UNIVERSIDAD DON BOSCO

FACULTAD DE INGENIERÍA

ESCUELA DE ELECTRÓNICA

DISEÑO DE UN CLUSTER DE ALTO RENDIMIENTO PARA LA UNIVERSIDAD DON BOSCO

TRABAJO DE GRADUACIÓN

PRESENTADO POR

AGUILAR GARCIA MARCO TULIO

RAFAEL ERNESTO ALFARO SANDOVAL

PARA OPTAR AL GRADO DE

Ingeniero en Electrónica

Asesor:

ING. JULIO ADALBERTO RIVERA PINEDA

Marzo de 2007

San Salvador – El Salvador – Centro América

UNIVERSIDAD DON BOSCO
FACULTAD DE INGENIERÍA
ESCUELA DE ELECTRÓNICA

AUTORIDADES:

RECTOR
ING. FEDERICO MIGUEL HUGUET RIVERA

VICERRECTOR ACADÉMICO
PBRO. VÍCTOR MANUEL BERMÚDEZ YANEZ, sdb

SECRETARIO GENERAL
LIC. MARIO RAFAEL OLMOS

DECANO DE LA FACULTAD DE INGENIERÍA
ING. ERNESTO GODOFREDO GIRÓN

DIRECTOR DE ESCUELA DE ELECTRÓNICA
ING. OSCAR GIOVANNI DURÁN VIZCARRA

ASESOR DEL TRABAJO DE GRADUACIÓN
ING. JULIO ADALBERTO RIVERA PINEDA

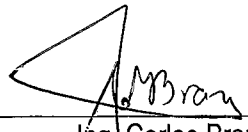
JURADO EVALUADOR
ING. JUAN CARLOS CRUZ DADA
ING. ANGEL SORIANO
ING. CARLOS BRAN

UNIVERSIDAD DON BOSCO
FACULTAD DE INGENIERÍA
ESCUELA DE ELECTRÓNICA

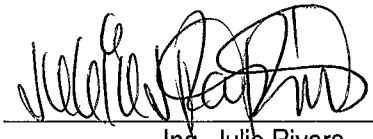
JURADO EVALUADOR DEL TRABAJO DE GRADUACIÓN

Ing. Juan Carlos Cruz Dada
JURADO

Ing. Angel Soriano
JURADO



Ing. Carlos Bran
JURADO



Ing. Julio Rivera
ASESOR

INDICE DE CONTENIDOS

CAPITULO 1 CLUSTERS.

1 NOCIONES GENERALES

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 interconectadas y comunicadas trabajando por un servicio conjunto.

1.1.2 Características esenciales de un cluster

En este punto explicaremos los requisitos que se deben cumplir para que un conjunto de computadoras interconectadas puedan ser consideradas un cluster.
Para crear un cluster se necesitan al menos dos nodos. Debe de existir un medio de comunicación (network) donde los procesos puedan migrar para procesarse en diferentes estaciones paralelamente. Un único nodo no cumple este requerimiento.
Las arquitecturas con varios procesadores no son consideradas clusters, bien sean máquinas SMP o mainframes, debido a que el bus de comunicación no suele ser de red, sino interno.
Por tanto esta es una característica de un cluster:

Un cluster consta de 2 o más nodos.

Como los nodos necesitan estar interconectados entonces podemos decir:

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

Los cluster se pueden caracterizar por el middleware utilizado, pues es este el que finalmente dotará al conjunto de máquinas la capacidad para migrar procesos, balancear la carga en cada nodo, etc.

Los clusters necesitan middleware especializado.

Las características del cluster son completamente dependientes del middleware, trataremos el modelo general de middleware que compone un cluster.
Para empezar, parte de este middleware se debe dedicar a la comunicación entre los nodos. Existen varios tipos de middleware que pueden conformar un cluster:

- **middleware a nivel de aplicación.**
  Este tipo de middleware se sitúa a nivel de aplicación, 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 la manera más abstracta posible. Este tipo de software suele generar elementos de proceso como rutinas, procesos o tareas, que se ejecutan en cada nodo del cluster y se comunican entre sí a través de la red.
- **middleware a nivel de sistema.**
  Este tipo de middleware se sitúa a nivel de sistema operativo, suele implementarse como parte del sistema operativo de cada nodo.
  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, es por eso que decidimos utilizar middleware a nivel de sistema operativo para el presente trabajo de graduación.

Cluster a Nivel de Sistema.

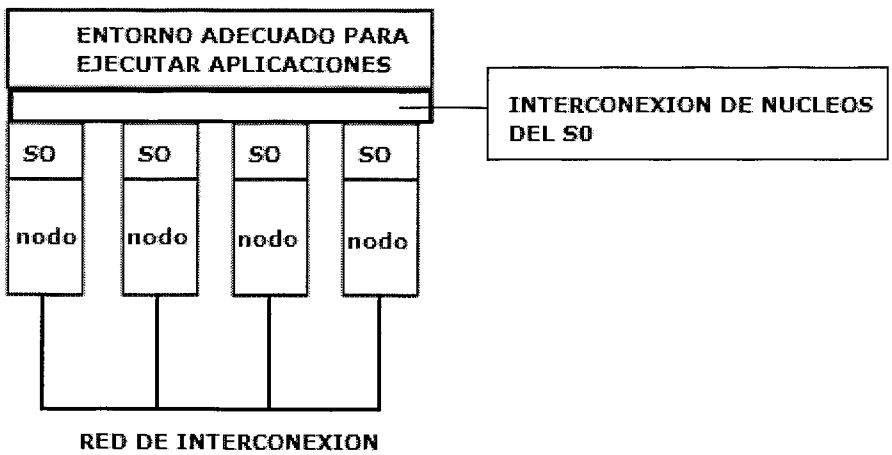


Figura 1.1, Cluster a nivel de sistema.
Cluster a nivel de aplicación.

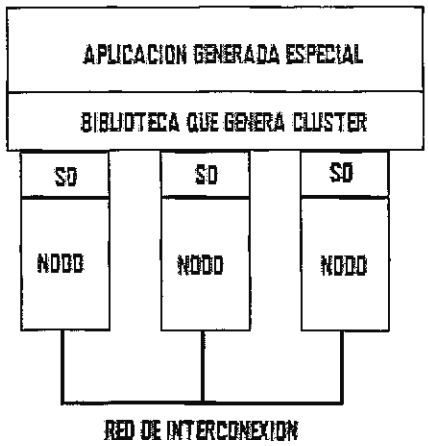


Figura 1.2, Cluster a nivel de aplicación.

1.1.3 Acoplamiento del cluster

Dependiendo del tipo de middleware utilizado el cluster presentara cierto grado de acoplamiento.

Se entiende por acoplamiento del middleware a la integración que tengan todos los elementos de middleware 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 cuan crítico es el middleware y su capacidad de recuperación ante fallos.

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

Existen 3 tipos de acoplamiento de middleware:

- middleware fuertemente acoplado.
- Middleware medianamente acoplado.
- Middleware débilmente acoplado.

Middleware fuertemente acoplado.

El middleware que entra en este grupo posee elementos que 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.

Este caso es el que se considera como más acoplado, de hecho no está catalogado como cluster, sino como sistema operativo distribuido. como Plan 9 la versión Unix II de los laboratorios Bell, desarrollada por los mismos creadores del lenguaje C y el Primer Unix.

Otro ejemplo son los cluster SSI (de imagen única), 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, para el sistema distribuido centralizado y un mapeo de los recursos físicos a este sistema de nombres.

Middleware medianamente acoplado.
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 ejemplo tenemos a openMosix y Linux-HA, (OpenMosix será la tecnología a utilizar para el presente trabajo de graduación).
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, en cual se encuentra la parte del fichero que falta en el nodo local, y por otro este tipo de acoplamiento no necesita un SSI para realizar otras tareas.

Middleware débilmente acoplado.
Generalmente se basan en aplicaciones construidas por bibliotecas preparadas para aplicaciones distribuidas. Es el caso de por ejemplo PVM y MPI. Éstos por sí mismos no funcionan en modo alguno con las características que antes se han descrito, y hay que dotarles de una estructura superior que utilice las capacidades del cluster para que éste funcione.
Ver referencia 1.

Resumen de características generales de un Cluster.
Las características básicas de un cluster de carácter general podrían resumirse en el siguiente esquema:

1. Un cluster consta de dos o más nodos.
2. Los nodos deben estar interconectados por un canal de comunicación funcional.
3. Es necesario middleware especializado.
   . El middleware puede ser:
   1. aplicación
   2. sistema
4. Se define acoplamiento de un cluster como nivel de colaboración del middleware que une los elementos del cluster. De este modo se clasifican en:

1. middleware fuertemente acoplado.
2. Middleware medianamente acoplado.
3. Middleware débilmente acoplado.
4.

