

EL PROYECTO DE CLUSTER SSI OPENMOSIX

David Santo Orcero

Desarrollador de las herramientas de área de usuario del proyecto OpenMosix
irbis@orcero.org, <http://www.orcero.org/irbis>

RESUMEN

OpenMosix es el único proyecto con licencia GPL con suficiente calidad para ser usado en entornos de producción capaz de implementar un cluster con un modelo SSI, frente a otros proyectos no libres –MOSIX– o que aún están en fase alfa –SSI-C– existentes en el mercado.

En este artículo vamos a estudiar los conceptos claves de OpenMosix, así como su historia y su funcionamiento.

1. LOS SISTEMAS SSI

Denominamos sistemas SSI –*Single System Image*– [1] a aquellos sistemas de clustering en los que todo el cluster ofrece al usuario la imagen de un único sistema, es decir, que el cluster entero se comportará para el usuario como una única máquina.

Desde un punto de vista menos formal, la idea de un cluster SSI es poder administrar y usar un cluster completo como si se tratara de una máquina SMP o, en el peor de los casos, de una máquina NUMA.

Los sistemas SSI siempre han sido de interés de la industria informática, y actualmente son objeto de investigación intensa en el área de los sistemas operativos distribuidos y de gran interés en la comunidad general, con espacio dentro de los foros de en clustering [2]. El problema que los SSI deben resolver es bastante complejo, ya que se supone que un sistema operativo SSI puro debe ser capaz de hacer transparente el paralelismo del cluster al proceso y al administrador. Esto significa que cualquier proceso debe poder migrar desde cualquier nodo del cluster a cualquier otro nodo y ejecutarse en él, independientemente de donde haya sido lanzado; y que puede hacerlo tantas veces como quiera, para así aprovechar mejor los recursos del sistema. Las aplicaciones, además, deben ser capaces de poder usar esta capacidad de migrar sin necesidad de ser recompiladas ni de enlazar ninguna biblioteca específica de migración.

Debemos destacar una diferencia fundamental en el plano económico de un sistema SSI puro respecto de un cluster: el costo total de propiedad. Como las aplicaciones no necesitan ser reescritas ni modificadas en un cluster SSI puro, mientras que sí lo necesitan en un cluster común, el coste

total de propiedad de un sistema SSI es mucho menor que el de un cluster común.

De entre los sistemas con alguna de las características de un sistema SSI que han existido –y existen– en el mercado, debemos destacar el VMS para VAX, de Digital, el Sysplex de IBM para su OS/390, o el sistema MOSIX [3]. Sin embargo, ninguno de estos sistemas SSI son libres.

Por otro lado, tenemos el SSI de Compaq, que ha tomado muchas ideas y código de MOSIX –como OpenMosix–, de la época en que MOSIX era libre; pero aún esta muy lejos de ser lo suficientemente estable como para poder ejecutarse en un cluster en producción.

En este artículo vamos a revisar el sistema libre SSI OpenMosix.

2. EL ORIGEN DE OPENMOSIX

El proyecto OpenMosix [4] es un fork libre del proyecto MOSIX. Por ello, vamos a comenzar estudiando la historia del proyecto MOSIX.

El proyecto MOSIX fue desarrollado por el profesor Amnon Barak y su equipo –Oren La’adan, Ammon Shiloh y Moshe Bar, entre otros– en la Universidad Hebrea de Jerusalén. Su desarrollo original fue parte de un proyecto financiado originalmente por el DARPA para un cluster de máquinas PDP-11/45 usado en las fuerzas aéreas norteamericanas. Después este sistema se adaptó también a clusters PDP-11/10. El sistema operativo era el Unix 6 de Bell Labs, y fue desarrollando entre 1977 y 1979. En aquella época, el proyecto se llamaba “*UNIX with Satellite Processors*”. La idea de este proyecto era parchear masivamente el kernel para permitir la migración automática de procesos según un algoritmo de equilibrado de carga del que hablaremos más adelante, de tal forma que el aprovechamiento de los recursos del cluster sea el mayor posible –y, por lo tanto, mayor la probabilidad de que un proceso determinado de uno de los nodos con más carga termine más deprisa, al no tener que competir por procesador con tantos procesos como competiría sin MOSIX–.

