mathematica** utiliza memoria compartida.
- **SAP** utiliza memoria compartida.
- **Oracle** utiliza memoria compartida.
- **Baan** utiliza memoria compartida.
- **Postgres** utiliza memoria compartida.

Podremos encontrar información más actualizada en la página Wiki[2].

[1] `http://howto.ipng.be/openMosixWiki/index.php/work%20smoothly`
[2] `http://howto.ipng.be/openMosixWiki/index.php/don't`

## 8.2. APÉNDICE B: Salidas de comandos y ficheros

En este apéndice podrás encontrar ficheros completos de configuración y salidas de algunos de los comandos que han surgido al largo del documento.

Debe tenerse en cuenta que esto implica que no estan generalizados sino que simplemente se corresponden a las distintas configuraciones de `mCii`, mi computadora. No obstante puede ayudar para hacer comprender al lector el tipo de salidas y configuraciones (y la consistencia entre ellas).

### 8.2.1. `lspci`

Nos da una lista de todos los dispositivos hardware de que disponemos conectados al bus PCI.

```
root@mCii:~ # lspci
00:00.0 Host bridge: VIA Technologies, Inc. VT8367 [KT266]
00:01.0 PCI bridge: VIA Technologies, Inc. VT8367 [KT333 AGP]
00:08.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-8139/8139C/8139C+ (rev 10)
00:09.0 Multimedia video controller: Brooktree Corporation Bt878 Video Capture (rev 11)
00:09.1 Multimedia controller: Brooktree Corporation Bt878 Audio Capture (rev 11)
00:0a.0 FireWire (IEEE 1394): VIA Technologies, Inc. IEEE 1394 Host Controller (rev 43)
00:0b.0 Multimedia audio controller: ESS Technology ES1978 Maestro 2E (rev 10)
00:11.0 ISA bridge: VIA Technologies, Inc. VT8233 PCI to ISA Bridge
00:11.1 IDE interface: VIA Technologies, Inc. VT82C586B PIPC Bus Master IDE (rev 06)
00:11.2 USB Controller: VIA Technologies, Inc. USB (rev 18)
00:11.3 USB Controller: VIA Technologies, Inc. USB (rev 18)
00:11.4 USB Controller: VIA Technologies, Inc. USB (rev 18)
00:11.5 Multimedia audio controller: VIA Technologies, Inc. VT8233 AC97 Audio Controller (rev 10)
01:00.0 VGA compatible controller: nVidia Corporation NV25 [GeForce4 Ti4200] (rev a3)
```

### 8.2.2. */proc/bus/pci/devices*

Nos da una lista de todos los identificadores de los dispositivos hardware de que disponemos conectados al bus PCI.
Hay una correspondencia con la salida de `lspci` y puede servirnos para ver los modelos de componentes de cada dispositivo (segunda columna).

### 8.2.3. *letc/mtab* y df

Para ver como quedan montados los dispositivos en nuestro sistema podemos ver el fichero mtab .

```
root@mCii:~ # less /etc/mtab
/dev/hdb3 /             reiserfs rw 0 0
proc      /proc         proc     rw 0 0
devpts    /dev/pts      devpts   rw,mode=0620,gid=5 0 0
/dev/hda1 /windows/C    vfat     rw,noexec,nosuid,nodev,gid=100,iocharset=iso8859-1,code=437 0 0
/dev/hda5 /windows/D    vfat     rw,noexec,nosuid,nodev,gid=100,iocharset=iso8859-1,code=437 0 0
/dev/hda6 /windows/E    vfat     rw,noexec,nosuid,nodev,gid=100,iocharset=iso8859-1,code=437 0 0
/dev/hda7 /windows/F    vfat     rw,noexec,nosuid,nodev,gid=100,iocharset=iso8859-1,code=437 0 0
/dev/hdb5 /debian       reiserfs rw,noexec,nosuid,nodev 0 0
/dev/hdb6 /test         ext3     rw 0 0
/dev/hdb7 /container    reiserfs rw 0 0
shmfs     /dev/shm      shm      rw 0 0
usbdevfs  /proc/bus/usb usbdevfs rw 0 0
/dev/ram1 /tmp/ramdisk1 ext3     rw 0 0
/dev/ram2 /tmp/ramdisk2 ext2     rw 0 0
/dev/ram0 /tmp/ramdisk0 ext3     rw 0 0
```

