Distribución de la Carga y Migración de Procesos en Sistemas Distribuidos

Por

Carlos Troncoso
Ricardo Maicas

Directores:

Msc. CS. Guillermo Delbue
Lic. Roberto Bevilacqua

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

2003

Agradecimiento

Quisiéramos aprovechar esta oportunidad para agradecer en primer lugar a nuestro primer director de tesis, Guillermo Delbue, por su apoyo y guía durante nuestros primeros pasos en el desarrollo de este trabajo, sus invalorables sugerencias e ideas nos ayudó a encausar nuestra tarea.

También deseamos agradecer a nuestro actual director de tesis, Roberto Bevilacqua, por toda su colaboración y esfuerzo brindado hacia nosotros para que podamos concretar este trabajo.

Por último, quisiéramos expresar nuestra gratitud a todos los miembros del Departamento de Computación de la facultad de Ciencias Exactas y Naturales (Universidad de Buenos Aires), por su vocación hacia la enseñanza y por la calidad del conocimiento que nos impartieron durante todos estos años de estudio.

Índice

1. Resumen

El ansia por obtener más potencia de cómputo ha sido una de las principales causas del gran desarrollo de las computadoras en los últimos años. El estudio de los sistemas distribuidos, pretende satisfacer los deseos y necesidades de procesamiento a gran escala, y en las consecuentes mejoras en los tiempos de cómputos.

En todo ambiente de procesamiento distribuido, resulta importante asegurar que la carga de trabajo sobre cada nodo de la red esté adecuadamente balanceado.

Este trabajo presenta una completa implementación de un mecanismo de distribución de carga dinámico y descentralizado. El mecanismo de distribución de la carga implementado se ha inspirado en un algoritmo propuesto por Kang Shin y Yi-Chieh Chang en su trabajo sobre Carga Compartida en Sistemas Distribuidos [ref. 1].

Para toda implementación de distribución de carga, se hace necesario un mecanismo de migración de procesos, lo cual implica interrumpir la ejecución del proceso seleccionado, y continuar su ejecución en otro nodo. Para cumplir con este requerimiento, hemos implementado un mecanismo de migración utilizando checkpoint de procesos para salvar el estado de un proceso y luego restaurarlo en el nodo destino seleccionado. Dicho mecanismo está basado en el utilizado por el sistema Condor [ref. 2,16], al cual se la han hecho mejoras para salvar algunas de las limitaciones que este posee, como ser la no necesidad de enlazar los programas de usuario con las librerías de migración/checkpoint y la posibilidad de migrar procesos que se comuniquen a través de sockets.

Abstract

The eager to increase the computing power has been one of the main causes of the great computer development in the last years. Studies on distributed systems try to satisfy the wishes and needs of activities with high computing processing for getting a better performance.

A balance of the workload on each node in the network is important in every distributed processing environment.

This work shows a complete implementation of a dynamic and decentralized mechanism of load distribution. The implemented mechanism was inspired in an algorithm proposed by Kang Shin and Yi-Chieh Chang in their work about Load Sharing in Distributed Systems [ref.1].

Every load distributed implementation needs a mechanism of process migration which implies the interruption of a selected process' execution and its restoring on a different node. To do this request, we have implemented a mechanism of migration using checkpointing to save the process' state, and after that, restore it on a selected target node. This implementation was based on the Condor system's mechanism, on which we have done improvements to overcome some restrictions that Condor has, for example: our implementation doesn't need to re-link user's programs with libraries of

migration/checkpoint, and the possibility of migrate process that could establish a communication through sockets.

2. Introducción

Debido al hecho de que los procesadores son cada vez más rápidos, la combinación del poder de procesamiento de las estaciones de trabajo en red, puede llegar a niveles que antes sólo era posibles de encontrar en supercomputadoras. A su vez, las redes locales actualmente disponibles, llegan a velocidades de transferencia de hasta 1 Gbps. Ante este nuevo escenario, promover la migración de tareas desde equipos sobrecargados de trabajo hacia otros que no lo estén se muestra imprescindible, sobre todo para aquellas tareas que hagan uso intensivo del procesador por largo tiempo. Este contexto está estimulando diversos estudios sobre la distribución de la carga sobre sistemas distribuidos en redes.

Un sistema distribuido no es más que una red convencional de computadoras independientes que contribuyen entre sí para ejecutar uno o varios procesos. Estos sistemas cada vez tienen mayor relevancia y comienzan a desplazar a las potentes estaciones de trabajo o mainframes, tanto en empresas, universidades, instituciones científicas, etc. Son numerosas las ventajas que presenta un sistema distribuido respecto a un mainframe:

- **Velocidad**: Pueden alcanzar un mayor nivel de procesamiento gracias al gran número de computadoras que conforman el sistema distribuido total.
- **Escalabilidad**: Un sistema de computadoras en red es fácilmente escalable, simplemente hay que añadir una nueva computadora a la red.
- **Fiabilidad**: si una computadora falla, el sistema puede continuar en funcionamiento.
- **Económica**: con el mismo precio que cuesta un mainframe, se pueden comprar decenas de computadoras personales bastante potentes que juntas resultan muchísimo más potentes.

Un sistema distribuido puede utilizarse como una máquina multiprocesador de memoria distribuida, para la ejecución de aplicaciones con grandes requisitos de procesamiento. Esta máquina virtual está conformada por un conjunto de computadoras o nodos, cada uno de los cuales tienen capacidades propias de cómputo, memoria, entrada/salida y comunicaciones que en el caso de los sistemas heterogéneos puede variar sustancialmente de unos a otros.

Los sistemas distribuidos proveen una alternativa económica a los tradicionales sistemas paralelos. Usando sistemas distribuidos, los investigadores no se encuentran limitados al poder de cálculo de un solo lugar, y pueden desarrollar y ejecutar aplicaciones que requieren un gran poder de cálculo, que va mas allá de lo que puede estar disponible en una determinada computadora.

Usuarios en diferentes computadoras pueden iniciar tareas en forma distribuida. Estas tareas, en general y dependiendo del problema, son asumidas como independientes. Una tarea en ejecución es llamada un proceso y cada proceso requiere y hace uso de recursos del sistema.

Las características claves de los sistemas distribuidos es el de soportar recursos compartidos, concurrencia, escalabilidad, tolerancia a fallas y transparencia.

No hay dudas acerca de la importancia de los sistemas distribuidos en el desarrollo de procesamientos de alto rendimiento para las siguientes décadas. Existe un gran número de políticas de distribución de la carga que han sido propuestas en los últimos años buscando resolver el problema del desequilibrio de la carga entre diferentes computadoras en un sistema distribuido.

Un usuario que se registra en una estación de trabajo de una red, no tiene conciencia de que el sistema es distribuido. Los archivos que pertenecen a un usuario pueden estar almacenados en otra computadora del sistema. El proceso iniciado por un usuario podría estar ejecutándose en una computadora diferente. Todo esto es transparente para el usuario, ya que él no está advertido de que su trabajo podría ser ejecutado por otras computadoras.

Cada nodo conectado al sistema distribuido tiene su propia memoria privada y su propia capacidad de procesamiento, algunos tienen almacenamiento propio y todos comparten el recurso de comunicación. Estos recursos son considerados como arquitecturalmente homogéneos, por lo que ellos pueden servir a requerimientos efectuados por procesos que se ejecutan en otras computadoras del sistema. Además, algunos recursos pueden ser compartidos de manera transparente.

En un sistema distribuido, un nodo del sistema podría estar desocupado, mientras que otros podrían estar teniendo una pesada carga de trabajo. Entonces puede resultar de gran ayuda si utilizáramos una estrategia de distribución de carga para eliminar estas desparejas relaciones de carga. Para poder utilizar de manera eficiente los recursos de computación provistos por un sistema distribuido, un mecanismo de distribución de carga debería estar presente dentro del esquema de ejecución del sistema. Por eso es que la distribución de la carga en un muy importante aspecto dentro de estos sistemas.

Hay distintas maneras de resolver el problema del equilibrio de carga en un sistema distribuido. Dependiendo de cómo esté administrada la información y como sea la arquitectura del mecanismo de distribución de la carga, podemos encontrarnos con algoritmos centralizados y algoritmos distribuidos. Atendiendo a la toma de decisiones se encuentran los algoritmos estáticos frente a los dinámicos.

- **Centralizados**: Son aquellos en los que un nodo central se encarga de todo el peso del algoritmo, es el que tiene toda la información acerca del estado del resto de los nodos y el que toma las decisiones pertinentes. Los demás nodos, por lo tanto, sólo tienen información de ellos mismos, y solo se comunican con el nodo central para informar acerca de su estado. Esta arquitectura es conocida como Master-Slave. Estos algoritmos tienen la ventaja de su gran sencillez y de su eficiencia con un número reducido de nodos, pero son mayores las desventajas, como ser que el nodo central se transforme en un cuello de botella para el sistema y el problema que esto representa, al poseer un límite en su grado de escalabilidad.