5. Los nodos con integran el cluster trabajan para cumplir una funcionalidad conjunta. Funcionalidad por la cual se caracteriza el sistema completo.

1.1.4 Control del Cluster

Este parámetro de control hace referencia al modelo de administración que propone el cluster. Este modelo de administración 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 central (master) desde el cual se puede configurar el comportamiento de todo el sistema. Esto hace más vulnerable al sistema respecto a la tolerancia a fallos, pero proporciona una ventaja debido a que se utiliza administración sencilla y minimista del cluster, de tal forma que se aprovechan al máximo los recursos, cono este tipo de gestión se hace posible utilizar estaciones de trabajo sin disco.
- **Control descentralizado:** Es un modelo mas distribuido, como los Grids de los cuales ya han sido tratados en el presente documento, cada nodo debe administrarse y gestionarse individualmente. Se pueden administrar mediante aplicaciones de más alto nivel de manera centralizada, pero la mayoría de la administración que hace el nodo local es obtenida a través de los archivos de configuración de su propio nodo. Una ventaja es que presenta más tolerancia a fallos, pero requiere de una administración que demanda mayor cantidad de tiempo.

1.1.5 Homogeneidad de un cluster

La homogeneidad del cluster hace referencia a la homogeneidad de los equipos y recursos que conforman a éste. Los clusters heterogéneos 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.

- **Clusters 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.
- **Clusters heterogéneos:** formados por nodos con distinciones que pueden estar en diversos tópicos: Tiempos de acceso distintos, Arquitecturas diferente, Sistema Operativo diferente, Rendimiento de los procesadores o recursos sobre una misma arquitectura distintos.
  El uso de arquitecturas distintas, o distintos sistemas operativos, trae como consecuencia la existencia de una biblioteca que sirva de interfaz. Por lo tanto este tipo de clusters serán implementados a nivel de aplicación y por lo tanto no podrán ser tan acoplados como los clusters homogéneos.

La imposibilidad de construir 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.

1.1.6 Clasificaciones generales de los clusters segun el servicio.

A continuación presentaremos una clasificación de los clusters según el tipo de servicio que estos brinden.

Alto rendimiento.

Los clusters de alto rendimiento han sido creados para compartir, 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, ya sea que posean middleware a nivel de sistema o a nivel de aplicación. A nivel de sistema operativo tenemos a openMosix, mientras que a nivel de aplicación se encuentran otros como MPI, PVM, Beowulf. En cualquier caso, estos clusters hacen uso de la capacidad de procesamiento que pueden tener varias máquinas.

Alta disponibilidad.

Los clusters de **alta disponibilidad** cumplen una función muy distinta a la de un cluster de alto rendimiento. Estos La principal funcionalidad es estar controlando y actuando para que un servicio o varios se encuentren activos durante el máximo periodo de tiempo posible.

Pueden existir otras catalogaciones en lo que se refiere a tipos de clusters, en nuestro caso, solamente hemos considerado las dos que más clusters implementados engloban.

1.1.7 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.

1.1.7.1 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.

1.1.7.2 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.

1.1.7.3 Técnicas que utilizan

Las técnicas utilizadas dependen de a qué nivel trabaje el cluster.

Los clusters implementados a nivel de aplicación 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 compartimento 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.

1.1.8 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 así como algún factor de implementación.

1.1.8.1 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.

1.1.8.2 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.

1.1.8.3 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 capítulo perteneciente a clusters HA.

1.2 CLUSTERS HA

1.2.1 Introducción

Los clusters HA están 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 24 horas, 7 días a la semana, 365 días al año. Estos clusteres son una alternativa real a otros sistemas usados tradicionalmente para estas tareas de hardware redundante que son mucho más caros.

1.2.2 El interés comercial

Imagínese 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. Existen distribuciones de linux comerciales que traen software especifico para brindar alta disponibilidad, tal como heartbeat, en Suse y Redhat.

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 están 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).

1.2.3 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 (el enfoque principal de este trabajo es un cluster de alto rendimiento). Basta 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 dolares 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.

1.2.3.1 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: pudiéndolo ralentizar, inutilizar o para algunos servicios.

1.2.3.2 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 recuperar lo 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.

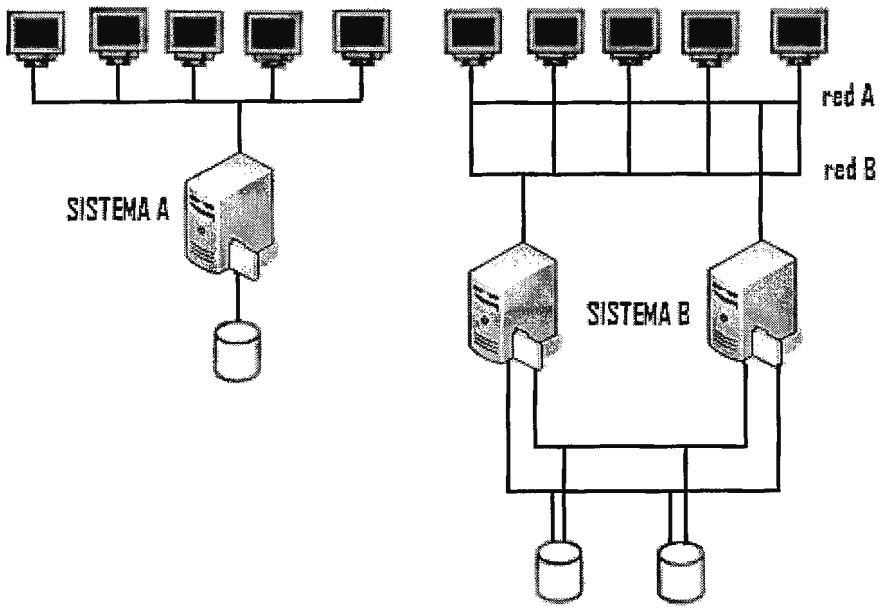


Figura 3.3: Clusters HA. Redundancia

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:

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`.

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:

- 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).
- 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.

Ver referencia 2.

1.3 CLUSTERS HP

1.3.1 Clusters HP: alto rendimiento

En este apartado realizaremos distinciones entre los clusters 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.
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 del tiempo de respuesta.
No existe ningún tipo de problema en concreto. Cualquier cluster que haga que el rendimiento del sistema aumente respecto al de uno de los nodos individuales puede ser considerado cluster de alto rendimiento.

Un Cluster de alto rendimiento puede brindar menores tiempos de proceso a problemas tales como:

- Cálculos matemáticos
- Renderizaciones de gráficos
- Compilación de programas
- Compresión de datos
- Descifrado de códigos
- Rendimiento del sistema operativo,

Las técnicas utilizadas dependen del nivel al que trabaje el cluster.
Los clusters implementados a nivel de aplicación 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 compartir recursos, balanceo de la carga de manera dinámica, funciones de planificación especiales y otros tantos factores que componen el sistema. Se intentan acercar a sistemas de imagen única (SSI).
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, esto sino se utiliza el parche experimental de migración de memoria del grupo MAASK de la India.

1.3.1.1 Migración y balanceo de carga en un cluster de alto rendimiento

Los clusters de alto rendimiento están dedicados a dar el mayor rendimiento computacional posible y existen multitud de formas de implementarlos.
En este documento se estudian unas cuantas de estas implementaciones (PVM, MPI, Beowulf) y se ahondará en una de ellas openMosix debido a que será el middleware a utilizar para el cluster que se implementara.

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 de alto rendimiento 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 (La universidad Don Bosco por ejemplo).
La segunda opción es que el middleware que se encarga del alto rendimiento 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 alto rendimiento es migrando procesos, dividiendo las aplicaciones grandes en procesos y ejecutando cada proceso en un nodo distinto.
Muchos de estos sistemas usan fundamentos estadísticos para la migración y balanceo de carga de los procesos, como por ejemplo openMosix.

1.3.2 PVM y MPI

(Ver referencia 3 y 4)

PVM y MPI utilizan el paso de mensajes. Los mensajes son pasados entre los procesos para conseguir que se ejecuten de manera colaborativa y de forma sincronizada.

1.3.2.1 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.
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 aplicación. 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.
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.

1.3.2.2 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 del 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.

1.3.3 Beowulf

El proyecto Beowulf fue iniciado por Donald Becker en 1994 para la NASA conocido hacker del kernel Linux. 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.