El resultado de este primer desarrollo fue excelente, y, tras un año dedicado a otras actividades, el prof. Barak comenzó a trabajar en el nuevo proyecto MOS. Este proyecto

siguió con las ideas del proyecto anterior; y supuso en la práctica una reimplementación completa del antiguo proyecto “*UNIX with Satellite Processors*” -“reencarnación”, tal y como el profesor Barak denominará las reimplementaciones de MOSIX en su bibliografía-. MOS fue desarrollado entre los años 1981 y 1982 para las máquinas PDP-11/45 y PDP-11/23, parcheando masivamente el sistema operativo Unix 7, también de Bell Labs.

En una tercera fase, que se desarrolló durante los años 1983 y 1984, el proyecto MOS fue portado a una máquina CADMUS/PCS MC68K, que ejecutaba el Unix 7 de Bell Labs con algunas extensiones de BSD 4.1.

Después de su adaptación a CADMUS/PCS MC68K, el proyecto MOS, aún completamente SSI, estuvo completamente detenido durante dos años. Pero en 1987 el profesor Barak retomó el desarrollo, ahora llamando a la nueva reencarnación de MOS con el nombre “NSMOS”. NSMOS fue desarrollado sobre una máquina NS32332, que ejecutaba AT&T Unix system V release 2. El proyecto NSMOS comenzó a dejar el área de kernel de cada proceso en el nodo raíz, ya que ya no era posible modificar todos los programas de área de usuario de Unix que eran precisos para mantener el área de kernel migrable. Este cambio, que se ha mantenido hasta nuestros días, permite que no sea necesario modificar ninguna de las aplicaciones de Unix para que sean ejecutadas en un cluster MOSIX/OpenMosix satisfactoriamente, a cambio de que el área de kernel de los procesos se ejecute siempre en su nodo raíz.

En 1998, el equipo del profesor Barak porta NSMOS a las arquitecturas VAX-780 y VAX-750, que también ejecutaban el AT&T Unix system V release 2. Como en el caso anterior, no es parcheado masivamente el sistema operativo, sino apenas su kernel. El proyecto pasa a llamarse MOSIX, nombre que conservaría hasta hoy en día.

Aún en 1988, el sistema SSI MOSIX es portado a arquitecturas NS32532, también con AT&T Unix system V release 2. Este porting llevó dos años, seguidos por dos años en que el profesor Barak paró el proyecto.

Ya entrando en los noventa, el profesor Barak comienza a trabajar con arquitecturas Intel. En 1991 comienza el porting de MOSIX a BSD/OS sobre 486, porting que llevaría tres años en completarse. Entre 1994 y 1997, volvería otra vez el profesor Barak a parar el proyecto, ahora durante cinco años.

En 1998, el profesor Barak vio que BSD/OS estaba desapareciendo, y decidió portar MOSIX a un sabor de Unix bastante popular en la época: Linux. Esta reencarnación de MOSIX sólo funcionaba para máquinas i80486 o superiores, y usaba el kernel 2.2. Un año más tarde de la reencarnación de MOSIX en Linux, por la naturaleza vérica de la licencia GPL con la que fue desarrollado el kernel de Linux, el profesor Barak se vio legalmente forzado a liberar el código de MOSIX. Este hecho fue aplaudido por alguno de los colaboradores del profesor Barak, tales como Moshe

Bar; aunque el profesor Barak no se quedó muy convencido del hecho. Las herramientas de área de usuario, imprescindibles para configurar y administrar un cluster MOSIX, no fueron liberadas hasta medio año más tarde; y debido a la presión de la comunidad Linuxera, no a ningún plan determinado del profesor Barak. Estas herramientas de área de usuario permanecieron libres hasta final del 2001.