- **Distribuidos**: Son aquellos en los que no existe un nodo central, sino que todos los nodos tienen información del estado del sistema y toman sus propias decisiones, dependiendo tanto de su propio estado como del estado del sistema en general. Estos algoritmos son más complicados que los centralizados y se genera una mayor cantidad de mensajes entre nodos, pero son muchísimo más eficientes porque eliminan los problemas de escalabilidad y los cuellos de botella, al no haber un nodo central que cargue con la responsabilidad de la administración del trabajo del resto de los nodos que conforman el sistema distribuido. Otra dificultad adicional es que al tener todos los nodos información del sistema, puede haber problemas de coherencia al estar replicada la información.

- **Estáticos**: Son aquellos en los que se toman decisiones a priori, de forma determinista o probabilística, antes de comenzar a ejecutar la aplicación y sin tener en cuenta el estado actual del sistema. Esta solución puede ser efectiva cuando la carga se puede caracterizar suficientemente bien antes de la ejecución, pero no es realmente eficiente si se producen numerosos cambios en el estado del sistema.

- **Dinámicos**: Son aquellos en los que se utiliza la información del estado del sistema para tomar decisiones, por lo que potencialmente mejoran a los algoritmos estáticos al mejorar la calidad de las decisiones. Sin embargo, se produce una pequeña penalización, debido a la sobrecarga producida por la necesidad de tener que recoger información del estado del sistema en tiempo real.

Nuestro proyecto implementa un algoritmo distribuido y dinámico para resolver el problema de la distribución de la carga, intentando conseguir de este modo, una mejor y más óptima distribución del esfuerzo de procesamiento entre los nodos del sistema.

3. Distribución Dinámica de la Carga

Un Sistema Distribuido es como una colección de computadoras autónomas conectadas a través de una red, con un software diseñado para producir una integración del poder de cómputo de todas ellas. Las estaciones de trabajo interconectadas por una LAN (red de área local) es el ejemplo más común de Sistemas Distribuidos.

El objetivo primario de un mecanismo de distribución de carga distribuido debería ser, además de lograr una adecuada distribución de la carga sobre cada uno de los nodos, el de minimizar la comunicación remota entre ellos, ya que esto es uno de los puntos débiles que posee la distribución de la información del estado del sistema en tiempo real entre los distintos nodos de la red.

Decisiones acerca de cómo compartir la carga entre los nodos pueden ser establecidas de manera estática o dinámica.

La ejecución de un algoritmo de equilibrio de carga dinámico requiere de una serie de mecanismos para mantener una vista consistente del estado del sistema y una política de negociación para migrar la carga de trabajo (ya sean procesos o datos) entre los distintos nodos del sistema.

A continuación se presenta la estructura más utilizada para dividir los algoritmos de distribución de carga dinámicos según las funcionalidades que este debe presentar [ref. 3]:

- Política de medida de estado
- Política de intercambio de información
- Política de iniciación
- Operación del distribución de la carga
    - Política de localización
    - Política de distribución
    - Política de selección

3.1. Política de medida de estado

Los algoritmos dinámicos de distribución de la carga utilizan información sobre el estado del sistema para tomar decisiones. Se basan fundamentalmente en la información sobre la capacidad y la carga de los nodos.

Un índice de carga debe ser un buen estimador del tiempo de respuesta de las tareas que se ejecutan en un determinado procesador. Sin embargo, debemos estimar la carga de trabajo basándonos en algún conjunto de parámetros mensurables, tales como el número de tareas, la cantidad de comunicación, la tasa de cambios del contexto, el tamaño de la memoria disponible, etc. [ref. 4]

Un sistema distribuido es un sistema muy dinámico y su estado puede cambiar drásticamente en un breve instante de tiempo. La medida del índice debe hacerse muy frecuentemente, para tener una idea real del estado del sistema, por lo que esta medida debe ser calculada rápidamente para evitar grandes sobrecargas. Por lo tanto, este índice de carga no debería estar conformado por un gran número de parámetros ni tampoco requerir una alta carga de procesamiento para su obtención. En su lugar se utiliza un conjunto pequeño de parámetros y alguna regla heurística para estimar la carga. La elección de un buen índice de carga tiene un efecto considerable en el rendimiento de los algoritmos de distribución de carga, ya que este índice es el que va a representar el estado de los nodos del sistema y del sistema en general. Además, los índices sencillos, tales como el número de procesos en la cola de ejecución o la memoria consumida por dichos procesos, son particularmente efectivos, ya que la medida del estado del nodo produce muy poca sobrecarga. [ref. 5]

3.2. Política de intercambio de información

Esta política especifica como recolectar y mantener la información sobre la carga de los distintos nodos que componen el sistema, necesaria para tomar decisiones relacionadas con el distribución de carga. Idealmente, un nodo debería conocer en todo momento el estado real del resto. Sin embargo, en la práctica, esto no es siempre posible debido a retrasos en las comunicaciones intrínsecos a los sistemas de memoria distribuida y agudizados en el caso de un sistema distribuido, por causa de la latencia de la red. Esta latencia provoca que las comunicaciones sean el cuello de botella del sistema, y por lo tanto, los intercambios de información deben reducirse al menor número posible. Una buena regla de intercambio debe encontrar un equilibrio entre tener un bajo costo de obtención de la información y mantener una vista precisa del estado de todos los nodos del sistema. Este compromiso se sigue en las siguientes tres reglas de obtención e intercambio de información:

- **Por demanda**: los nodos recogen la información sobre el estado de los demás, solamente cuando es necesario realizar una operación de distribución de carga. [ref. 6]
- **Periódica**: los nodos informan periódicamente al resto acerca de su estado, independientemente de si la información se va a utilizar o no. En este caso la imagen que cada nodo tiene del sistema puede no estar actualizada, e incluso cada nodo puede tener una imagen del sistema diferente. La ventaja de este modelo es que cuando es necesario hacer una operación de distribución se dispone de toda la información y no es necesario sincronizar a todos los nodos. [ref. 7]
- **Por cambios en el estado del sistema**: en este último caso los procesadores envían su información de estado, solamente cuando esta cambia en cierto grado, es decir cuando los cambios superan ciertos umbrales o cotas predeterminadas. [ref. 3]

La política bajo demanda minimiza el número de mensajes intercambiados, pero pospone la recolección de la información de carga hasta el instante en el que se tiene que realizar la operación de distribución. Su principal desventaja es que introduce un

retardo adicional en las operaciones de distribución, al tener que sincronizar y recibir la información de todos los nodos.

Por el contrario la política periódica permite que las operaciones de equilibrio se inicien inmediatamente, basándose en la información de estado mantenida de forma local. Esta regla se utiliza principalmente con políticas de iniciación periódicas. El problema en este caso es determinar la duración del intervalo de intercambio. Un intervalo excesivamente pequeño provoca una fuerte sobrecarga de comunicación, mientras que un intervalo excesivamente grande penaliza la precisión de la información. La política basada en cambios de estado es un compromiso entre ambas políticas anteriores.

Todas las políticas discutidas anteriormente son distribuidas pues todos los nodos mantienen información sobre los demás independientemente. Basándose en esta información, los nodos pueden tomar decisiones de distribución de carga individualmente. Para mantener una vista global del estado del sistema es necesario utilizar operaciones de comunicación colectivas, cuya sobrecarga se incrementa de forma polinomial con el tamaño del sistema. Por lo tanto, en sistemas muy grandes el costo de mantener una vista global del sistema es prácticamente imposible. Una solución más práctica consiste en hacer intercambios de información local entre un subconjunto de nodos, llamado dominio. Esto reduce los intercambios de información a un subconjunto de nodos, entre los que también se producen las operaciones de distribución de carga. Además de la información de estado actual, se puede utilizar información de las decisiones previas, para tomar una nueva decisión de distribución.

3.3. Política de iniciación

Esta política determina cuando se debe dar comienzo a una operación de distribución de carga. La ejecución de una operación de distribución lleva asociada una sobrecarga considerable y por lo tanto la decisión de iniciarla debe ponderar si los beneficios pueden superar dicha sobrecarga de trabajo.

Generalmente las operaciones de distribución de la carga pueden ser iniciadas por un nodo sobrecargado, denominadas *iniciadas por el emisor* [ref. 8] o bien iniciadas por un nodo infrautilizado, denominadas *iniciadas por el receptor* [ref. 9], o periódicamente en tiempo de ejecución [ref. 10]. Las dos primeras políticas necesitan distinguir si el estado del nodo es normal o bien está en uno de los estados anteriormente comentados. Un método típico es definir un umbral superior por encima del cual se considera que el nodo está sobrecargado, y un umbral inferior por debajo del cual se considera que el nodo está infrautilizado.

Para tomar las ventajas de los dos métodos anteriores se han propuesto también políticas de iniciación simétricas [ref. 11,12]. Cambian entre iniciadas por el emisor o por el receptor en tiempo de ejecución. Pero para hacer este intercambio deben tener los umbrales adecuados lo cual es muy complejo en la práctica. Para evitar esto, se definen los estados sobrecargados e infrautilizados como medidas relativas. Bajo esta política un procesador inicia una operación cuando su carga está por encima o por debajo de un porcentaje de la que tenía desde la última operación. Una alternativa es definir el estado

de un procesador relativo a sus vecinos directamente conectados con él. Si su carga es superior a la media de sus vecinos su estado es sobrecargado, si su carga es inferior a la media de sus vecinos su estado es infrautilizado.