O también ejecutar el comando df para ver el tipo y tamaño de cada partición:

```
root@mCii:~ # df
Filesystem           1K-blocks      Used Available Use%  Mounted on
/dev/hdb3              6184828   2639964   3544864  43%  /
/dev/hda1              1036021    818990    217031  80%  /windows/C
/dev/hda5             33824576  17391872  16432704  52%  /windows/D
/dev/hda6              3142548   1157896   1984652  37%  /windows/E
/dev/hda7              1042136    918212    123924  89%  /windows/F
/dev/hdb5              6032184   1336996   4695188  23%  /debian
/dev/hdb6               695684    413944    246400  63%  /test
/dev/hdb7             16273308   2396104  13877204  15%  /container
shmfs                   127984         0    127984   0%  /dev/shm
/dev/ram1                 9677      1043      8134  12%  /tmp/ramdisk1
/dev/ram2                38733        13     36720   1%  /tmp/ramdisk2
/dev/ram0                48409      4127     41782   9%  /tmp/ramdisk0
```

### 8.2.4. */etc/lilo.conf*

Este fichero se encuentra en */etc* y permite configurar LILO para el arranque de los sistemas que tengamos instalados.

```
lba32
boot=/dev/hda
install=/boot/boot-menu.b
map=/boot/map
delay=20
message=/boot/xray-blue.boot

prompt
timeout=150
default=SuSE-8.1

# particion principal
image=/boot/vmlinuz
append=" hdd=ide-scsi"
label=SuSE-8.1
root=/dev/hdb3
initrd=/boot/initrd
vga=792
read-only

image=/debian/boot/vmlinuz-2.4.18-bf2.4
label=Debian-Woody
root=/dev/hdb5
vga=792
read-only

image=/boot/vmlinuz.shipped
append=" hdd=ide-scsi"
label=SuSE-8.1-safe
root=/dev/hdb3
initrd=/boot/initrd.shipped
vga=792
read-only

image=/debian/boot/vmlinuz-2.4.18-bf2.4.old
label=Woody-OLD
root=/dev/hdb5
vga=792
read-only
optional

# esta imagen se corresponde con la particion utilizada
# para construir un Live Linux CD
image=/test/boot/vmlinuz
label=LiveCD
root=/dev/hdb6
ramdisk=35000
vga=792
read-only
optional
```

```
other=/dev/hdb1
  label=BeOS-1.1
  restricted

other=/dev/hda1
  label=Redmond

image=/boot/memtest.bin
  label=memTest86
```

### 8.2.5. syslinux.cfg

El fichero *syslinux.cfg* se incluye dentro de la imagen de disquete arrancable proporcionada por varias distribuciones en sus cdroms de muestra. Podemos verlo como un *lilo.conf* para arranques de cdroms y disquetes.

En el ejemplo podemos ver algunos cambios para adaptarlo a nuestras necesidades, tal como se desarrolla en el capítulo referente a *Live Linux CD!*

```
default linux
prompt 1
timeout 600
display boot.msg

 #indicamos las teclas de funcion que invocaran diferentes pantallas
F1 boot.msg
F2 general.msg
F3 param.msg
F4 colors.msg

label linux
  kernel vmlinuz
  append ramdisk_size=35000 vga=791 root=/dev/hdc
label text
  kernel vmlinuz
  append text ramdisk_size=8192 root=/dev/hdc
label expert
  kernel vmlinuz
  append expert ramdisk_size=8192 root=/dev/hdc
label nofb
  kernel vmlinuz
  append nofb ramdisk_size=8192 root=/dev/hdc
label lowres
  kernel vmlinuz
  append lowres ramdisk_size=8192 root=/dev/hdc
```

### 8.2.6. rpld.conf

```
HOST
{
        ethernet = 00:50:FC:B1:B1:BD;

        FILE
        {
                path = "/rplboot/eb-5.0.10-rtl8139.lzrom";
                load = 0x1000;
        };

        execute = 0x1006;
        framesize = 1500;
        blocksize = 1440;
};
```