Una vez liberadas las herramientas de área de usuario, el proyecto MOSIX se popularizó rápidamente, ya que permitía SSI real en clusters Linux, algo que otros paquetes aún sólo prometían. MOSIX fue incluido por defecto en la mayor parte de las distribuciones importantes, tales como Debian o Mandrake, y una parte importante de la comunidad intentó apoyar el proyecto MOSIX enviando parches y nuevas funcionalidades, sin que ninguno despertase el interés del profesor Barak.

En el año 2000 comenzó el porting al kernel 2.4 de Linux, que terminó un año más tarde. Este código también estaba liberado bajo GPL, pero el hecho de que MOSIX no avanzara durante todo el 2001 y que el profesor Barak no aceptaba cualquier tipo de ayuda externa que él no controlase directamente, terminó desencadenando el fork.

En el año 2002, la situación terminó estallando en las manos del profesor Barak. El profesor Barak decidió cambiar la licencia de MOSIX retirando el derecho a adaptar el código al uso propio allí donde le fue posible modificar la licencia sin violar la GPL. Además, el proyecto MOSIX llevaba un año parado, el profesor Barak estaba rechazando los parches que aportaban mejoras y nuevas funcionalidades a MOSIX, y estaba ocultando este hecho a sus colaboradores, entre ellos a Moshe Bar –el antiguo alumno del profesor Barak más brillante–, en esta época ya la mano derecha del profesor Barak y coresponsable del proyecto.

Todo esto originó que grupo importante del equipo que había desarrollado MOSIX y de usuarios de MOSIX entre los que se encontraba Moshe Bar –coadministrador del proyecto MOSIX y mano derecha del profesor Barak, y autor del MFS, y del DFSA–, Mark Veltzer –*Windows2Linux*, *sgmltools-lite*, contribuciones al CPAN–, Muli Ben Yehuda –*Syscalltrack*, *pptp*–, Brian Bilbrey –*LinuxBook*, gran cantidad de documentación libre para Linux–, Brian Pontz –*Apache log rotator*–, Louis Zechter, Michael Farnbach, Bruce Knox, y David Santo Orcero –el autor de este artículo– tomaron la última versión de Mosix que era posible demostrar en un tribunal que era GPL, y comenzaron a trabajar por su cuenta. Con modelo de desarrollo de bazar, bajo GPL y con CVS abierto a todos. Y, sorprendentemente, comenzaron masivamente las descargas de OpenMosix y las colaboraciones en forma de parches y localización de errores. Había comenzado el proyecto OpenMosix.

Durante los dos primeros años, el proyecto se centró en la actualización de código y corrección de errores. Moshe Bar fué el responsable de la parte de área de kernel, y David Santo Orcero fué el responsable de la parte de área de

usuario.

Actualmente el proyecto MOSIX tiene una licencia de código cerrado para una parte muy importante del código por la que no es legal modificarlo –y mucho menos distribuir las mejoras–, y ha estado los dos últimos años prácticamente sin avances. Por otro lado, el proyecto OpenMosix, completamente libre, bajo GPL y con modelo de desarrollo de bazar, ha tenido un rápido crecimiento en características y estabilidad.

3. QUÉ ES OPENMOSIX

OpenMosix es un conjunto de parches al kernel y unas utilidades y bibliotecas de área de usuario que permiten tener un sistema SSI completo para Linux. Al estar basado en el código de MOSIX, comparte algunas de sus características y limitaciones. Sin embargo, OpenMosix lleva medio año de actividad extremadamente activa, que MOSIX no ha tenido.

Por ejemplo, OpenMosix hace uso del parche de Rik van Riel de mapeado inverso de memoria, que permite que el proceso que consume más recursos de OpenMosix pase de tener una complejidad computacional de $O(n)$ a una complejidad de k . En la práctica, elimina una de las partes del código de OpenMosix que pueden consumir una cantidad apreciable de procesador.