3.4. Operación de distribución de Carga

Una operación de distribución de carga queda definida por tres políticas: política de **localización**, política de **distribución** y política de **selección**.

Toda operación de distribución de carga necesita de la participación de al menos dos nodos, uno que la inicia y otro con el que se equilibra la carga. La *política de localización* determina el *socio* en la operación de distribución. Al conjunto de nodos con los cuales uno dado puede intercambiar su carga se le denomina *dominio de equilibrio*. La *política de distribución* por su parte determina como distribuir la carga entre los nodos dentro de dicho dominio. Por último, la *política de selección* selecciona qué proceso o conjunto de datos son los más adecuados para la transferencia entre los nodos que participan en la operación.

Según la política de selección un algoritmo de distribución de la carga se puede clasificar de dos maneras: con interrupción o sin interrupción. Los algoritmos sin interrupción siempre seleccionan procesos recién creados, mientras que los algoritmos con interrupción pueden seleccionar procesos que ya están en ejecución. La migración de un proceso que ya está en ejecución requiere que se suspenda el proceso, se transfiera todo su estado y se reinicie en el nuevo nodo, lo cual es mucho más costoso que el caso anterior. [ref. 13]

Además de la sobrecarga de transferencias, la política de selección también debe tener en cuenta la sobrecarga de en la comunicación, debida a la migración del proceso. Por ejemplo, si se separan dos procesos fuertemente acoplados, esto generará una fuerte sobrecarga de comunicaciones en el futuro que puede reducir las ventajas del distribución.

Las políticas de localización y distribución juntas toman las decisiones de distribución de la carga. El *dominio de equilibrio* se puede caracterizar como global o local. Si el dominio es global el socio se puede buscar a lo largo de todo el sistema, mientras que si es local se restringe esta búsqueda a un conjunto reducido de vecinos o nodos asociados. Estas reglas tienen que ir acompañadas por información global o local respectivamente. En general, los algoritmos basados en información y política de localización local se denominan algoritmos de vecino más próximo. Estos son iterativos en el sentido de que pueden ser necesarios varios pasos para alcanzar el equilibrio. [ref. 14]

4. Características relacionadas con la migración de procesos y la distribución de la carga

4.1. Política y Mecanismo

Existen dos tópicos que se encuentran fuertemente relacionados con la distribución de la carga. La política trata con mediciones de la carga de un sistema y la comunicación de esta información con el resto de los nodos del sistema. Decide cuando un proceso debería ser migrado desde un nodo a otro. El mecanismo especifica el método de transferencia de un proceso a otro nodo, asegurando que el proceso obtenga aproximadamente el mismo ambiente como el que este poseía en el nodo original.

4.2. Migración de tipo *preemptive* y *non-preemptive*

Una migración del tipo *preemptive*, involucra la transferencia de un proceso en ejecución de un nodo a otro. Esto requiere detener al proceso y transferir sus tablas de información del proceso y su espacio de direcciones hacia el nodo receptor de la migración, en donde el proceso será restaurado. Por el otro lado, una migración del tipo *non-preemptive* comienza un nuevo proceso sobre un nodo diferente, por esto, un proceso de migración de tipo *non-preemptive* es a menudo llamado de ejecución remota transparente.

Debido a que en una migración del tipo *non-preemptive* no existe transferencia del espacio de direcciones, este puede ser fácilmente implementado con una menor sobrecarga de trabajo. Se ha verificado que una migración de tipo *preemptive* genera una gran sobrecarga al tener que transferir el espacio completo de direcciones del proceso, lo que podría producir una dilapidación de sus beneficios, por eso este tipo de migración es beneficioso solo en casos donde existen tareas con un alto grado de procesamiento. Además, la heterogeneidad en sistemas operativos y arquitecturas no pueden ser llevadas a cabo utilizando la migración *preemptive*, ya que los binarios para dos sistemas operativos diferentes serán, con seguridad, incompatibles, mientras que en el caso de migraciones *non-preemptive*, la heterogeneidad puede ser más fácilmente abordada debido a que un nuevo proceso es iniciado sobre un nodo destino.

4.3. Distribución de la carga y balanceo de carga

Otros dos ítems que se encuentran relacionados son la distribución de la carga y el balanceo de carga. En la distribución de la carga, el sistema intenta compartir la carga de un nodo sobrecargado, migrando un proceso desde esta maquina a otra maquina con poca carga; en cambio en el balanceo de carga, el sistema intenta mantener una carga similar sobre todos los nodos del sistema. El balanceo de carga no necesariamente ofrece mejores rendimientos, debido a que este puede requerir una mayor cantidad de migraciones de procesos.

4.4. Transparencia

Uno de los más deseables objetivos de una implementación de sistemas distribuidos es la transparencia. Esto significa que el usuario debería tener la ilusión de estar trabajando sobre un solo sistema de uso compartido, en vez de tener la visión de que se esta trabajando sobre un conjunto de computadoras conectadas a través de una red. El sistema debería permitir a los usuarios acceder a cualquier recurso sin que estos se preocupen acerca de su ubicación física. Existen algunas soluciones desarrolladas, tal como NFS, los cuales proveen acceso transparente a los archivos en la red, sin embargo, no existen soluciones universalmente aceptadas que puedan proveer un acceso remoto transparente a CPUs o dispositivos.

4.5. Heterogeneidad

La heterogeneidad puede ser debido a diferentes tipos de arquitecturas, sistemas operativos y/o configuraciones. En un ambiente de trabajo típico, existen computadoras con diferentes arquitecturas que ejecutan posiblemente diferentes sistemas operativos. Las computadoras pueden tener distintas configuraciones (monto de memoria RAM, disco rígido, etc.). Por lo tanto, un sistema que toma en cuenta la heterogeneidad del ambiente de trabajo, puede utilizar de mejor forma los recursos que este posee.

4.6. Escalabilidad y Rendimiento

Un sistema distribuido es escalable si su rendimiento no decae al incrementarse el número de nodos que lo conforman. Uno de los principios fundamentales al diseñar estos sistemas es el de evitar centralizar componentes. Estos componentes centralizados podrían recibir una alta sobrecarga de trabajo si la cantidad de nodos del sistema crece y pueden transformarse en puntos con alta probabilidad de fallas del sistema, ya que si estos componentes fallan, el sistema por completo podría detener su normal funcionamiento. Además, una gran cantidad de transferencia de mensajes entre nodos debería ser evitada ya que estos pueden generan una gran cantidad de tráfico en la red, lo cual puede hacer disminuir el rendimiento del sistema.

5. Estado del Arte

En el pasado, múltiples mecanismos de distribución de la carga y migración de procesos han sido desarrollados. Estos sistemas pueden ser clasificados, de una manera genérica, en dos categorías: Implementaciones a nivel de usuario e implementaciones a nivel de kernel. Ambas categorías tienen sus propias ventajas y desventajas. Aquí daremos un vistazo general de algunas implementaciones desarrolladas junto con sus características

5.1. Implementaciones a nivel de usuario

La ventaja que ofrecen las implementaciones a nivel de usuario es que ellas pueden ser fácilmente instaladas sobre diferentes variantes de Unix. Sin embargo, ellas son típicamente mas lentas que las implementaciones a nivel de kernel y además establecen severas restricciones con respecto al tipo de aplicaciones que pueden participar de la distribución de la carga debido a que no tienen acceso a las estructuras del kernel. Una de los principales restricciones es que la mayoría de este tipo de implementación posee es que las aplicaciones no pueden utilizar ningún mecanismo de comunicación entre procesos (IPC). Algunos sistemas tampoco permiten que los procesos migrados ejecuten algunas llamadas al sistema tales como fork, exec, etc. Otra de las principales desventajas es que a menudo los programas deben ser vinculados con librerías especiales antes de que ellas puedan ser consideradas para una migración.

Tales sistemas, generalmente trabajan interceptando las llamadas al sistema con la ayuda de una librería modificada Estas librerías proveen funciones para la extracción del estado de los procesos los cuales incluyen las tablas del proceso y el espacio de direcciones, volcando dicha información a un archivo. La información que se almacena en este archivo es usada luego para reiniciar el proceso sobre el nodo receptor de la migración del proceso. El mecanismo de transferencia es transparente para el usuario, pero el programa debe ser re-vinculado con la librería modificada.

A continuación hablaremos sobre algunos de los sistemas que caen dentro de esta categoría.

5.1.1. Condor

Condor [ref. 2,16] provee facilidades para checkpointing y migración. Este provee su propio conjunto de librerías de funciones, las cuales son responsables el trabajo de extracción del estado del proceso y restauración del mismo en el nodo remoto. El acceso remoto a los archivos es provisto por un proceso *shadow* que es mantenido en el nodo original. Cuando se necesita acceder a un archivo, el requerimiento es reenviado al nodo original que posee el proceso *shadow*, el cual ejecutará la operación y luego retornará los datos. Todas las llamadas al sistema relacionadas con archivos son interceptadas en la librería del usuario y reenviadas al nodo original usando mecanismos de llamadas a procedimientos remotos (RPC).