### 8.2.7. dhcpd.conf

```
default-lease-time 600; # 10 min
max-lease-time 7200;  # 2 hores

# If this DHCP server is the official DHCP server for the local
# network, the authoritative directive should be uncommented.
authoritative;

# You must add a ddns-update-style statement to /etc/dhcpd.conf.
# To get the same behaviour as in 3.0b2pl11 and previous
# versions, add a line that says "ddns-update-style ad-hoc;"
# Please read the dhcpd.conf manual page for more information. **
ddns-update-style ad-hoc;


    subnet 192.168.1.0 netmask 255.255.255.0 {
option subnet-mask 255.255.255.0;
option domain-name openMo6;
range dynamic-bootp 192.168.1.1 192.168.1.10;
   }

    group{
#    filename <imatge kernel TFTP diskless>

host metrakilate {
  hardware ethernet 00:50:FC:B1:B1:BD;
  fixed-address 192.168.1.2;
  filename "/tftpboot/vmlinuz-metrakilate";
  option root-path "192.168.1.1:/tftpboot/metrakilate";
}

host mCi {
  hardware ethernet 00:E0:7D:D5:25:3A;
  fixed-address 192.168.1.3;
  filename "/tftpboot/vmlinuz-mCi";
  option root-path "192.168.1.1:/tftpboot/mCi";
}

    }
```

## 8.3. APÉNDICE C: Acrónimos

**API** - Application Programming Interface
**ATM** - Asynchronous Transmission Mode
**BOOTP** - Bootstrap Protocol
**broadcast** - dirección por la que todos los elementos de una red reciben la información
**CC** - Cache Coherent
**COTS** - Common Off The Shelf, *componentes comunes en el mercado*
**CPU** - Central Process Unit, *unidad central de procesamiento*
**DFSA** - Direct File System Access
**DHCP** - Dynamic Host Configuration Protocol
**DIPC** - Distributed Inter Process Communication
**diskless** - sin discos
**DMA** - Direct Memory Access, *acceso directo a memoria*
**ESS** - Earth & Space Sciences
**firewall** - elemento de seguridad para el acceso a una red. Identifica los paquetes de llegada o salida y gestiona su paso a través de él.
**FS** - File System, *sistema de ficheros*
**gateway** - pasarela, se utiliza en el contexto de redes
**GID** - Group Identification
**GUI** - Graphical User Interface
**IP** - Internet Protocol
**IPC** - Inter Process Communication
**kernel** - núcleo del sistema operativo
**LVS** - Linux Virtual Machine
**MFS** - Mosix File System
**MHz** - Megahercios
**mknbi** - make kernel net boot image
**MPI** - Message Passing interface
**NFS** - Network File System, *sistema de ficheros en red*
**NIC** - Network Internet Controller
**NOW** - Network of Workstations, *red de estaciones de trabajo*
**NUMA** - Non Uniform Memory Access
**path** - ruta en el árbol de directorios de un sistema de ficheros
**PC** - personal Computer
**POSIX** - Portable Operating System based on UNIX
**PPM** - Preferent Process Migration
**PVM** - Parallel Virtual Machine
**PVFS** - Parallel Virtual File System
**RAM** - Random Access Memory, *memoria de acceso aleatorio*
**RARP** - Reverse Address Resolution Protocol
**RPL** - Remote Program Load
**SCI** - Scalable Coherent Interface
**SMP** - Symmetric Multi Process, máquinas con varios procesadores en la misma placa madre
**SO** - Sistema Operativo
**SSI** - Single System Image, permite ver un cluster com un único equipo
**sniffer** - trazador de paquetes para monitorizar el tránsito de una red
**SWAP** - parte del disco duro utilizada como memoria virtual
**TCP** - Transfer Control Protocol, *protocolo fiable orientado a bytes para intercambio de información entre ordenadores que establecen una conexión*
**TFTP** - Trivial FTP
**trashing** - paginación excesiva por parte de un proceso
**UDP** - User Datagram Protocol
**UHN** - Unique Home Node
**UID** - User Identification
**VFS** - Virtual File System