1.3.4 openMosix

OpenMosix es un software para conseguir clustering en GNU/Linux, migrando los procesos de forma dinámica. Consiste en unos algoritmos de compartimiento 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. (Ver referencia 5)

CAPITULO 2 (OPENMOSIX)

2.1 OPENMOSIX

2.1.1 Una muy breve introducción al clustering con openmosix.

La mayor parte del tiempo su computadora permanece ociosa. Si el usuario ejecuta un programa de monitorización del sistema como xload o top, usted verá probablemente que la lectura de la carga de su procesador permanece generalmente por debajo del 10%.

Si tiene 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 necesite potencia computacional (como por ejemplo para comprimir un fichero, o para una gran compilación) no podrá 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 brindarle el conjunto de computadoras de que pueda disponer para poder aprovechar aquellos que permanecen sin usar, básicamente 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 se presentará al lector gradualmente.

2.1.1.1 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.

2.1.1.2 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.

2.1.1.3 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, básicamente cada usuario con derechos suficientes puede ejecutar PVM.

2.1.2 Una aproximación histórica

2.1.2.1 Desarrollo histórico

Inicialmente Mosix empezó siendo una aplicación para BSD/OS 3.0.

La plataforma GNU/Linux para el desarrollo de posteriores versiones fue elegida en la séptima generación, 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.

2.1.2.2 openMosix no es 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 decidió no participar ya que no quería poner Mosix bajo licencia GPL.

Como había una significativa base de usuarios clientes de la tecnología 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 (www.openmosix.org) con un árbol CVS y un par de listas de correo para los desarrolladores y para los usuarios.

2.1.2.3 openMosix en acción: un ejemplo

Los clusters openMosix pueden adoptar varias formas. Para demostrarlo intente imaginar que comparte el piso de estudiante con un tipo que estudia ciencias de la computación. Imagine también que tiene las computadoras conectadas en red para formar un cluster openMosix.

Asume también que se encuentra convirtiendo ficheros de música desde sus CDs de audio a Ogg Vorbis para su uso privado, cosa que resulta ser legal en su país.

Su compañero de habitación se encuentra trabajando en un proyecto de C++, pero en este justo momento esta ocupado en otras tareas, y evidentemente su computadora está a la espera de ser intervenida de nuevo.

Resulta que cuando inicia 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 su máquina y en la de su compañero y deciden que es mejor migrar las tareas de compresión ya que el otro nodo es más potente, a la vez que en ese momento permanece ociosa ya que no se encuentra frente a ella.

Así pues lo que normalmente en un pentium233 tardaría varios minutos se da 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 su compañero, de forma transparente a usted.

Minutos después se encuentra escribiendo y su compañero de habitación vuelve a su computadora. Éste reanuda sus pruebas de compilación utilizando `pmake`, una versión del `make` optimizada para arquitecturas paralelas. Se da cuenta que openMosix está migrando hacia su 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 su computadora es parte del *pool*: sus procesos serán ejecutados, al menos en parte, en otras computadoras, pudiendo atentar contra su privacidad de datos.

Evidentemente las tareas de los demás también podrán ser ejecutadas en la suya.

2.2 CARACTERÍSTICAS DE OPENMOSIX

2.2.1 Pros de openMosix

- No se requieren paquetes extra
- No son necesarias modificaciones en el código de las aplicaciones.

2.2.2 Contras de openMosix

- Es dependiente del kernel
- No migra todos los procesos siempre, tiene limitaciones de funcionamiento
- Problemas con memoria compartida (si no se usa el parche migshm)

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.

2.2.3 Subsistemas de openMosix

Actualmente podemos dividir los parches de openMosix dentro del kernel en dos grandes subsistemas, veámoslos.

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.

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.

2.2.4 El algoritmo de migración

De entre las propiedades compartidas entre Mosix y openMosix podemos destacar el mecanismo de migración, en el que puede migrarse cualquier 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*, 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 cluster 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
- 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 (a menos que se use el parche migshm)

Cumpliendo todas estas condiciones el proceso puede migrar y ejecutarse migrado. No obstante, como podemos sospechar, openMosix no adivina nada. openMosix no sabe a 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 asíncrono. 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 señal 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.

2.3 INSTALACIÓN DE UN CLUSTER OPENMOSIX

La construcción de un cluster openMosix se compone de varios pasos.

1. el primero es compilar e instalar un kernel con soporte openMosix y aplicarle el parche migshm si se quiere migrar procesos que trabajen con memoria compartida;
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.

2.3.1 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.

http://openmosix.sourceforge.net/

Una vez descargado el parche del kernel openMosix habrá que descomprimirlo:

```
gunzip openMosix-2.4.26-2.gz
```

Descargar el parche migshm

Migshm es un parche DSM para openmosix. DSM quiere decir Distrubución de memoria compartida. Este habilita la migración de procesos que usan memoria compartida en openmosix, como ejemplo apache. (Ver referencia 6)

Actualmente, una de las principales limitaciones de openMosix es que las aplicaciones que usan memoria compartida y multihilo no son migrables en el cluster. Migshm cubre esta necesidad.

Migshm habilita la migración de procesos usando memoria compartida SYSV a través de las llamadas al sistema shmget(), shmat(), y shmctl(). Hilos creados

utilizando clone() pueden ser migrados también por migshm.

http://opensource.codito.com/migshm/

Descomprimimos migshm:

```
gunzip migshm-2.4.26-2.patch.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 2.4.26-2 sólo puede ser aplicado sobre el kernel 2.4.26.

Una vez descargado el kernel de Linux lo descomprimimos:

```
tar -zxf linux-2.4.26.tar.gz
```

Movemos el directorio donde hemos descomprimido el kernel a *linux-openmosix*:

```
mv linux-2.4.26 linux-openmosix
```

y aplicamos el parche de openMosix:

```
patch -p0 openMosix-2.4.26-2
patch -p0 migshm-2.4.26-2.patch
```

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.*

Para obtener mayor información relacionada al kernel Linux ver referencia: 7

Opciones del kernel de openMosix (con migshm)

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.

Support clusters with a complex network topology: Las máquinas que pertenecen al cluster openMosix pueden pertenecer a la misma red (segmento), 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 envío 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*.

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*-.

Enable extension localtime:

Esta caracteristica permite ejecutar time(2) localmente, usando time() desde el nodo de ejecución, este necesita tener una buena sincronización en los nodos del cluster.

Shared memory migration support: (parche migshm)

Soporte para migración de tareas con memoria compartida

flush() support for consistency

Aplicaciones que no necesitan la sincronización y son satisfechos así por consistencia débil, pueden mantener esta política remotamente. El programador puede mover explícitamente los cambios a la región compartida a cualquier propietario remoto de la región compartida, en cualquier momento durante la ejecución de programa, llamando la llamada del sistema del "flush()". Así, aun si las aplicaciones no utilizan semáforos, ellos puede alcanzar consistencia usando la llamada del flush().

Kernel Debug Messages

Mensajes de depuración del kernel con migshm

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:

Proceso de compilación del kernel:

```
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.

Si se usa Grub en lugar de lilo, entonces se procede a editar el archivo /boot/grub/grub.conf o /boot/grub/menu.lst

y se agregan las siguientes lineas:

title GNU/Linux

kernel /boot/kernel-openmosix

donde kernel-openmosix es el nombre de la imagen del kernel creada anteriormente.

OpenMosix en las distribuciones GNU/Linux mas populares:

Debian GNU/Linux.

Debian GNU/Linux inició su andadura de la mano de Ian Murdock en 1993. Debian es un proyecto totalmente no-comercial; posiblemente el más puro de los ideales que iniciaron el movimiento del software libre. Cientos de desarrolladores voluntarios de alrededor del mundo contribuyen al proyecto, que es bien dirigido y estricto, asegurando la calidad de una distribución conocida como Debian.En cualquier momento del proceso de desarrollo existen tres ramas en el directorio principal: "estable", "en pruebas" e "inestable" (también conocida como "sid").

Cuando aparece una nueva versión de un paquete, se sitúa en la rama inestable para las primeras pruebas, si las pasa, el paquete se mueve a la rama de pruebas, donde se realiza un riguroso proceso de pruebas que dura muchos meses. Esta

rama solo es declarada estable tras una muy intensa fase de pruebas.

Como resultado de esto, la distribución es posiblemente la más estable y confiable, aunque no la más actualizada. Mientras que la rama estable es perfecta para servidores con funciones críticas, muchos usuarios prefieren usar las ramas de pruebas o inestable, más actualizadas, en sus ordenadores personales. Debian es también famosa por su reputación de ser dificil de instalar, a menos que el usuario tenga un profundo conocimiento del hardware de la computadora. Compensando este fallo está "apt-get" un instalador de paquetes Debian. Muchos usuarios de Debian hacen bromas sobre que su instalador es tan malo por que solo lo han de usar una vez, tan pronto como Debian está en funcionamiento, todas las actualizaciones, de cualquier tipo pueden realizarse mediante la herramienta apt-get.