Además de esto, OpenMosix carece de algunas de las condiciones de carrera de MOSIX gracias a un parche de David Santo Orcero, tiene autodetección de los nodos del cluster gracias a una aplicación de Louis Zechter, y gracias al trabajo de Moshe Bar ya funcionan correctamente las aplicaciones que usen hebras sobre OpenMosix. Gran parte de los desarrolladores de OpenMosix están trabajando en el porting a IA-64 y a la corrección de errores, con lo que globalmente OpenMosix es más rápido, eficiente y tiene más características innovadoras que MOSIX.

Se ha hecho también un esfuerzo importante en documentar correctamente OpenMosix: parte del equipo está escribiendo documentación moderna y precisa, y el proceso OpenMosix dispone de un FAQ, un HOWTO y una base de conocimientos Wiki propia.

Haciendo un sumario de lo que hace OpenMosix, en OpenMosix los procesos migran de un nodo a otro de un cluster Linux para ejecutarse en aquellos nodos que permitirán que el aprovechamiento de los recursos del cluster sea máximo. Esto significa que los procesos migrarán de CPUs sobrecargadas a CPUs libres, y que migrarán al nodo en el que físicamente están escribiendo los datos para escribirlos a más velocidad si escriben mucho en dicho nodo, usando el sistema de archivos de OpenMosix oMFS. Por último, los procesos que comiencen a swapppear en exceso migrarán a nodos en los que tengan memoria suficiente para ejecutarse sin swap. Todo esto se evalúa teniendo en cuenta el coste de la migración, para evitar el “ping-pong” de procesos –que

un proceso vaya y vuelva de un nodo constantemente–. En ningún momento es necesario ni recompilar ni reprogramar ningún proceso, por lo que tenemos un sistema SSI real, que nos permite hacer el máximo aprovechamiento de los recursos de cualquier cluster Linux, sea dedicado o no, para que podamos ejecutar en él la mayor parte de las aplicaciones posibles de la forma más rápida posible.

El modelo de programación de OpenMosix es “*fork and forget*”. Una vez que el proceso ejecuta la llamada “*fork*” –llamada estandar Posix para que un proceso se transforme en dos–, el hijo y el padre podrán ejecutarse en nodos distintos si así mejora el aprovechamiento global de los recursos computacionales del cluster. La migración de procesos y su ejecución efectiva en nodos distintos no quita que sus comunicaciones sean simples, ya que se pueden comunicar los procesos migrados entre sí tal y como se comunicaban antes de que migrasen, es decir, usando los mismos mecanismos de comunicación locales.

El código de OpenMosix se compone de dos partes bien diferenciadas: por un lado, una parte en área de kernel que consiste en un conjunto de parches al kernel para dotar al kernel de Linux de las funcionalidades de un cluster SSI. Este conjunto de parches incluye modificaciones al planificador de Linux que se activan cíclicamente cuando crece la carga, algunas inclusiones en el algoritmo de gestión del swap que se activan cuando la carga asociada al swap es intensa, rutinas de lanzamiento remoto de llamadas al kernel que se activan cuando un proceso migrado realiza una llamada al kernel que no puede resolver en el nodo de ejecución, un sistema de ficheros propio que permite acceder de forma local a las particiones remotas de las máquinas del cluster, y una rutina que permite escuchar en un puerto las llamadas al kernel lanzadas remotamente y que deben ser atendidas localmente, entre otros parches.

Por otro lado, las herramientas de área de usuario se componen de un conjunto de utilidades y bibliotecas en área de usuario que tienen como objetivo facilitar la tarea de administración y uso del cluster, así como una biblioteca con un API que nos permiten acceder a las funciones propias de OpenMosix. Las herramientas de área de usuario, por ejemplo, incluyen utilidades para configurar los nodos del cluster, para detectar automáticamente los nodos OpenMosix de la red e incluirlos en el cluster, una utilidad para lanzar procesos definiendo cómo van a migrar, una utilidad para forzar migraciones y forzar que vuelvan procesos migrados, y utilidades de monitorización del cluster, entre otras utilidades.