# Capítulo 9

# GNU Free Documentation License

*Software is like sex: it's better when it's free.*

*Linus Torvalds*

GNU Free Documentation License
Version 1.2, November 2002



## 9.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of copyleft , which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 9.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The Document , below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as you . You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A Modified Version of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A Secondary Section is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The Invariant Sections are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The Cover Texts are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A Transparent copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available

drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not Transparent is called Opaque.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The Title Page means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, Title Page means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section Entitled XYZ means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as Acknowledgements, Dedications, Endorsements, or History.) To Preserve the Title of such a section when you modify the Document means that it remains a section Entitled XYZ according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 9.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 9.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one

year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 9.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section Entitled History, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled History in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the History section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. For any section Entitled Acknowledgements or Dedications, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section Entitled Endorsements. Such a section may not be included in the Modified Version. N. Do not retitle any existing section to be Entitled Endorsements or to conflict in title with any Invariant Section. O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled Endorsements, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 9.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled History in the various original documents, forming one section Entitled History; likewise combine any sections Entitled Acknowledgements, and any sections Entitled Dedications. You must delete all sections Entitled Endorsements.

## 9.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 9.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an aggregate if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 9.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled Acknowledgements, Dedications, or History, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically

terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 9.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or any later version applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

# Bibliografía

[SCP] Kai Hwang y Zhiwei Xu, *Scalable Parallel computing,* Ed. McGraw-Hill.

[SO] William Stallings, *Sistemas Operativos.*

[SOCD] Milan Milenković *Sistemas Operativos. Conceptos y Diseño.*

[OSC] Silberschatz Galvin, *Operating System Concepts.*

[PSIC] Miltos D. Grammatikakis, D. Frank Hsu & Miro Kraetzl, *Parallel System Interconnections and Communications.*

[DS] George Coulouris, Jean Dollimore, Tim Kindberg, *Distributed Systems, Concepts and Design.*

[SO] Peterson Silberschatz, *Sistemas operativos.*

[PE] M. A. Rodríguez i Roselló, *Programación Ensamblador para 8088-8086/8087.*

[CN] larry L. Peterson & Bruce S. Davie, .*Computer networks. A Systems Aproach.*

[CU] Jean-Marie Rifflet, *Comunicaciones en UNIX.*

[K24] Fernando Sánchez, Rocío Arango, *El Kernel 2.4 de Linux.*

[LI] John Lombardo, *Linux incrustado.*

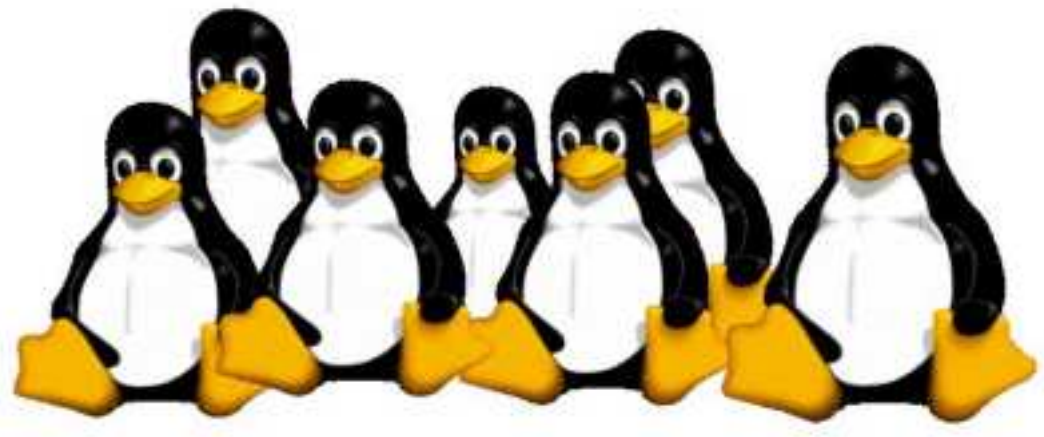
[ARPCL] Jason Fink & Matther Sherer, *Ajustes dek rebdunuebti y Planificación de ka capacidad con Linux.*

[TL] Todo Linux Magazine, *vol. 21-28*

[LM] The Linux Magazine, *vol.1-2.*

*openMosix y el mundo de la programación libre avanzan a pasos agigantados. Es un mundo donde el sol nunca se pone y donde nadie entiende de fronteras, razas ni religiones. Lo que cuenta es el código. Y llega en gran cantidad, y de gran calidad.*

- Moshe Bar -



Puedes encontrar este manual y sus futuras revisiones en la dirección `http://como.akamc2.net`

# ERRANDO DISCITUR