Debian GNU/Linux posee otra ventaja adicional haciendo referencia a openmosix, en su ultima versión 3.1 (sarge) facilmente puede estar funcionando con un kernel Linux 2.4 o con un kernel Linux 2.6, lo cual brinda una gran versatilidad, una vez que openmosix 2.6 sea estable, la migración desde 2.4 es sencilla.

Debian trabaja con runlevels al estilo System-V, y posee un runlevel especifico solo para la inicialización del sistema, cada tarea que se encuentre en dicho runlevel posee una prioridad, de tal forma que el usuario puede decidir que servicios se ejecutaran primero, servicios tales como udev, devfs, etc... o bien el usuario los puede quitar facilmente.

El sistema udev solo funciona con kernels 2.6, por lo tanto es necesario eliminarlo del runlevel de iniciación, para asi evitar conflictos con openmosix 2.4, que no utiliza el sistema udev para los dispositivos.

Por lo tanto podemos decir que Debian GNU/Linux es un candidato excelente para ejectutar openMosix.

Gentoo 2005.1

Gentoo Linux fué creada por Daniel Robbins, un conocido desarrollador de Stampede Linux y FreeBSD. Fue el contacto del autor con FreeBSD y su función de autobuild llamada "ports" lo que le inspiró a incorporar los "ports" en Gentoo bajo el nombre de "portage". Un informe detallado de los principios de Gentoo se puede encontrar en esta trilogía llamada Creando la distribución. La primera versión estable de Gentoo fue anunciada en Marzo del 2002.

Gentoo Linux es una distribución basada en código fuente. Mientras que los sistemas de instalación proveen de varios niveles de paquetes pre-compilados, para obtener un sistema Linux básico funcionando, el objetivo de Gentoo es compilar todos los paquetes de código en la máquina del usuario. La principal ventaja de esto es que todo el software se encuentra altamente optimizado para la arquitectura de tu computadora.

También, actualizar el software instalado a una nueva versión es tan fácil como teclear un comando, y los paquetes, mantenidos en un repositorio central, se

mantienen bastante actualizados.

Gentoo es un universo de opciones, y configuraciones, aunque requiere un poco mas de conocimiento por parte del administrador, comparándola con las distribuciones mas conocidas.

Gentoo posee un esquema de runlevels distinto al de Debian, Fedora, Suse. Gentoo al igual que Debian permite la potencia al usuario de decidir que servicios se ejecutan en el arranque del sistema, al momento del boot, de tal forma que Gentoo 2005.1 con kernel 2.6 por defecto, facilmente es configurable a un kernel 2.4 sin afectar la integridad del sistema, es decir es fácil evitar que udev se ejecute al inicio y otros servicios propios de la rama 2.6 del kernel Linux.

Por lo tanto concluimos que Gentoo es una buena distribución para la ejecución de OpenMosix.

Fedora Core 4

A diferencia de Debian y Gentoo, con runlevels altalmente configurables, que brindan al usuario el control sobre ellos, Fedora es muy diferente. Pues no utiliza un runlevel para los scripts de iniciación, a diferencia esta distribución utiliza un complejo script bash, /etc/sysinit, no diseñado para ser configurado por el usuario, por lo tanto es muy complejo deshacerse del udev.

Fedora ademas posee gcc 4.0 en Fedora 4 , el cual no puede compilar kernels 2.4, aunque se pudiesen compilar en otra maquina con un gcc 3.5 que si compila kernels 2.4. El problema radica en que los kernels 2.6 utilizan otro sistema para la carga de modulos del kernel y quitarlos acarrea un grave problema de dependencias. Ocurre otro grave problema de dependencias si se desinstala gcc 4.0.

Por lo tanto Fedora 4 no es una distribución adecuada para trabajar con openmosix 2.4.

Suse 9.3

Suse 9.3 al igual que Fedora vienen siendo una distribución bastante elaborada, y poco versátil, Suse 9.3 no trae soporte por defecto, para compilar kernels de la rama 2.4 de Linux, existen símbolos (instrucciones) no soportados, ya que Suse 9.3 solo trae soporte para un kernel 2.6 o mayor. Colocar un compilador que reconozca los símbolos de la rama 2.4 acarrea muchos problemas de dependencias.

Por lo tanto Suse 9.3 no es una distribución adecuada para trabajar con openMosix 2.4.

2.3.2 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.

Por ejemplo si descargamos la versión 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 monitoramientodel 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 algún 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-.

2.3.3 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.

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. 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 conexionar 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 *à 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.

2.3.4 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, así como 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.

`migración`: 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.

2.3.5 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 aquí queda un poco difícil de entender, así que 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.

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.

CAPITULO 3

3.1 OpenMosix internamente

3.1.1 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 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.

3.1.2 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 están basadas en información proveniente 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 logeado. 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. 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. Al mismo tiempo, mantiene una pequeña ventana con la información llegada más reciente. Este esquema soporta escalabilidad, difusión de información uniforme y configuraciones dinámicas.

3.1.3 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.

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 interface 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.

3.1.4 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 porcentaje de memoria libre o la carga de procesos y no la escasez de recursos.

Memory Ushering inspira su nombre en una política 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.

3.1.5 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 están 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.

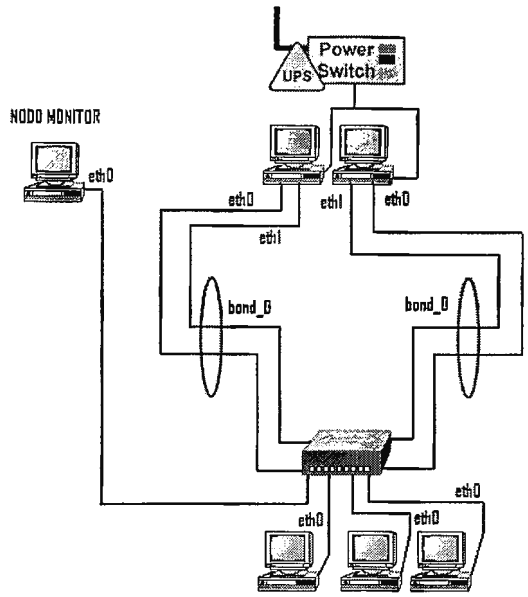
Un proceso que usa alguna de estas funciones es asignado indefinidamente a su UHN. Si el proceso ya había estado migrado, se retorna al UHN.

CAPITULO 4

4.1 IMPLEMENTACION DEL CLUSTER OPENMOSIX DE LA UNIVERSIDAD DON BOSCO.

La implementación del cluster openmosix de la Universisdad Don Bosco, se realizó utilizando el siguiente esquema:

- ✔ Un servidor monitor.
- ✔ Un servidor central con kernel OpenMosix
- ✔ Nodos de balanceo de carga con un kernel Openmosix cargado en ram a traves de un Live CD.
- ✔ Alta disponibilidad de las interfaces Ethernet.



La fabricación del Live CD se debe a la necesidad de escalabilidad del Cluster, a parte de la versatilidad que le brinda al centro de computo actual de la universidad don Bosco, ya que cualquier maquina basada en x86 puede ser un miembro potencial del cluster de la Universidad Don Bosco, con solo seleccionar en la BIOS del nodo, el arranque a traves del Live CD.
El Monitor permite verificar el estado de los nodos, así como el servicio

openmosix.
El nodo central es donde el usuario realizara las operaciones de computo intensivo.
La alta disponibilidad se provee a través de bonding.

4.1.1 DRISTIBUCION GNU/Linux de la Universidad Don Bosco, "UDB Cluster GNU/Linux".

CONSTRUYENDO EL LIVE CD OPENMOSIX.

Esta distribución GNU/Linux ha sido un factor clave en la implementación del Cluster de la Universidad Don Bosco, sin embargo, en este documento no explicaremos el largo proceso de diseño y construcción de un Live CD, que involucran muchos conceptos, que perfectamente podrían formar otra tesis a parte, por lo que nos limitaremos a brindar conceptos basicos en este documento, el lector podrá ver referencia 8, donde podra encontrar amplia información relacionada al diseño y contrucción de un Live CD.

En primer lugar, definiremos qué es un Live-CD. Pues no es más que un sistema operativo funcional que se puede ejecutar directamente desde un cd, en lugar de tener que hacerlo desde el disco duro.

Hace ya mucho tiempo que existen estos sistemas, e incluso existen sistemas tan pequeños que se ejecutan desde un disquete.

En realidad, dichos sistemas, no funcionan directamente desde el cd-rom, o la disquetera, esto sería demasiado lento. En vez de eso, lo que hacen es crear un sistema de ficheros en la memoria ram y después copian en él, una imagen de un sistema instalado.