Debemos incidir en uno de los conceptos fundamentales a OpenMosix: aunque existan la biblioteca y las aplicaciones de área de usuario para tomar el control del cluster, no es necesario que el usuario o el programador usen las bibliotecas y las aplicaciones de área de usuario: un cluster OpenMosix, de por sí solo, realiza la migración automática de las tareas de un nodo a otro con objeto de asegurar que la utilización de los recursos computacionales del clus-

ter sea siempre óptima. La biblioteca de área de usuario nos permite controlar el cluster, decidiendo sobre la política de migración, dando información heurística al planificador de OpenMosix sobre lo que el proceso piensa hacer y forzando migraciones, pero las aplicaciones pueden migrar sin usar la biblioteca.

4. EL MECANISMO DE MIGRACIÓN AUTOMÁTICA DE PROCESOS EN OPENMOSIX

En OpenMosix, determinados procesos pueden ejecutarse en un nodo distinto al que se lanzaron. En este caso se dice que el proceso ha migrado, y que no se ejecuta su área de usuario en el nodo raíz. En el nodo raíz –el nodo donde se generó el proceso– solo se ejecutarán las llamadas al kernel que no puedan ser ejecutadas en el nodo de ejecución por falta de información o por hacer uso dicha llamada al kernel de los recursos locales al nodo raíz.

Un proceso no puede migrar y se le marca como no migrable si cumple alguna las siguientes condiciones:

- El proceso usa segmentos de memoria compartida System V. Desgraciadamente, este es el caso del sistema gestor de base de datos PostgreSQL. Esto supone una limitación de OpenMosix heredada de MOSIX que el equipo de OpenMosix está trabajando para eliminarla.
- El proceso se ejecuta en modo de emulación VM86. Desgraciadamente, este es el caso del VMware; pero es un problema que casi nunca se da en programas de Linux y que tiene muy mala eliminación por la naturaleza del modo de emulación VM86 –si migrásemos un proceso en este estado, la carga de llamadas al nodo raíz sería tan alta que habría sido contraproducente haberlo migrado–.
- El proceso ejecuta instrucciones en ensamblador propias de la máquina donde se lanza, y que no tiene la máquina de destino. Esto es una limitación evidente, ya que por muy bueno que sea OpenMosix jamás podremos lanzar un programa compilado para un Pentium IV que use las instrucciones propias de Pentium IV en un Pentium. Esta condición, en principio, no es una condición demasiado preocupante, ya que casi todas las distribuciones de Linux tienen todos sus programas compilados para Pentium. Programas que usen instrucciones más específicas en clusters heterogéneos pueden provocar que el proceso vuelva antes de tiempo a su nodo raíz, o que sólo pueda migrar a un número determinado de nodos.
- El proceso mapea memoria de un dispositivo en la RAM, o acceder directamente a los registros de un

dispositivo. Esta limitación es evidente, ya que el acceso a los recursos hardware del nodo raíz hace que no sea rentable su migración a otro nodo. El caso más destacado de proceso que viola esta limitación es el servidor de X-Window, aunque hay algunos programas que no pueden migrar por el uso de la función “mmap”. Actualmente el equipo está viendo como evitar este problema.

En principio, todos los procesos que estén declarados como migrables –propiedad heredada del padre del proceso, pero que puede ser modificada con las herramientas de área de usuario– pueden migrar. En el momento que un proceso no migrado cumpla alguna de las condiciones anteriormente citadas, no podrá migrar más hasta que deje de violar las restricciones –salvo en el caso de usar instrucciones de una arquitectura distinta, en cuyo caso queda marcado como no migrable permanentemente–.

Por otro lado, si un proceso migrado intenta cumplir alguna de las condiciones anteriores –por ejemplo, porque intente reservar o usar un segmento de memoria compartida, o intente usar directamente un registro hardware de la máquina– el proceso migrado genera un trap y será suspendido por el kernel, para ser enviado al nodo raíz –el nodo donde se originó el proceso–. Una vez que el proceso esté otra vez en el nodo raíz, el proceso será planificado para ejecución de nuevo, y volverá a ejecutar la operación prohibida en su nodo raíz, siendo marcado dicho proceso como no migrable hasta que deje de violar las restricciones.