Aunque esto es útil para cálculos intensivos y tareas de larga duración, es un mecanismo lento y no soporta comunicación entre procesos ni señalización. Además, el código del usuario debe ser re-vinculado con la librería de checkpoint de Condor para crear un programa que pueda ser candidato para la migración. Por lo tanto, aplicaciones existentes cuyo código fuente no esta disponible no pueden ser migradas, lo cual limita severamente su utilidad. Además, algunas de las operaciones (tales como manipulación de archivos) son realizadas por el nodo original, lo cual genera una dependencia con este, llamada *dependencia residual*. Por otro lado, esta implementación es altamente portable, ya que está completamente implementada en el nivel de usuario y ha sido instalada en un gran numero de plataformas.

5.1.2. Utopia

Utopia [ref. 26] es una facilidad de distribución de carga para ambientes heterogéneos, los cuales no soportan un migración de procesos ya iniciados (preemptive migration). Este asume que existe un espacio de nombres de archivos compartido a través del sistema (por ejemplo NFS). El sistema consiste de un Administrador de Información de Carga (Load Information Manager – LIM), el cual es usado para medir e intercambiar la información de carga. Existe un servidor de ejecución remota (Remote Execution Server – RES) que provee mecanismos para una transparente ejecución remota. Hay también una librería de carga compartida (Load Sharing Library – LSLIB) que es usada para desarrollar aplicaciones de carga compartida. Las aplicaciones son de dos tipos: las que directamente usan la interface provista por LSLIB y las que usan tales aplicaciones como su contexto de ejecución. Estas últimas aplicaciones no requieren ser re-vinculadas para poder ser ejecutadas remotamente, pero los procesos hijos creados por ellos sobre algún nodo remoto no pueden ser ejecutados remotamente. Cuando un proceso tiene que ser ejecutado remotamente, el padre se contacta con el servidor de ejecución remota (RES) de la máquina destino y envía el ambiente de ejecución local del proceso. Este ambiente es creado en el nodo remoto por el proceso remoto antes de que este comience. El RES también maneja las entradas y las salidas de tareas remotas, así como también las señales de Unix.

Aunque el sistema soporta ejecución remota, este no es completamente transparente, así como tampoco soporta grupos de procesos distribuidos. Además, algunas aplicaciones tienen que ser rescritas para utilizar las ventajas de la ejecución remota. Las aplicaciones que no son rescritas, obtienen solo una limitada ventaja ya que sus procesos hijos no pueden ser ejecutados remotamente. De este modo, la utilidad del sistema decrece cuando las posibilidades de carga compartida se van perdiendo.

5.1.3. Global Layer Unix

GLUnix [ref. 27] provee un mecanismo de ejecución remota transparente de aplicaciones secuenciales y paralelas. Este usa identificadores de procesos globales (Network PIDs-NPIDs) y números de nodos virtuales (Virtual Node Numbers – VNNs) para proveer un amplio espacio de nombres de procesos en el sistema. También soporta Entrada/Salida de terminales remotas y señalización. Además provee un soporte

especializado de planificación para programas paralelos. También usa NFS para proveer una vista uniforme del sistema de archivos a lo largo del sistema.

Este mecanismo consta de tres componentes: un maestro por sistema, un demonio por nodo y una librería por aplicación. El maestro del GLUnix mantiene el estado del sistema (por ejemplo: la carga de información sobre cada nodo) y genera decisiones centralizadas de ubicación de recursos. El demonio del GLUnix colecta información de carga local y la envía al maestro, además sirve como un proxy local para el maestro al inicializar y enviar señales a procesos, así como para detectar la finalización de un proceso. La librería del GLUnix provee una API para aplicaciones que explícitamente requieren servicios del GLUnix. Las aplicaciones existentes pueden ejecutarse sin modificaciones con la ayuda de un proceso stub que es requerido para la ejecución remota de un proceso en un nodo del sistema.

Aunque GLUnix provee una ejecución remota transparente, posee algunas limitaciones, por ejemplo, no soporta heterogeneidad, al tener un servidor maestro, esto puede incrementar la posibilidad de caídas del sistema completo.

5.2. Implementaciones a nivel de kernel

Las implementaciones a nivel de kernel permiten una compatibilidad binaria sobre aplicaciones existentes, esto es, las aplicaciones existentes pueden ejecutarse sin ser compiladas o re-vinculadas. Estas implementaciones soportan un mas amplio rango de aplicaciones que las que soportan las implementaciones a nivel de usuario. Ellas también son mas eficientes, comparadas con las implementaciones de nivel de usuario. Pero, la mayor desventaja que poseen es que no son fácilmente portables a diferentes variantes de Unix, ya que es mas difícil hacer cambios dentro del kernel que implementándolas a nivel de usuario, sobre todo si no se dispone del código fuente del sistema operativo.

Estos sistemas pueden ser implementados desde cero, tal como el Amoeba [ref. 16], System V[ref. 16], Sprite [ref. 16] y LOCUS [ref. 16] o pueden ser implementadas realizando cambios a los kernel de Unix existentes, tal es el caso de MOSIX [28] y Solaris MC [ref. 29].

En esta sección presentaremos algunos sistemas que implementan la migración de procesos a nivel de kernel.

5.2.1. V Distributed System

El V Distributed System [ref. 16] es un sistema operativo diseñado para un cluster de estaciones de trabajo conectadas por una red de alta velocidad. Cada nodo ejecuta una copia separada del kernel, los cuales cooperan para provee una única abstracción del sistema. Este cluster de nodos es llamado un dominio V.

El kernel contiene tres principales componentes: comunicación entre procesos (IPC), servidor de kernel, y servidor de dispositivos. El IPC provee la base para

conectar diferentes componentes del sistema. El servidor de kernel provee operaciones de administración de procesos y memoria a los procesos. El servidor de dispositivos provee acceso y administración de dispositivos usando el protocolo V I/O el cual esta basado en un objeto abstracto llamado objeto uniforme de entrada/salida (uniform I/O object - UIO). Los procesos pueden acceder transparentemente a cualquier servidor dentro de un dominio V usando mensajes. El emisor y el receptor de un mensaje son especificados por identificadores de procesos. V soporta *preemptive process migration* para realizar la distribución de la carga. La naturaleza de los mecanismos de IPC sin conexión hacen mas fácil la migración, aun para procesos que se comunican con otros procesos. Si un proceso es migrado mientras esta sirviendo un requerimiento, el cliente es informado de que el requerimiento no pudo ser satisfecho debido a que el proceso fue migrado.

La principal desventaja de este sistema es que usa difusión de mensajes para asegurar que las identificaciones de los procesos sean únicas en toda la red, haciendo que el sistema no sea escalable. Además, este sistema no soporta heterogeneidad.

5.2.2. LOCUS

LOCUS es un sistema operativo distribuido, el cual soporta ejecución remota de procesos, migración de procesos y un sistema de archivos distribuidos. Este sistema de archivos soporta replicación, lo que provee tanto rápido acceso como tolerancia a fallos. También provee acceso transparente a dispositivos y a pipes. El sistema de archivos distribuidos implementado por LOCUS, provee una única visión del sistema de archivos tanto a usuarios como a aplicaciones. Esto se hace a través de una única estructura de árbol para toda la red. Esta estructura también soporta descriptores de archivos compartidos entre dos procesos ejecutando en distintos nodos de la red. Utiliza un esquema de tokens, donde el proceso que tiene el token, es quien tiene permiso de acceso al archivo, mientras que el otro tiene que esperar a tener el token. También provee recuperación de fallos y tolerancia a fallos, por utilizar replicación de almacenamiento. Para proveer migración de procesos y ejecución remota, LOCUS utiliza un identificador global de procesos (unique site identifier). Una lista de nodos a donde el proceso puede ser migrado es mantenida en el ambiente del proceso, y un usuario puede agregar o borrar nodos de esa lista.

Ejecución remota entre arquitecturas heterogéneas se implementa utilizando directorios ocultos. Estos directorios ocultos contienen programas ejecutables para diferentes arquitecturas, y el programa apropiado es seleccionado transparentemente por el sistema, cuando invocamos a la llamada al sistema execve() con el nombre del directorio oculto.

5.2.3. Sprite

Sprite usa migración de procesos en forma transparente para utilizar nodos inactivos. El diseño de Sprite apunta a dos objetivos principales: la distribución de la carga debe ser transparente para el usuario, y cuando un nodo inactivo retorna a la

actividad debido a la creación de nuevos procesos propios de ese nodo, los procesos migrados a dicho nodo deben inmediatamente retornar al nodo origen. Estos procesos expulsados no tienen ninguna dependencia residual en el nodo del cual fueron expulsados. La implementación de la migración de procesos es a nivel de kernel y la distribución de la carga a nivel de usuario. Sprite utiliza el concepto de nodo origen, el cual es el nodo donde el proceso tendría que estar siempre ejecutándose de no existir la migración. Este sistema operativo utiliza dos formas de hacer llamadas al sistema. Aquellas llamadas al sistema que sean independientes del nodo donde ejecuta, son hechas en el nodo donde está ejecutando, y aquellas que dependan de donde se ejecutan, son ejecutadas remotamente en el nodo origen. Sprite provee acceso remoto a dispositivos y archivos, y utiliza identificadores de procesos únicos para toda la red.