De esta manera utiliza la memoria ram, como si se tratase del disco duro. Ésto tiene sus ventajas, no se necesita disco duro, ni instalar nada, no se tiene que perder tiempo en instalaciones, se puede utilizar cualquier equipo que esté soportado por el sistema, sin importar que tiene instalado, y ademas se puede probar, sin riesgo a estropear el software del equipo.

Es por estas cualidades por la que un Live CD es ideal para el cluster implementado en la Universidad Don Bosco, debido a que las computadoras existentes poseen software instalado, sistemas operativos privativos en disco duro que sirven al estudiante durante sus practicas, por lo que no sería probable conseguir 5 computadoras, las cuales tuviesen una distribución Linux instalada.

El Live CD nos permitirá utilizar un ramdisk, para ejecutar Linux sin emuladores, y formar parte del nodo del cluster solo en el momento que este host a cargado el sistema operativo UDB CLUSTER GNU/LINUX a través del CD ROM.

4.1.2 NODO MONITOR:

El nodo monitor posee dos aplicaciones que realizan funciones especificas.

Estas aplicaciones son Nagios y Cacti.

El Nagios permitirá al cluster la notificación de el estado de los nodos vía SMS. Nos permitirá conocer cuando el proceso OpenMosix de alguno de los nodos esté corrupto, o incluso cuando la pila TCP/IP de alguno de estos ya no responda. En cualquier caso un programa diseñado especialmente para este caso permitirá la notificación por SMS, al insertar este programa dentro de un plugin de Nagios.

El cacti permitirá monitorear el trafico de las interfaces de red de los nodos integrantes del cluster, tarea que se realizara a través de SNMP.

A continuación ampliaremos un poco mas la visión a cerca de Nagios y Cacti:

4.1.2.1 NAGIOS

Nagios es una aplicación para el monitoreo de sistemas y redes. Verifica equipos y servicios que nosotros le especificamos, alertándonos cuando las cosas van mal y cuando se normalizan. (Ver referencia 9)

Nagios fue originalmente diseñado para correr en Linux, pero también debe trabajar en la mayoría de los unix.

Algunas de las varias características de Nagios son:

Monitoreo de servicios de red (SMTP,POP3,HTTP,NNTP,PING,etc.)

Monitoreo de recursos de equipos (carga en el procesador, uso de disco duro, etc)

Diseño simple de plugins que permite a los usuarios un desarrollo fácil de sus propias verificaciones de servicios.

Revisión de servicios en paralelo.

La habilidad de definir una jerarquía de los equipos de la red utilizando equipos "padre" (parent), permitiendo la detección y distinción de los equipos que están abajo y aquellos que son inalcanzables.

Notificaciones a contactos cuando un servicio o equipo presenta problemas y necesita resolverse (vía email, page o método definido por el usuario).

Habilidad de definir manejadores de eventos (event handlers) para ser ejecutados cuando un servicio o equipo presente eventos para que pueda ser proactiva la resolución del problema.

Rotación de los archivos de log automática.

Soporte para implementar equipos redundantes para monitoreo.

Interface WEB opcional para ver el estado actual de la red, notificaciones, historial de problemas, archivo log, etc.

Requerimientos de sistema

El único requerimiento para ejecutar Nagios es una computadora con Linux (o una variante de Unix) y compilador C. Probablemente necesitemos tener configurado TCP/IP, por que la mayoría de las revisiones de los servicios se realizara por medio de la red.

No requerimos utilizar los CGIs incluidos con Nagios. Sin embargo, si decidimos utilizarlos, necesitaremos que el siguiente software este instalado...

1. Un servidor WEB (preferentemente Apache)
2. Thomas Boutell's biblioteca gd versión 1.6.3 o mayor (requerido para los CGIs de statusmap y trends)

El plugin utilizado es check_tcp, el cual nos permite monitorear el puerto a través del cual openmosix realiza la migración de procesos.

Para tener una información mas amplia de las características del nagios, referirse a www.nagios.org.

Los archivos de configuración de nagios se pueden editar de una forma mas amigable desde una interface web, un proyecto separado de nagios, llamado fruity: http://fruity.sourceforge.net/

La notificación por mensaje corto se realiza a través de un script desarrollado para enviar mensajes vía web, este fue desarrollado en lenguaje perl, a través de las librerías LWP.

Para tener una visión mas amplia de perl y el modulo LWP referirse a www.cpan.org

```
#!/usr/bin/perl -sw
use strict;
use LWP;
use LWP::UserAgent;
use HTML::Form;
use HTTP::Cookies;
######## MAIN
if ($ARGV[0] eq "telecom")
{
my $ua = LWP::UserAgent->new(
                agent           => 'Mozila/4.0 (compatible; MSIE 6.0; windows NT 5.1)',
             #  cookie_jar      => HTTP::Cookies->new(
             #  file            => my $conf{f},
             #  autosave        => 1,
             #  ignore_discard  => 1,
       #)
```

```
    );

$ua->requests_redirectable;
my $req = HTTP::Request->new(GET => "http://www.telecom.com.sv/notiavanzados/centroMensajeria.asp");
$req->header('Accept' => 'text/html');
my $res = $ua->request($req);
my @forms = HTML::Form->parse($res);
$forms[2]->value( txtPhone => $ARGV[1] );
$forms[2]->value( areaMsg => $ARGV[2] . ': ' . $ARGV[3] );
$req = $forms[2]->make_request;
push @{ $ua->requests_redirectable }, 'POST';
$res = $ua->request($req);
if ($res->status_line == "200 OK"){
print "Success 1\n"; } else
{print "Failed\n";}
}



if ($ARGV[0] eq "telemovil") {
my $ua = LWP::UserAgent->new(
                agent           => 'Mozila/4.0 (compatible; MSIE 6.0; windows NT 5.1)',
              #  cookie_jar      => HTTP::Cookies->new(
              #  file            => $conf{f},
              #  autosave        => 1,
              #  ignore_discard  => 1,
        #)
    );
$ua->requests_redirectable;
my $req = HTTP::Request->new(GET => 'http://area.mensajito.com/interactivo555/client.php?orden=1&nick=' . $ARGV[2] .
'&foo=' . int(rand(30000)));
$req->header('Accept' => 'text/html');
my $res = $ua->request($req);
my $session= $res->content;
$req = HTTP::Request->new(GET => 'http://area.mensajito.com/interactivo555/client.php?orden=2&' . $session . '&nick='.
$ARGV[2] . '&dstphone=503' . $ARGV[1] . '&pin=1&foo=' . int(rand(30000)));
$req->header('Accept' => 'text/html');
$res = $ua->request($req);
$req = HTTP::Request->new(GET => 'http://area.mensajito.com/interactivo555/client.php?orden=3&' . $session . '&nick=' .
$ARGV[2] . '&mensaje=' . $ARGV[3] . '&foo=' . int(rand(30000)));
$req->header('Accept' => 'text/html');
$res = $ua->request($req);
if ($res->status_line eq "200 OK"){
print "Success\n"; } else
{print "Failed\n";}
#$req = HTTP::Request->new(GET => 'http://area.mensajito.com/interactivo555/client.php?orden=4&' .  $session . '&nick=' .
$ARGV[2] . '&dstphone=503' . $ARGV[1]  . '&pin=1&foo=' . int(rand(30000)));
#$req->header('Accept' => 'text/html');
#$res = $ua->request($req);
```

```
#if (!$res->is_success) {
#        print 'Error1 ' . $res->status_line . "\n";
#        print "\n";
#        exit 1;
#        }
$req = HTTP::Request->new(GET => 'http://area.mensajito.com/interactivo555/client.php?orden=6&nick=' . $ARGV[2] . '&foo='
. int(rand(3000)) . '&'. $session );
$req->header('Accept' => 'text/html');
$res = $ua->request($req);


}
```

4.1.2.2 CACTI.

El Cacti es un frontend completo para la RRDTool, almacena toda la información necesaria para crear gráficos y para poblarlos con datos en una base de datos de MySQL. El frontend es totalmente implementado en PHP. Para poder con mantener gráficos, fuentes de datos, y archivos Round Robin en una base de datos, el cacti maneja la recolección de los datos. Posee soporte SNMP incluso.

(Ver referencia 10)

Fuentes de datos.

Para manejar la obtención de los datos, el cacti puede hacerlo a través de un script, por medio de un cron-job y llena de datos los archivos round robin y la base de datos Mysql.

Las fuentes de datos también pueden ser creadas, lo cual corresponde a los datos actuales en la gráfica. Por ejemplo, si un usuario quiere graficar los tiempos de ping a un host, se debe crear una fuente de datos utilizando un script que hace ping a un host y retorna su valor en mili segundos, después de definir opciones para la RRDTool tales como la forma de almacenar los datos, usted estará en capacidad de definir cualquier información adicional que la fuente de datos de entrada requiera. Tal como el host a hacer ping en este caso. Una vez que las fuentes de datos están creadas, estas son mantenidas en intervalos de 5 minutos.