5. EL ALGORITMO DE EQUILIBRADO AUTOMÁTICO DE CARGA

En OpenMosix las migraciones pueden ser de dos clases: pueden ser forzadas explícitamente por el usuario propietario de un proceso, por el grupo propietario de un proceso, o por root; pero también puede ser migraciones automáticas de procesos. Vamos a hablar de la migración automática de procesos.

Antes de entrar en detalle en el algoritmo de migración automático de carga, debemos destacar un detalle importante. El algoritmo de equilibrado automático de carga es completamente distribuido: cada nodo decide localmente si va a migrar o no algún proceso, y lo decide cuando el nodo que toma la decisión esté sobrecargado, decisión que tomará en base a datos parciales de uso del cluster recolectados por él y almacenados localmente. Esta naturaleza distribuida del algoritmo permite que los clusters OpenMosix puedan escalar hasta los miles de nodos sin problemas –de hecho, actualmente hay clusters OpenMosix con miles de nodos en producción–, y además permite que no haya un nodo central que sea punto único de fallo del cluster completo, y que en caso de caída de dicho nodo central caiga el cluster entero. En caso de que caiga un nodo en un cluster OpenMosix,

esta caída solo afecta a los procesos ejecutados y generados en el nodo caído: esto es especialmente importante en clusters grandes, ya que en estos cada día siempre hay algún nodo que falla.

El algoritmo de migración automático de carga es un algoritmo de optimización que emplea como función de costo el uso de los recursos del sistema de un cluster, incluyendo el uso que se hace de la memoria y el uso que se hace del procesador de los nodos de un cluster, entre otros parámetros.

El algoritmo que empleamos en OpenMosix tiene un orden de complejidad lineal respecto al número de nodos del cluster del que tenemos información local, lo que sin lugar a dudas favorece la escalabilidad del cluster.

El algoritmo de migración automático de carga busca maximizar el uso de los recursos de un cluster en base a la información que tiene sobre la carga de los nodos y el uso del procesador y del disco que hacen los procesos. Esto permite que, por regla general, los programas se ejecuten más rápido de lo que se ejecutarían en un cluster sin equilibrado automático de carga, aunque no tienen que ejecutarse necesariamente todos los procesos más rápido.

Veamos con un ejemplo la lógica de la función de costo. Supongamos que tenemos dos nodos en nuestro cluster. Un nodo ejecuta dos procesos, y el otro nodo no ejecuta ningún proceso. El uso del procesador que hacen en el primer nodo los procesos es del 100%; pero en el segundo nodo el uso que hacen del procesador es del 0%. Supongamos ahora que migramos uno de los dos procesos al nodo desocupado: el uso del procesador del primer nodo caerá probablemente al 95%, pero el uso del procesador en el segundo nodo subirá al 95%. El uso del cluster ha aumentado $-100+0$ es menor que $95+95$; y, como efecto secundario, estamos aprovechando mejor los recursos del cluster y los procesos se ejecutarán más rápido.

Como algoritmo de optimización OpenMosix usa un algoritmo de tipo greedy. Este algoritmo es extremadamente simple y muy eficiente, por lo que conseguimos soluciones buenas a nuestro problema sobrecargando poco el sistema.

El algoritmo de equilibrado de carga no se lanza constantemente, para no perjudicar el rendimiento del cluster. Este algoritmo sólo se lanza cuando el nodo que lo lanza esté demasiado cargado. Entonces toma la decisión de que si lo mejor será aguantar el exceso de carga, o que será mejor para aprovechar el cluster mandar alguno de los procesos ejecutados localmente a algún nodo remoto menos cargado.

Para realizar esta operación, un nodo OpenMosix no cuenta con toda la información que describe todo el estado de todo el cluster. Esto haría prohibitiva la recolección de esta información, especialmente en clusters grandes.