Cada nodo de la red mantiene un demonio que monitorea la carga de ese nodo. Cuando un nodo está infrautilizado, informa de ese estado a un nodo llamado "central migration server", el cual es el encargado de mantener la lista de nodos infrautilizados o inactivos.

5.2.4. Mosix

Mosix es sistema estilo Unix mejorado, con algoritmos adaptativos de carga balanceada, para la utilización eficiente de recursos, en un ambiente homogéneo de nodos.

Utiliza migración de procesos con suspensión de ejecución para migración transparente de procesos, para así mantener la distribución de la carga.

Cada proceso tiene un nodo origen único (Unique Node Home- UNH), el cual es el nodo donde el proceso fue creado. El contexto del proceso es dividido en dos partes: el contexto de usuario, llamado remoto, el cual consiste de código, stack, datos y registros, y el contexto de sistema, el cual consiste de la parte del proceso que es dependiente de nodo donde fue creado, y el stack del kernel para ejecución del código del sistema. Cuando un proceso es migrado, se migra solamente el contexto remoto, mientras que el contexto del sistema, permanece en el nodo origen. Todas las llamadas al sistema que sean dependientes del sistema son ejecutadas remotamente por el contexto de sistema del proceso, mientras que las otras son manejadas en el nodo donde el proceso está ejecutando.

Mosix no tiene un nodo que centralice el control de la distribución de la carga. Cada nodo es capaz de operar independientemente y tomar todas sus decisiones con respecto a la distribución de la carga de manera independiente. Mosix no soporta heterogeneidad.

5.2.5. Solaris MC

Solaris MC es un sistema operativo distribuido , que provee una única imagen del sistema a través de todos los nodos de la red.

Está desarrollado como un nivel de abstracción del sistema operativo Solaris, y provee las mismas APIs que dicho sistema operativo.

Este nivel de abstracción se desarrolló como una colección de objetos en C++ utilizando CORBA. Utiliza un sistema de archivos distribuidos llamado "proxy filesystem" (PXFS).

La administración de procesos, se realiza por intermedio de un nivel de abstracción por sobre el kernel de Solaris, que maneja una visión global de los procesos. Este nivel de abstracción consiste de objetos proceso virtual (vproc), por cada proceso local, y un objeto administrador de procesos para cada nodo.

El objeto vproc maneja de manera global la relación padre/hijo entre procesos. El objeto administrador se utiliza para localizar vproc, nodos y realizar operaciones en los nodos.

Solaris MC tiene algunas limitaciones, como ser no poder compartir descriptores de archivo entre padre/hijo, cuando estos están en nodos distintos.

6. Alcance de nuestro trabajo

El objetivo de nuestro trabajo es el de desarrollar un mecanismo de distribución de la carga y migración de procesos en ambientes homogéneos, sobre un sistema operativo no concebido como distribuido, en donde todos los nodos participantes de este mecanismo puedan compartir transparentemente sus recursos. Dicho mecanismo debe seguir un modelo localidad-independiente, es decir, donde la ejecución de un proceso pueda desarrollarse de manera independiente al nodo de la red donde se ejecute.

El sistema propuesto solo soporta migración del tipo *preemptive*, por lo tanto es adecuado para procesos con alto consumo de procesador y de larga duración, debido al alto costo que representa migrar un proceso en ejecución. La implementación de este sistema se realizó a nivel de usuario, lo cual permite que la solución no sea exclusiva para un determinado sistema operativo. Se emplea un algoritmo de distribución de carga dinámico basado en la difusión de cambios de estado, lo que permite una reducida carga adicional sobre el nodo, así como también un mínimo tráfico adicional sobre la red.

Esta tesis describe la implementación de este sistema sobre una plataforma GNU/Linux con kernel 2.4.18.

7. Implementación desarrollada

7.1. Distribución de Carga: Algoritmo Implementado.

En esta sección se pretende dar una visión global de cómo funciona el algoritmo de distribución de la carga, cuales son sus principales componentes y cómo interactúan entre sí, sin profundizar demasiado en los detalles de la implementación, los cuales serán analizados mas adelante.

El algoritmo de distribución de carga implementado se ha inspirado en un algoritmo propuesto por Kang Shin y Yi-Chieh Chang en su trabajo sobre Carga Compartida en Sistemas Distribuidos en Tiempo Real [ref. 1].

Un nodo posee dos fuentes de arribo de tareas, las tareas externas y las tareas transferidas, y un solo servidor de tareas (en el caso de nodos con un procesador). Las tareas que arriban en cada nodo pueden ser ejecutadas localmente, o remotamente en algún otro nodo del sistema.

Los sistemas distribuidos son candidatos naturales para sistemas con altas exigencias de cálculo, debido a su potencial de procesamiento de alto rendimiento y a la confiabilidad que se logra utilizando múltiples computadoras. Sin embargo, si las tareas a ejecutar no son correctamente distribuidas entre los nodos del sistema, algunos nodos puede terminar sobrecargados de trabajo, entorpeciendo de este modo el rendimiento general del sistema, mientras que otros se encuentran libres o infrautilizados. Una forma de aliviar este problema es mediante una correcta distribución de la carga entre los nodos, así algunas tareas ingresadas sobre los nodos sobrecargados serán transferidas a nodos desocupados o con poca carga para su ejecución.

La distribución de la carga, en sistemas distribuidos de propósito general ha sido estudiada extensamente por muchos desarrolladores. Tal como fue expuesto anteriormente, las decisiones acerca de cómo compartir la carga entre los nodos pueden ser estáticas o dinámicas. Una decisión estática es tomada de manera independiente al estado actual del sistema, mientras que una decisión dinámica depende del estado del sistema en el momento de tomar la decisión. Las distribuciones de carga estáticas pueden ser también vistas como una ubicación no determinística de tareas en un sistema, en donde un nodo sobrecargado Ni transferirá alguna de sus tareas al nodo Nj con una probabilidad Pij, la cual es independiente del actual estado del sistema. Aunque la distribución de carga estática es simple y fácil de analizar con modelos de encolado, su potencial beneficio es limitado, ya que no se adapta por si mismo cuando el estado del sistema varía con el paso del tiempo. Por ejemplo, aun cuando Ni esté sobrecargado, tiene que aceptar tareas de otros nodos con la misma probabilidad que tenía cuando estaba desocupado. Por otro lado, cuando un algoritmo de distribución de carga dinámico es utilizado, un nodo sobrecargado puede transferir sus tareas a otros nodos usando la información del estado actual del sistema. Debido a que cualquier política

dinámica requiere que cada nodo conozca el estado de los otros nodos, esto los convierte en más complejos que aquellos que utilizan una política estática. La ventaja de una política dinámica, es que estos algoritmos adoptan decisiones cuando, a medida que va transcurriendo el tiempo, va variando el estado del sistema, y de este modo, pueden superar el problema asociado con los algoritmos de distribución de carga estáticos.

También expusimos que los algoritmos de distribución de carga pueden ser: *iniciadas por el emisor* o *iniciadas por el receptor*, dependiendo de cuales nodos inician el proceso de transferencia. El nodo al cual envían tareas externas es el nodo emisor, y el nodo que procesa esas tareas es el nodo receptor. En la aproximación *iniciada por el emisor*, un nodo origen sobrecargado inicia la transferencia de una nueva tarea externa, basado en alguna estrategia, mientras que en el caso de la aproximación *iniciada por el receptor*, un nodo desocupado o infrautilizado evalúa a cada uno de sus potenciales nodos emisores con quien podría compartir su carga.

Además, un algoritmo de distribución de carga esta compuesto de una política de transferencia y una política de locación. La política de transferencia determina cuando un nodo debería transferir sus tareas al encontrarse sobrecargado. La política de locación determina hacia que nodo enviará las tareas. Sus objetivos son los de minimizar el tiempo de respuesta promedio del sistema, moviendo tareas desde nodos sobrecargados a nodos desocupados. Un simple umbral es comúnmente utilizado por la política de transferencia, esto es, cuando la longitud de la cola de un nodo excede su umbral, el nodo intentará transferir sus tareas a otros nodos.

El método de distribución de carga de Shin y Chang intenta superar la debilidad de la mayoría de los métodos de distribución de carga genéricos al recolectar información del sistema, proponiendo un nuevo esquema de distribución de carga en el cual cada nodo necesita mantener la información del estado de solo un pequeño conjunto de nodos, llamados *nodos asociados*.