Gráficas.

Ya que una o mas fuentes de datos están definidas, una gráfica de la RRDTool puede ser creada usando los datos. El cacti te permite crear al menos cualquier gráfica RRDTool, usando todos los tipos de gráficas y funciones de consolidación standard de la RRDTool. Una función de selección de color y modificación de texto permite una creación mas fácil de las gráficas.

Existen muchas formas de desplegar las gráficas, existe un árbol de gráficas que permite colocar las gráficas en árbol jerárquico, para propósitos de organización.

Administración del usuario.

Ademas de las muchas funciones del cacti, una herramienta de administración basada en el usuario esta incluida, de tal manera que puedes agregar usuarios y brindarle los derechos a ciertas áreas del cacti. Esto permitirá a alguien crear algunos usuarios que puedan cambiar parámetros de las gráficas, mientras que otros solo podrán verlas. Cada usuario mantiene sus propias configuraciones cuando este observa las gráficas.

Plantillas.

Últimamente, el cacti esta habilitado para escalar a un gran numero de fuentes de datos y gráficas a través del uso de plantillas. Esto permite la creación de una sola plantilla de gráfica o fuente de datos que define cualquier gráfica o fuente de datos asociada con el. Las plantillas de host permiten definir las capacidades de un host, de tal manera que el cacti puede obtener la información a través de agregar un nuevo host.

Para obtener una visión mas amplia del cacti referirse a www.cacti.net

4. BONDING :

Linux bonding provee un método para agregar múltiples interfaces de red dentro de una sola interfaz lógica llamada bond. El comportamiento de la bond depende de los modos de configuración, que generalmente proveen alta disponibilidad y balanceo de carga.

Para la instalación de bonding se necesita configurar el kernel y en la sección de soporte para dispositivos de red, seleccionar modulo de soporte para bonding. También se necesita instalar la aplicación "ifenslave" que se puede extraer de la misma fuente del kernel.

PARÁMETROS DEL MODULO:

mode:

Los posibles valores de los modos de configuración son:

0= round robin:

Transmite los paquetes en orden secuencial desde la primer interfaz de red esclavo disponible, hasta el ultimo.

1= active backup:

En este modo solamente una interfaz de red esclava en la bond esta activo.

Una interfaz de red esclava diferente se activa solamente si, la interfaz esclava

que estaba activa falla. La MAC addres de la interfaz bond es vista externamente como un solo puerto, de esta manera se logra que el switch no se confunda.

2= XOR balance:

La transmisión se basa en políticas hash. Por defecto la política es un simple modulo de conteo de las interfaces de red esclavas existentes (operación XOR entre la MAC address fuente y la MAC address destino).

3= broadcast: política de broadcast

Transmite todos los paquetes, por sobre todas las interfaces de red esclavas disponibles.

4= 802.3 ad: Agregados de enlaces dinámicos IEEE 802.3ad.

Se agregan grupos de interfaces de red que comparten la misma velocidad y ajuste de transmisión duplex. Utiliza todos las interfaces de red esclavas que han sido agregados al grupo, de acuerdo a la especificaciones 802.3ad.

La seleccion del esclavo para el trafico de salida, se hace de acuerdo a la política de transmisión has. Dicseleccionuconfiguración se puede cambiar en el modo XOR a través de la opción xmit_political_policy_transmision

5= balance tlb: Transmisión adaptativa para balanceo de carga.

En este modo el canal bonding no requiere algún soporte especial para el switch. El trafico de salida es distribuido de acuerdo al flujo de la carga en cada interfaz de red esclava (se realiza un calculo relativo a la velocidad) .

El trafico de paquetes de entrada se recibe por la interfaz de red que el modulo destina para esa función. Si la interfaz de red esclava que esta recibiendo paquetes falla, la otra esclava asume el control de la MAC address de la interfaz esclava que estaba recibiendo.

miimon:

Usa un valor entero de frecuencia (ms) para monitorear el enlace MII. Tiene un valor de cero por defecto y significa que el monitoreo del enlace estará deshabilitado.

Un buen valor de configuración es de 100 ms, si lo que se desea monitorear es el

estado de un enlace.

downdelay:

Usa un valor entero (ms) para retardar la deshabilitación del enlace después que se ha detectado una falla del enlace. El valor de este modo debe de ser múltiplo de miimon.

updelay:

Usa un valor entero (ms) para retardar la habilitación del enlace después que se ha detectado el restablecimiento del enlace. El valor de este modo debe de ser múltiplo de miimon.

arp_interval:

Usa un valor entero para la frecuencia (ms) de monitoreo de arp. Este modo no debe de configurarse en conjunto con miimon

arp_ip_target:

Especifica la dirección ip a usar como monitoreo de arp para cuando el intervalo es mayor que cero. Estos son los objetivos que la requisición de arp envía para determinar la salud de los enlaces que son el objetivo

ALTA DISPONIBILIDAD DEL CLUSTER:

La alta disponibilidad se alcanza utilizando el reporte del estado de la MII. El controlador del bonding puede regularmente estar chequeando los enlaces de todos los esclavos a través de los registros de estado de la MII. Los intervalos de chequeo se especifican en miimon el cual es un valor entero que representa el tiempo en ms. Este valor no debe de ser tan cerrado porque le puede restar interactividad del sistema.

CONFIGURACIÓN:

Esta configuración es muy problemática debido a que podría confundir sobre el hecho de que hay múltiples puertos, ademas, la MAC address del nodo debe ser vista en el puerto como una sola y evitar confundir al switch.

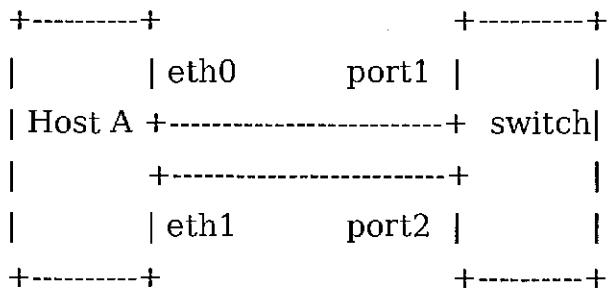
Si se necesita saber cual interfaz estará en modo activo y cual estará en modo de backup, habría que usar ifconfig. Todas las interfaces de backup tienen seteada la bandera de NOARP

En el host hay un constante monitoreo de la interfaz activa, al momento de detectar que ha fallado, le transfiere el control a la interfaz que esta en modo de backup.

En esta configuracion el host se ve fuertemente afectado por el tiempo de expiración de la tabla de re-envió (forwarding) del switch.

La configuración queda así:

Usando dos tarjetas de red para cada nodo y conectadas al switch para configurar failover en la NIC

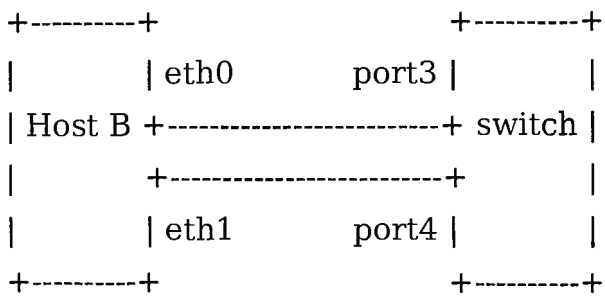


Para el nodo A como root se digita:

modprobe bonding miimon=100 mode=1

ifconfig bond0 10.0.19.1

ifenslave bond0 eth0 eth1



Para el nodo B como root

modprobe bonding miimon=100 mode=1

ifconfig bond0 10.0.19.2

ifenslave bond0 eth0 eth1

Para mayor información referente al modulo bonding referirse a www.kernel.org, existe información relevante dentro del codigo fuente del kernel linux.

LIMITACIONES:

Con el modulo bond solamente se puede monitorear el estado del enlace de las interfaces de red.

Si el switch en el otro lado falla por alguna razón, no se logra alta disponibilidad. Aunque exista esta falla, la alta disponibilidad en las interfaces de red se mantienen habilitadas.

Esta configuración puede ser útil para switches que envíen información de forma multicast en sus enlaces.

4.2 Prueba de desempeño del Cluster.

La prueba de desempeño se realizo a través de un programa que al ejecutarse demanda alta cantidad de procesamiento, el cual se ha diseñado tomando las necesidades de paralelización según las condiciones en las cuales OpenMosix migrara procesos.