OpenMosix toma un planteamiento alternativo para recolectar la información. Cada cierto tiempo le pregunta a un grupo de nodos, escogidos aleatoriamente de entre la lista de nodos, toda la información que necesita para operar. Con el tiempo, consigue tener información de un número

suficiente de nodos como para tomar decisiones razonadas. Aunque la decisión individual de migración de un proceso puede que no sea la óptima para el mejor aprovechamiento del cluster, en su conjunto todas las decisiones distribuidas de todos los nodos si son óptimas. De hecho, el profesor Barak ha demostrado matemáticamente de que el algoritmo usado por MOSIX y por OpenMosix para un tiempo infinito tiene el mismo rendimiento que un algoritmo óptimo de ventana finita –es decir, un algoritmo que ve el futuro, pero que no es capaz de ver el futuro más allá de una fecha en particular–.

6. EL MEMORY USHERING

Como hemos comentado anteriormente, en caso de sobrecarga el algoritmo de OpenMosix evalúa donde va a mandar los procesos y qué procesos va a mandar usando para ello gran cantidad de datos; entre ellos, y destacando como los más importantes, el uso de la memoria y el uso del procesador. Sin embargo, hay una circunstancia crítica en la cual este algoritmo sería inefectivo: es el caso del trashing.

Denominamos trashing al fenómeno por el cual el sistema está siendo completamente inusable debido al alto uso que hacemos del swap.

La condición de trashing es especialmente crítica, ya que cuando un nodo comienza a hacer trashing, su eficiencia pasa a ser pésima. Por ello, el trashing es algo que OpenMosix intenta evitar a toda costa. Y lo evita gracias al sistema de memory ushering.

El sistema de memory ushering es un sistema que usa el mismo algoritmo que el sistema de equilibrado de carga –un greedy–, pero que tiene como función de coste la memoria física utilizada –función que intentará maximizar– y como condición de activación que un proceso comience a hacer swap intensamente. Es un sistema predictivo, que evita de antemano llegar al trashing.

El subsistema de memory ushering es considerado más crítico que el de equilibrado automático de carga. Por ello, el subsistema de memory ushering siempre tiene prioridad sobre el subsistema de equilibrado automático de carga de OpenMosix, es autónomo a este, y se lanza de forma independiente.

7. EL MECANISMO DE MIGRACIÓN DE 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 seguirá ejecutándose como si siguiese en el nodo de origen: si escribe o lee en el disco, lo hará a través del nodo de origen –lo que supone leer o grabar remotamente en el nodo de origen si el sistema de archivos

donde se lee o se escribe corresponde a un sistema de archivos local al nodo de origen—. Si hacemos un “ps” en el nodo origen de un proceso migrado, el proceso migrado aparentemente estará aún ahí, aunque realmente no esté ocupando recursos de procesador. Si hacemos un “ps” en el nodo al que ha migrado un proceso, no veremos el proceso: el proceso se está ejecutando como una hebra del kernel.

Una vez que el proceso migra, no se ejecuta por completo en el nodo de destino. Algunas de las llamadas al kernel se ejecutan en el nodo raíz. Por ello, una de las funciones del mecanismo de migración de procesos es, dado un proceso migrado, capturar todas las llamadas al kernel que haga el proceso migrado, seleccionar aquellas llamadas que no pueden ser resueltas en el nodo en el que se ejecuta actualmente el proceso migrado—frecuentemente porque necesitan un recurso que sólo está en el nodo de origen del proceso migrado— y mandar dichas llamadas al kernel por red al nodo raíz para que se ejecuten en él. Por ello, decimos que el área de usuario de un proceso en OpenMosix se ejecuta en el nodo migrado, y el área del sistema se ejecuta habitualmente en el nodo raíz. Esto permite que los procesos migrados mantengan todos los descriptores de ficheros abiertos, y que puedan seguir operando con su entorno de la misma forma que si se ejecutasen en el nodo raíz. Esta modificación se hizo en MOS para evitar tener que reescribir el sistema operativo entero y las aplicaciones para funcionar en MOS, y ha sido heredada por MOSIX primero y por OpenMosix después.