Dentro de este método, un nodo puede estar en uno de cuatro estados posibles: Under ("U"), Medium ("M"), Full ("F") y Over ("V"). En nuestra implementación utilizaremos la longitud de la cola para medir la carga de un nodo (llamamos QL a la longitud de la cola) y tres umbrales o cotas, llamadas THu, THf y THv, serán usadas para definir el estado de carga de un nodo. Un nodo está en estado "U" si QL <= THu, en estado "M" si THu<QL<=THf, en estado "F" si THf< QL <= THv, y en estado "V" si QL>THv.

Siempre que un nodo cambie de estado, ya sea porque superó o retrasó alguna cota definida debido al arribo, transferencia y/o finalización de alguna tarea, este difundirá su cambio de estado a todos los otros nodos de su conjunto de nodos asociados. Cada nodo que reciba esta información actualizará su propia información del estados de los nodos, eliminando o agregando al nodo que envió la información, de su lista de potenciales nodos receptores de tareas. Esta lista ordenada es denominada lista de *nodos preferidos*. Un nodo sobrecargado seleccionará el primer nodo en su lista de nodos preferidos y transfiere una tarea a ese nodo seleccionado.

Podemos observar que este método es completamente diferente a los métodos *iniciados por el emisor* convencionales, en donde un nodo sobrecargado evalúa otros nodos con quien compartir el trabajo; por el contrario, en este método se transfieren

tareas desde nodos sobrecargados a nodos desocupados usando la difusión del cambio de estados entre los respectivos grupos de nodos asociados.

El uso de una lista de nodos preferidos produce una transferencia de tareas excedentes sólo a los primeros nodos de la lista. Más importante aún, diferentes conjuntos de nodos asociados son armados superponiéndose entre si unos a otros, ya que las tareas deben ser distribuidas uniformemente sobre el sistema completo, y no tan solo sobre un conjunto reducido de nodos.

En el método de distribución de carga propuesto, cada nodo debe mantener y actualizar la información del estado de los otros nodos. Un nodo sobrecargado puede transferir una tarea a otro nodo basado en la información de estados, sin pérdida de tiempo por tener que examinar el estado de los otros nodos. Para implementar este método debemos desarrollar:

- Medios eficientes para recolectar y actualizar la información del estado de cada nodo. Esta recolección de información de estados no debe entorpecer las comunicaciones normales del sistema, tales como la comunicación entre tareas y la transferencia de tareas.

- Un método automático para seleccionar un nodo servidor, en caso que haya más de un nodo desocupado, minimizando la probabilidad de que más de un nodo sobrecargado transfiera simultáneamente sus tareas excedentes al mismo nodo desocupado.

Estos tópicos son abordados a continuación.

7.1.1. Información de estados

Para recolectar la información de estados, se debe decidir desde cuales nodos la información debería ser recolectada y con que frecuencia esta información debería ser actualizada. Un método directo es que cada nodo colecte y actualice la información de estado de todos los nodos del sistema en un intervalo de tiempo fijo. Sin embargo, es muy difícil determinar un intervalo apropiado para la recolección y actualización, que asegure la exactitud de la información de estados y a su vez mantenga en un nivel aceptable el tráfico de red adicional generado por la propia recolección de la información de estados de los nodos. Aunque un intervalo de tiempo corto asegura la exactitud de la información de estado, esto introducirá una gran sobrecarga al tráfico de la red del orden de N^2 cada vez que se requiera recolectar información de estados, en donde N es el número de nodos en el sistema. Esto podría, a su vez, generar un severo entorpecimiento en las comunicaciones normales entre tareas y en las transferencias de tareas, degradando de este modo el rendimiento del sistema. Por otra parte, la sobrecarga de tráfico decrecerá si se disminuye el intervalo de frecuencia de recolección y actualización de la información de estados, pero esto podría causar que la información de estados registrada en un nodo sea obsoleta. Por ejemplo, si el estado de un nodo ha cambiado, pasando de infrautilizado a sobrecargado antes de la siguiente actualización de información de estados, otros nodos podrían transferir sus tareas a este nodo que ahora se encuentra sobrecargado, basados en información obsoleta.

Idealmente, cada nodo debería mantener la información de estados lo más exacto y actualizado posible, a la vez que mantenga la sobrecarga de tráfico en la red tan bajo como sea posible. Para alcanzar este objetivo, se propone la difusión "broadcast" de los cambios de estados dentro del conjunto de nodos asociados de cada nodo para recolectar y actualizar la información de estados en el nodo. Así, cada uno de los nodos necesita mantener solo la información de estados de un pequeño conjunto de nodos del sistema. Cada nodo difundirá su cambio de estado al resto de su conjunto de *nodos asociados*, sólo si este pasa de un estado infrautilizado a sobrecargado y viceversa. Ya que un nodo recibirá nueva información solo cuando el estado de otro nodo en su conjunto de nodos asociados haya sido modificado, cada uno de ellos tendrá la exacta información del estado de todos los otros nodos en su conjunto de nodos asociados, logrando que esta recolección y actualización no genere un significativo aumento en el tráfico de red del sistema. Ya que el conjunto de nodos asociados de cada nodo esta formado por aquellos nodos con vecindad física, la demora generada en la difusión el cambio de estado y/o transferencia de tareas no debería, en general, ser demasiado larga. Además, como veremos, el trafico de red adicional generado por la difusión de los cambios de estados puede ser controlada ajustando las cotas THu y THf.

7.1.2. Lista de nodos preferidos

En la política de locación del tipo *iniciados por el emisor*, un nodo sobrecargado transferirá una tarea al primer nodo desocupado que haya sido encontrado durante el chequeo de estados de los nodos asociados. El problema de la política de locación propuesta, es que un nodo sobrecargado puede encontrar más de un nodo infrautilizado en su conjunto de nodos asociados, y más de un nodo sobrecargado podría seleccionar el mismo nodo hacia el cual transferir sus tareas. Uno debe, por lo tanto, establecer una regla para seleccionar un nodo receptor de tareas entre los múltiples potenciales nodos desocupados disponibles, minimizando la probabilidad de que más de un nodo sobrecargado transfieran simultáneamente sus tareas al mismo nodo infrautilizado. La lista de nodos preferidos nos ayudará a resolver este problema. Debido a que cada nodo mantiene la información de estado de los otros nodos en su conjunto de nodos asociados, uno puede organizar un orden en cada una de las lista de nodos preferidos de cada nodo. Así, el primer nodo de esta lista será el nodo ***más preferido*** y el segundo de la lista será el ***segundo nodo más preferido***, y así sucesivamente. Notar que el orden de preferencias cambia con el tiempo, por ejemplo, si el nodo *más preferido* se transforma en sobrecargado, entonces el *segundo nodo más preferido* pasará a ser el nodo *más preferido*, en caso de que no se encuentre sobrecargado en ese momento. Un nodo sobrecargado transferirá sus tareas a su nodo *más preferido* disponible en su lista de nodos asociados.

Basado en la topología del sistema, el orden estático de los nodos en cada lista de nodos preferidos para cada nodo del sistema, es permutado de manera tal que un nodo sea el nodo *más preferido* de uno y solo un nodo en el correspondiente conjunto de *nodos asociados* dentro del sistema. Este orden no cambiará con el tiempo, aunque algunos nodos sean eliminados de la lista de receptores disponibles cuando estos se

transformen en nodos sobrecargados, y del mismo modo recuperarán sus posiciones en la lista cuando se conviertan nuevamente en nodos infrautilizados.

Debido a que cada nodo posee un primer nodo preferido que no puede repetirse como primer nodo preferido de cualquier otro nodo, esto permite reducir la probabilidad que ocurra el problema de que más de un nodo sobrecargado vuelque sus tareas sobre un mismo nodo infrautilizado [ref.1] (La probabilidad se ve reducida frente a la alternativa de que este mismo suceso ocurra utilizando una lista de nodos sin un orden definido). Sin embargo, este caso podría llegar a suceder si, por ejemplo, el tercer nodo más preferido de un nodo, llamado Nx, de un nodo sobrecargado N0 pueda transformarse en su nodo *más preferido*, mientras que Nx es también el nodo *más preferido* de otro nodo sobrecargado.

7.1.3. Recolección de la información de estados

Como mencionamos anteriormente, tres umbrales o cotas, THu, THf y THv son utilizadas para determinar el estado de un nodo. Estas cotas pueden ser aplicadas a distintas mediciones de carga de un nodo (Ejemplos de mediciones sobre las que aplicar las cotas pueden ser la longitud de la cola de tareas o la acumulación del tiempo de ejecución de las tareas.). En nuestro trabajo utilizamos la longitud de la cola de tareas (QL).