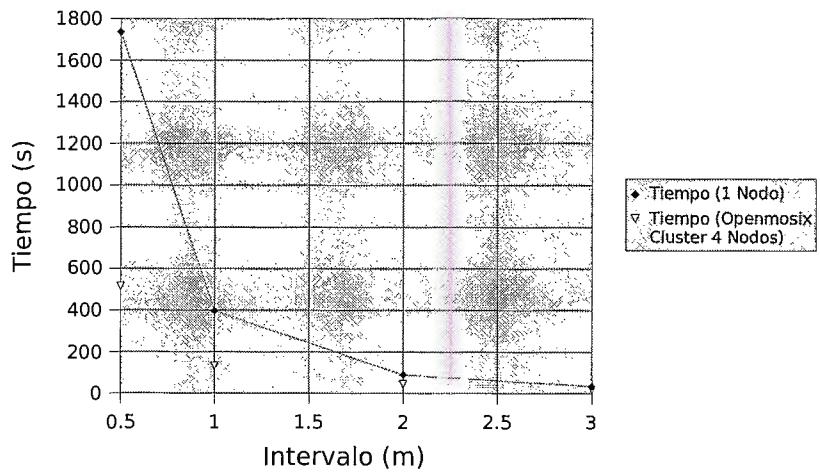
Este programa representa la resolución de la primera parte del del examen final de la carrera de técnico en mantenimiento aeronáutico para el año 2006, puede verse el parcial en el Anexo a este documento.

El primer problema es simplemente una tabla de valores para distintos valores de altura, con una variación en las funciones que generan dichos datos según los rangos de altura, el programa resuelve distintos valores de altura, de 0 a 47000 m variando la precisión, es decir, de metro en metro, de centímetro en centímetro, etc...

A continuación se presentan los datos obtenidos en las pruebas realizadas:

No	Intervalo (m)	Tiempo (s)(1 Nodo)	Tiempo (s)(Openmosix Cluster 4 Nodos)
1	3	36.020145	21.686840
2	2	91.137381	47.042610
3	1	395.696845	134.128386
4	0.5	1736.827143	514.713468

A Continuación presentamos una gráfica comparativa:



Como podemos apreciar, el cluster a funcionado exitosamente, a medida aumenta la carga de procesamiento el cluster se comporta de forma mas cercana a su comportamiento ideal. Cuando el intervalo de medida es menor (mayor precisión) mayor es la carga a procesar.

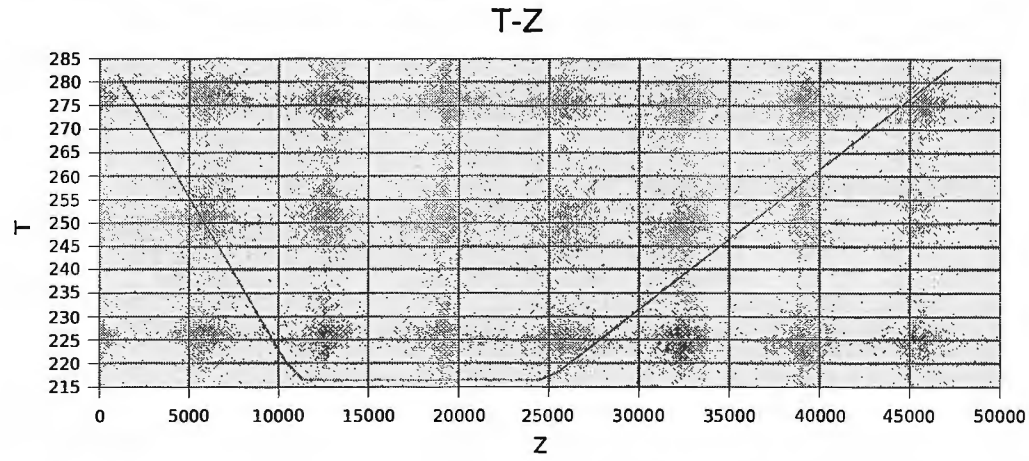
A continuación presentamos el programa utilizado en la prueba de desempeño del Cluster:

```
#!/usr/bin/octave -qf
time0 = clock ();
g = -9.8;
Mmolecularaire=28.9e-3;
Raire= 8.31/Mmolecularaire;
precision=1000;
lamda=-6.5e-3;

i=0;
T_z=0;

T0=288;
P0=101325;
ro0=1.225;

lamda_11000  = 0;
T_11000 = 216.5;
P_11000 = 22552;
ro_11000 = 0.3629;

lamda_25000 = 3e-3;
T_25000 = 216.5;
P_25000 = 2481;
ro_25000 = 0.0399;

global buffer="";
global output="out"

boundary=0;

altitud_max=47000;
global j=10;

wide = altitud_max/j;

global process=0;
for k=1:j
retfork=fork();
```

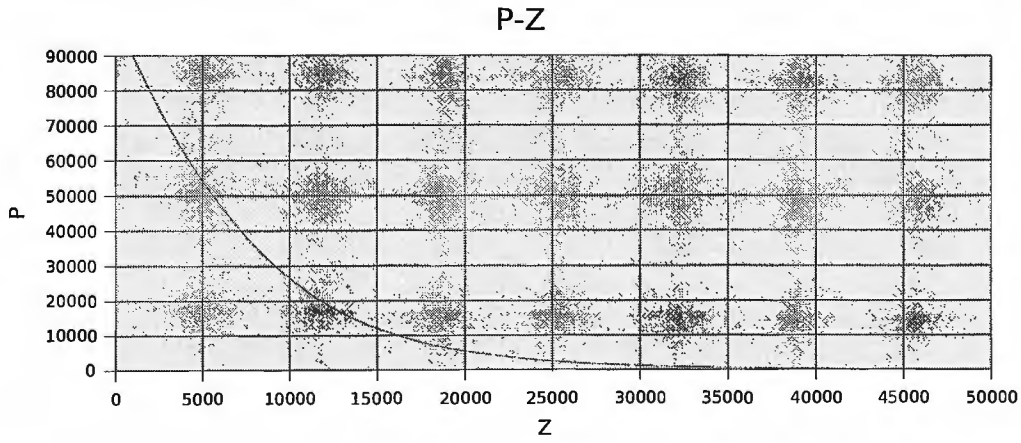
```
if (retfork == 0)
i=fix(wide*(k-1));
boundary=fix(wide*(k));
process=k;
break;
endif

endfor


if (retfork==0)

while (i <= boundary)

i=i+precision;

if (i<=11000)

T_z = T0 + lamda*i;
P_z = P0*((T_z)/T0)^(g/(Raire*lamda));
roz = ro0*(T_z/T0)^(g/(Raire*lamda) - 1);

endif

if (i>11000 && i<=25000)

T_z = T_11000;
P_z = P_11000*exp(g*(i-11000)/(Raire*T_z));
roz = ro_11000*exp(g*(i-11000)/(Raire*T_z));

endif

if (i>25000)

T_z = T_25000 + lamda_25000*(i-25000);
P_z = P_25000*(T_z/T_25000)^(g/(Raire*lamda_25000));
roz = ro_25000*(T_z/T_25000)^((g/(Raire*lamda_25000))-1);

endif


buffer = sprintf ("%s %d, %f, %f, %f \n", buffer, i, T_z,P_z,roz);
endwhile

#printf ("%s", buffer);
myfile = fopen(strcat("/tmp/",int2str(k),".clusterdat"),"wt");
fputs (myfile, buffer);
fclose (myfile);
```

```
exit(0);
endif

if (retfork>0)

for k=1:j
waitpid(-1);
endfor

system('cat /tmp/1.clusterdat /tmp/2.clusterdat /tmp/3.clusterdat /tmp/4.clusterdat /tmp/5.clusterdat /tmp/6.clusterdat
/tmp/7.clusterdat /tmp/8.clusterdat /tmp/9.clusterdat /tmp/10.clusterdat > output.csv');
system('rm /tmp/*.clusterdat');
elapsed_time = etime (clock (), time0);
printf("Time elapsed: %f sec\n", elapsed_time);
exit(0);
endif
```

A continuación se presentan las graficas obtenidas.