Destacamos un ejemplo importante: una aplicación interactiva migrada. El usuario no percibe que la tarea ha migrado, ya que la interacción con el usuario se realiza en área de kernel, por lo que la petición de datos se hace en el nodo raíz, y el procesamiento de los datos se hace en el nodo de ejecución.

El área de usuario corresponde al segmento de código, el segmento de datos, el segmento de pila y el segmento de heap, así como las estructuras internas del kernel necesarias para mantener la integridad de estas estructuras. En la migración, todos estos segmentos y estructuras son enviados a través de la red al nodo de destino. Una vez que el proceso ha migrado, se ejecuta en el nodo de destino como una hebra del kernel.

Por otro lado, el área de kernel en OpenMosix corresponde a la pila del kernel asociada al proceso, y todas las estructuras de datos relativas al uso de los recursos del kernel y el uso de los recursos de la máquina raíz que emplea el proceso. Observamos que no hay segmento de código; es lo que es normal y correcto, ya que el código que se ejecuta en área de kernel es código del kernel del sistema operativo. Los segmentos y estructuras relativos al área del kernel no migran al nodo de destino. Siempre están en el nodo raíz del proceso, y serán usados por el kernel cuando el proceso migrado haga una llamada al kernel para poder atender a esta llamada correctamente. Las llamadas al kernel que pueden

ser resueltas sin estas estructuras son las que no usan recursos de la máquina raíz o de las estructuras de datos del kernel de la máquina raíz; y, por lo tanto, pueden ser resueltas en el nodo al que migró el proceso. Pero esto no incluye las llamadas que consumen más recursos—llamadas a disco y a red—, que siempre se ejecutarán en el nodo raíz.

8. LA COMUNICACIÓN ENTRE LAS DOS ÁREAS

Un dato importante que podemos tener interés en estudiar es cómo se realiza la comunicación entre el área de usuario y el área de kernel en OpenMosix. Desgraciadamente, no tenemos espacio en este artículo para estudiarla entera, ni tendremos tiempo en la presentación para comentarla con detalle, por lo que recomendamos la lectura de [5], [6], [7], [8] y [9] donde encontramos descritos también los escenarios de comunicación “proceso migrado-¿nodo raíz” por una llamada al kernel, “nodo raíz-¿proceso migrado” por una señal, y la suspensión de procesos migrados.

9. CONCLUSIÓN

OpenMosix es una prometedora plataforma SSI para Linux, bajo licencia GPL y de desarrollo de bazar que ya permite usar clusters SSI en entornos de producción enteramente con software libre, y permitirá en el futuro a los investigadores en clustering nuevos avances en el campo de la computación distribuida.

10. BIBLIOGRAFIA

- [1] David Santo Orcero, “Los clusters SSI,” *Todo Linux*, , no. 22, pp. 57–60, 2002.
- [2] “<http://www.hispacluster.org>,” .
- [3] “<http://www.mosix.org>,” .
- [4] “<http://www.openmosix.org>,” .
- [5] David Santo Orcero, “El algoritmo de migración de OpenMosix,” *Todo Linux*, , no. 23, pp. 57–60, 2002.
- [6] David Santo Orcero, “Cómo construir un cluster OpenMosix,” *Todo Linux*, , no. 24, pp. 57–60, 2002.
- [7] David Santo Orcero, “Configurando la topología de un cluster OpenMosix,” *Todo Linux*, , no. 25, pp. 58–61, 2002.
- [8] David Santo Orcero, “Las herramientas de área de usuario de OpenMosix,” *Todo Linux*, , no. 26, pp. 58–61, 2003.
- [9] David Santo Orcero, “Optimizando un cluster OpenMosix,” *Todo Linux*, , no. 27, pp. 56–60, 2003.