Teniendo en cuenta que los nodos pueden tener cuatro diferentes estados de carga (Under ("U"), Medium ("M"), Full ("F") y Over ("V")), se plante el siguiente esquema: Un nodo en estado "U" puede aceptar una o más tareas de otros nodos. Un nodo en estado "F" no puede aceptar tareas de cualquier otro nodo pero puede completar todas sus propias tareas en tiempo y forma. Un nodo en estado "V" se encuentra en serias dificultades para completar sus tareas, y de este modo, debe transferir, si es posible, alguna de sus tareas a otros nodos. Ya que un nodo en estado "U" puede compartir el trabajo de otros nodos, se dice que el nodo esta en modo compartido. Un nodo en estado "F" nunca aceptará tareas provenientes de otros nodos ni transferirá sus propias tareas a otros, y se dice que está en modo independiente. Un nodo en estado "V" debe transferir algunas de sus tareas, y se dice que está en modo de transferencia. Notar que el estado "V" es usualmente un estado transitorio, porque si el arribo de una nueva tarea a un nodo lo transforma a este en un nodo de estado "V", el nodo transferirá esta tarea a algún otro nodo desocupado y entonces luego se volverá a convertir a estado "F". Sin embargo, si un nodo en estado "V" no puede encontrar un nodo en estado "U" dentro de su conjunto de nodos asociados, este se verá forzado a mantener su estado en "V".

De acuerdo a nuestra difusión de cambios de estados, cada nodo difundirá el cambio de estado a todos los otros nodos dentro de su conjunto de nodos asociados, solo cuando este pase del estado "U" al estado "F" ,del estado "M" al estado "F" o del estado "F" al estado "U". Una vez recibido el cambio de estado, cada nodo dentro del conjunto de nodos asociados actualizará la información de estado del nodo emisor.

Dos cotas diferentes, THu y THf, son utilizadas en el algoritmo de distribución de carga propuesto debido a la siguiente razón; si solo una cota fuese utilizada, un nodo

debería estar en estado "U" o "F" cuando su longitud de cola sea menor o mayor a esa cota. En tal caso, un nodo en estado "U" puede pasar al estado "F" después de recibir una tarea desde otro nodo, y un nodo en estado "F" puede volver a estar en estado "U" después de haber completado una tarea. Ya que un nodo en estado "U" aceptará tareas de otros nodos sobrecargados, es probable que pase a estado "F". Por otro lado, un nodo en estado "F", solo acepta sus propias tareas y es probable que vuelva a pasar al estado "U". De este modo, cada nodo en el sistema cambiaría frecuentemente entre los estados "U" y "F", incrementando, de este modo, el tráfico de la red debido a la difusión de estos repetitivos cambios de estado. Los cambios de estado ocurrirán de una manera infrecuente cuando la diferencia entre THu y THf sea grande, y por lo tanto, los cambios de estado ocurrirán de manera frecuente si la diferencia entre THu y THF es pequeña.

La tercera cota, THv, es usada para evitar una innecesaria transferencia de tareas. Si combinamos THf y THv en una cota, entonces la aceptación de una tarea transferida puede generar un nodo con estado "F" (Full) y "V" (Over) al mismo tiempo. En este caso dicho nodo deberá transferir sus propias tareas recién llegadas, y así, una de las dos tareas de transferencia no hubiera sido necesaria. Introduciendo la cota THv, cada nodo difundirá el cambio de estado cuando este pase a estado "F", previniendo que otros nodos transfieran sus tareas a este nodo. La diferencia existente entre THf y THv es usada para controlar tareas de transferencia innecesarias.

7.1.4. La lista de nodos preferidos

El propósito de construir una lista de nodos preferidos en cada nodo, es para evitar la sobrecarga causada por la evaluación de los nodos para poder obtener uno infrautilizado en donde volcar las tareas excedentes. Como el costo de transferir una tarea se incrementará en función de la distancia física entre los nodos emisores y receptores, entonces el nodo receptor debería estar ubicado lo más cerca posible del nodo emisor. La lista de nodos preferidos en cada nodo se construirá en base al número de saltos entre los nodos emisores y receptores.

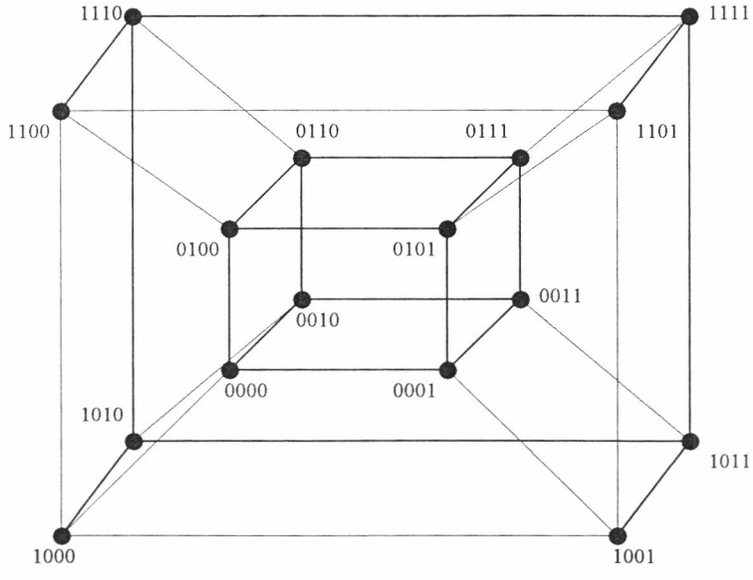
Las primeras posiciones en la lista de nodos preferidos serán entonces asignadas a sus vecinos inmediatos, y las posiciones siguientes serán para aquellos nodos a "dos saltos" de distancia del nodo y así sucesivamente. Cuando haya más de un nodo con la misma cantidad de saltos del nodo origen, estos nodos deberán ser ordenados de manera tal que permitan minimizar el problema de la transferencia concurrente de tareas hacia un mismo nodo.

Para demostrar como ordenar los nodos de cada conjunto de nodos asociados basados en la topología del sistema, tomaremos como ejemplo un sistema con n nodos N1, N2,..., Nn, donde el grado de Ni es k para todo i. El vínculo j de Ni es asignado a la dirección dj, $0 <= j <= k-1$. La lista de nodos preferidos de Ni es entonces construida de la siguiente forma: el conjunto de vecinos inmediatos de Ni, denotados Pi(1) serán ubicados en primer lugar en la lista de nodos preferidos. Los nodos vecinos que se encuentran "a dos pasos" de Ni, denotados por Pi(2), son los nodos que se encuentran en la primera ubicación de cada nodo en Pi(1), excluyendo aquellos nodos duplicados.

En general, podemos decir que Pi(m) es el conjunto de nodos que se encuentran en la primera ubicación de cada nodo en Pi(m-1), excluyendo los nodos duplicados.

Entre los nodos en Pi(1), el nodo en la dirección d0 será el elegido para ser el nodo *más preferido* de Ni, denotado por N1(i1), y el nodo en dirección d1 será el segundo nodo más preferido, denotado como N2(i1) y así sucesivamente. Los nodos en Pi(2) son ordenados del siguiente modo: Los nodos en la primera ubicación de N1(i1) son examinados de acuerdo a su orden en la lista. Si un nodo en la primera ubicación de N1(i1) no aparecen en ninguna ubicación anterior de Ni, entonces será copiado en la segunda ubicación de Ni con el mismo orden que tenían en la primera entrada de N1(i1). Después que todos los nodos en la primera ubicación de N1(i1) han sido chequeados y copiados, los nodos en la primera ubicación del nodo N2(i1) serán chequeados y copiados mediante el mismo procedimiento. Este procedimiento se repetirá hasta que Pi(2) se haya completado. El ordenamiento de los nodos en Pi(m), para todo m>2, será definido de la misma manera.

Como un ejemplo, consideremos la construcción de la lista de nodos preferidos en cada nodo de un sistema hipercubo de grado 4 (dimensión 4). La identidad (ID) de cada nodo es codificada con un número de 4 bits (b3, b2,b1,b0). La dirección d0 de Nk es el vínculo que conecta Nk a un nodo cuyo ID difiere del ID de Nk en la posición i, con 0 <= i <= 3. Podemos ahora aplicar el procedimiento que detallamos anteriormente para construir la lista de nodos preferidos para cada nodo en un sistema hipercubo como lo muestra la figura 1.



Orden de preferencia	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Nodo 0	1	2	4	8	6	10	12	3	5	9	14	13	11	7	15
Nodo 1	0	3	5	9	7	11	13	2	4	8	15	12	10	6	14
Nodo 2	3	0	6	10	4	8	14	1	7	11	12	15	9	5	13
Nodo 3	2	1	7	11	5	9	15	0	6	10	13	14	8	4	12
Nodo 4	5	6	0	12	2	14	8	7	1	13	10	9	15	3	11
Nodo 5	4	7	1	13	3	15	9	6	0	12	11	8	14	2	10
Nodo 6	7	4	2	14	0	12	10	5	3	15	8	11	13	1	9
Nodo 7	6	5	3	15	1	13	11	4	2	14	9	10	12	0	8
Nodo 8	9	10	12	0	14	2	4	11	13	1	6	5	3	15	7
Nodo 9	8	11	13	1	15	3	5	10	12	0	7	4	2	14	6
Nodo 10	11	8	14	2	12	0	6	9	15	3	4	7	1	13	5
Nodo 11	10	9	15	3	13	1	7	8	14	2	5	6	0	12	4
Nodo 12	13	14	8	4	10	6	0	15	9	5	2	1	7	11	3
Nodo 13	12	15	9	5	11	7	1	14	8	4	3	0	6	10	2
Nodo 14	15	12	10	6	8	4	2	13	11	7	0	3	5	9	1
Nodo 15	14	13	11	7	9	5	3	12	10	6	1	2	4	8	0

Figura 1: Ejemplo de lista de nodos preferidos de un sistema con topología de hipercubo.