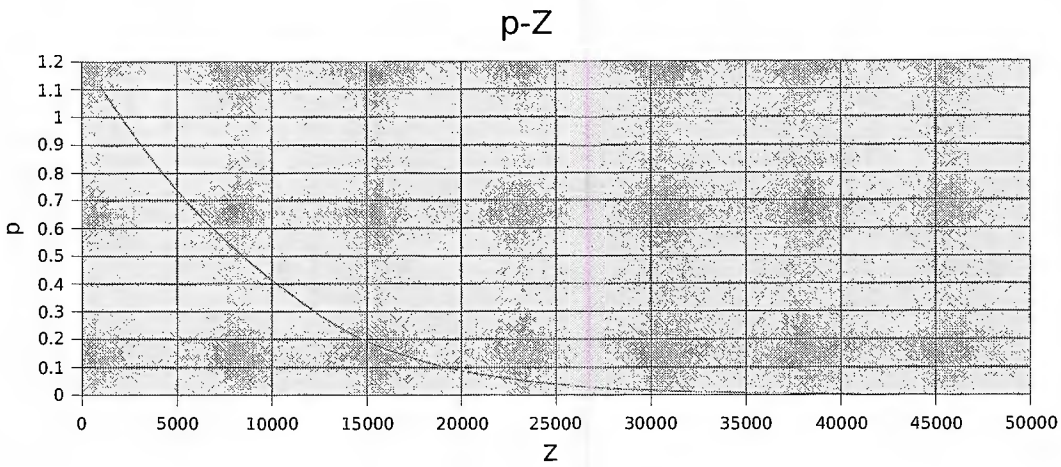


T en Kelvin. Z en metros.



P en Pascales.

Z en metros.



p en Kg/m^3.

Z en metros.

4.3 Activación del cluster:

Activar el nodo central

Iniciar openmosix en una consola (En el nodo central).

Ingresar al BIOS de cada nodo esclavo, y cambiar el medio de arranque, de tal forma que apunte al drive en el cual se encuentra el CDROM.

Insertar los Live Cds en los drives seleccionados en el paso anterior.

Activar openmosix en cada uno de los nodos que tienen cargado el live Cd en RAM.

El cluster esta listo para su uso.

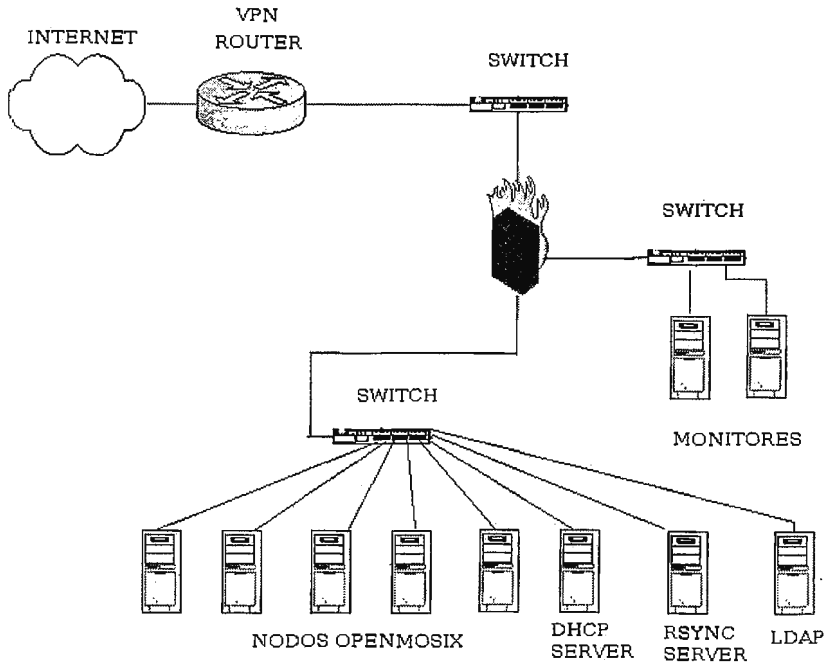
Activar bonding en las interfaces de red (opcional).

Activacion de los nodos monitores “/etc/init.d/nagios start” (opcional).

4.4 Propuesta Optimizada del Cluster.

Si bien el cluster elaborado en este proyecto es totalmente funcional, posee deficiencias obvias, como por ejemplo, nadie garantiza el ancho de banda óptimo para la transferencia de procesos, debido a que estamos en una LAN compartida con otras aplicaciones y otros hosts ajenos al cluster, nadie nos garantiza que el DHCP server estara siempre activo para brindarnos la configuración dinamica de los nodos, tampoco si las IPs seran realmente exclusivas, y muchas otras situaciones que se pueden afectar el rendimiento del cluster.

La propuesta ideal para solventar muchos de estos inconvenientes es:



Detallaremos cada uno de los componentes:

El Router nos permitira tener enlace hacia Internet ademas de brindarnos la capacidad de montar una VPN que nos permita comunicarnos hacia nodos esclavos ubicados en otro punto geografico, con el objetivo de permitir el crecimiento del cluster a nivel Geografico, nos solo a nivel de segmentos de red locales. Este Equipo puede ser perfectamente una maquina con GNU/Linux, con "quagga" para efectuar el enrutamiento y "openswan" o "freeswan" como herramienta de montaje de la VPN.

El Firewall permitirá restringir el acceso a nivel de red hacia nuestro cluster, este equipo tambien puede ser un host con GNU/Linux, el cual poseera un Kernel compilado con el modulo "Netfilter" y con una serie de cadenas y reglas de filtrado elaboradas a través de "iptables".

El DHCP server sera necesario para configurar dinamicamente los nodos openmosix, debido a que a medida que este aumente en numero de nodos, nos brindará un menor tiempo de configuracion IP de los nodos openmosix.

El RSYNC server sera necesario para configurar de una forma centralizada los archivos de configuración de openmosix, tal como openmosix.map, si este cluster creciera geograficamente, los nodos distribuidos a lo largo de la WAN podrían adquirir facilmente la configuración por este metodo.

El LDAP Server sera necesario para realizar autenticación por PAM con el parche LDAP, lo que nos permitira mantener los perfiles de los usuarios que quieran acceder al cluster de forma centralizada.

Los nodos monitores, estaran en una DMZ, ya que si la Universidad Don Bosco lo requiere, estos podrían ser utilizados para realizar el monitoreo de otros servicios de red ajenos al cluster de alto rendimiento.

La activación de interfaces bond en cada uno de los nodos y hosts que brindan servicios de red al cluster mejorara la disponibilidad del cluster.

El nodo Central podrá tener configurado VNC-http, para que los usuarios del cluster puedan acceder a este a través de un Navegador caulquiera con plugin Java desde una maquina remota.

Esta configuración propuesta mejorará sustancialmente la disponibilidad y rendimiento del cluster en general.

CONCLUSIONES.

- El Cluster representa una herramienta para la solución de problemas de computo intensivo en el ambito académico de la Universidad Don Bosco.

- El cluster posee un diseño minimalista, una solución muy economica, esto se debe gracias a la elaboración de la distribución GNU/Linux de la Universidad Don Bosco "UDB Cluster GNU/Linux", el cual permite convertir cualquier maquina x86 en un nodo del cluster, por lo que es un cluster muy escalable.

- El Cluster de alto rendimiento se acerca mas al comportamiento ideal de un cluster a medida se realizan procesamientos de mayor carga.

- Los procesos se dividen en dos partes: la parte del usuario y la del sistema. La parte, o área, de usuario es transferida al nodo remoto mientras el área de sistema espera en el nodo raíz. Cada proceso tendrá un único nodo raíz (UHN, unique home node) que corresponde al nodo que ha generado dicho proceso. openMosix se encargará de establecer la comunicación entre estos dos procesos.

- 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 sigue 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.

- El montaje de un sistema de archivos raíz compartido por NFS en el nodo central vuelve mas fragil al sitema, debido a que si ocurre alguna falla en la red, que corte la comunicación con el nodo central a todos los nodos, todo el sistema se vendrá abajo, aunque el sistema este pasivo en un determinado momento, por eso se tomo la opción de crear la distribución UDB Cluster GNU/Linux, ya que vuelve mas robusto al cluster.

Referencias.

1. Mike Bennett Sr., Network Engineer Lawrence Berkeley Labs
http://www.force10networks.com/applications/roe.asp?content=1

http://www.gridcomputing.com/

2.http://www.linux-ha.org/

3. Beowulf Cluster computing with Linux by Thomas Sterling
MIT Press Publish date October 2001.ISBN 0262692740.

4. Beowulf History, by Phil Merkey
http://www.beowulf.org/

5. http://www.openmosix.org

Por Daniel Robbins,
http://www.intel.com/cd/ids/developer/asmo-na/eng/20449.htm

6.http://opensource.codito.com/migshm/
Por Mulyadi Santosa on Tue, 2004-01-06 02:00.
http://www.linuxjournal.com/article/7341

7. www.kernel.org

Por Mark Mithchell, Jeffrey Oldham, Alex Samuel.
http://www.advancedlinuxprogramming.com/

Por Robert Love
Linux Kernel Development ISBN: 0672327201 Novell Press

8. www.linuxfromscratch.org

9. by Wolgang Barth.
Nagios System and Network Monitoring.

http://www.nagios.org (Sitio Oficial de nagios).

10. www.cacti.net (Sitio Oficial de Cacti).