Una vez que la lista de preferidos de cada nodo ha sido construida, un nodo sobrecargado Ni podrá seleccionar un nodo desocupado de la siguiente manera: Examinar N1(i1) primero; si este esta desocupado, Ni transferirá una tarea a N1(i1); de otro modo, N2(i1) será chequeado, y así sucesivamente. (Este chequeo puede fácilmente ser implementado con un puntero, el cual apunta al primer nodo disponible de la lista). Si todos los nodos en Pi(1) están sobrecargados, Ni examinará secuencialmente los nodos en Pi(2). Si un nodo sobrecargado no encuentra ningún nodo infrautilizado en su lista de nodos preferidos, todas sus tareas se verán forzadas a ser ejecutadas localmente.

La lista de nodos preferidos construida anteriormente posee las siguientes ventajas. Primero, ya que cada nodo es el nodo más preferido de uno y solo un nodo en el sistema, la probabilidad de que un nodo infrautilizado sea seleccionado por más de un nodo sobrecargado es muy pequeña. Segundo, el costo de transferencia de la tarea es mínimo, ya que un nodo receptor es seleccionado con una alta probabilidad de que sea un nodo fisicamente vecino del nodo emisor. Además, la sobrecarga de trabajo para la selección de un nodo desocupado es insignificante, ya que el tiempo utilizado por los procedimientos de chequeo de estados de los nodos por los mecanismos convencionales no es necesario.

Sin embargo, un conjunto de nodos asociados no debe ser demasiado grande, porque cuanto más grande sea el tamaño del conjunto de nodos asociados, más alta será la sobrecarga de tráfico generada por la difusión de los cambios de estados. Por lo tanto, deberíamos logra un equilibrio entre la capacidad de mantener la distribución de la carga que abarque todo el sistema y la sobrecarga de tráfico de la red causada por los cambios de estado.

7.2. Distribución de la Carga: Detalles de la Implementación

Esta implementación ha sido desarrollada sobre un sistema Linux con arquitectura Intel utilizando el lenguaje C++ y con el compilador Kylix.

Nuestro desarrollo, al haberse realizado en el nivel de usuario, no ha requerido modificación alguna al kernel del sistema operativo. Esta característica dota a la propuesta desarrollada de una alta flexibilidad y facilidad de implementación sobre sistemas operativos existentes.

Si bien la implementación se ha realizado sobre Linux, este desarrollo podría ser fácilmente implementado en otros sistemas operativos "estilo Unix".

La implementación del mecanismo de distribución de la carga del sistema, consta de 2 aplicaciones (un ejecutable "lanzador" de procesos y un demonio), y un archivo de configuración por nodo.

Todo este mecanismo de distribución de carga fue pensado para no necesitar modificaciones sobre el kernel de un sistema operativo, por lo tanto, para que un proceso pueda ser tenido en cuenta dentro esquema de distribución de carga, deberá ser lanzado por un programa que se encargará de registrarlo dentro de la lista de procesos de distribución de carga, ya que sino el proceso de distribución de carga no podrá ser capaz de registrar información de los procesos que van a participar del mismo. Este programa, llamado "setloadbalanced", es un ejecutable que será invocado por el usuario de una terminal y se le indicará como parámetro el nombre del programa que se desee que participe del proceso de distribución de carga. Una vez registrado un proceso dentro de la lista, este se verá involucrado en todo el mecanismo de distribución de carga, pudiendo ser migrado a otro nodo del sistema sin que el usuario se entere de esa situación.

Para la implementación de todo el mecanismo de distribución de la carga, se pensó en una aplicación demonio que pudiese correr en background en cada uno de los nodos. La función del mismo será la de mantener la información asociada a la distribución de la carga dinámica en cada nodo, y llevar adelante la aplicación de las reglas de locación, distribución y selección. Ver figura 2.

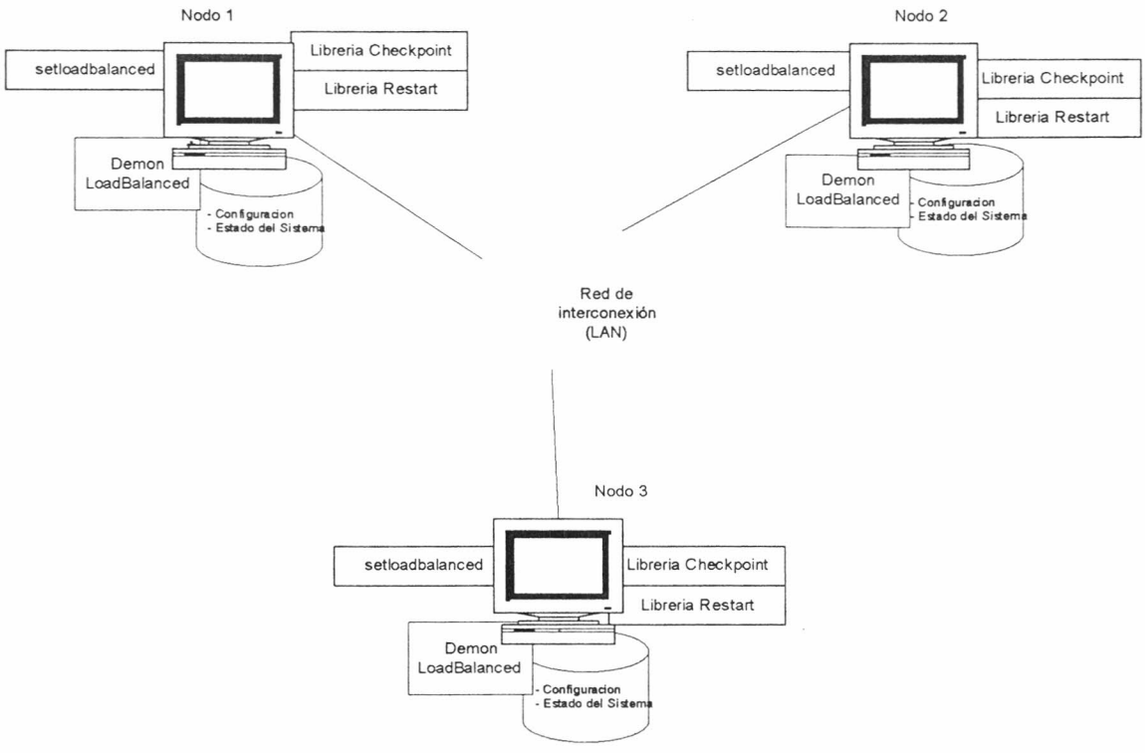


Figura 2: Esquema general de los componentes participes de la distribución de carga propuesta, sobre un sistema distribuido.

Cada nodo poseerá un demonio de distribución de carga propio llamado "LoadBalanced", que si bien es el mismo, este se configurará de manera local en cada máquina para que sus reglas de tomas de decisiones se encuentren definidas específicamente según la capacidad y los recursos disponibles en cada nodo de la red.

El demonio LoadBalanced será responsable de:

- Manejar la lista de procesos que participan del control de distribución dentro de cada nodo del sistema.
- Llevar el control del estado de carga del nodo, verificando la longitud de la cola de procesos dentro de las cotas preestablecida en su configuración.
- Leer y fijar los datos de configuración local del demonio en cada nodo.
- Selección del proceso a migrar cuando el nodo se encuentre sobrecargado.
- Invocar la restauración de un proceso una vez que este ha sido migrado, proporcionando al proceso de restauración todos los datos que este necesita para su correcto funcionamiento.
- Eliminar el proceso *shadow* del *nodo origen*, cuando el proceso

migrado termina su ejecución.

- Verificar que un proceso se encuentre activo, eliminándolo de la lista de procesos en caso contrario.

- Manejo de las comunicaciones entre los diferentes nodos del sistema.

- Transmisión de los cambios de estados de carga del nodo al conjunto de nodos asociados.

- Transmisión y recepción del espacio de direcciones del proceso migrado (en nuestra implementación, *archivo de checkpoint*).

Para el correcto desempeño del servicio de distribución de la carga, se hace imprescindible la configuración del demonio loadbalanced sobre cada nodo participe dentro del sistema distribuido, para ello, este necesitará un archivo local de configuración que será personalizado para cada nodo y de donde se obtendrán los parámetros que influirán tanto en su comportamiento como en la toma de decisiones.

Los datos almacenados sobre este archivo de configuración son los siguientes:

- Lista de nodo asociados.
- Cota THu
- Cota THf
- Cota THv
- Puerto de conexión.
- Frecuencia de revisión de procesos activos.
- Programa de restauración de procesos
- Programa shadow
- Librería de restauración de procesos
- Librería de checkpoint
- Tipo de selección del proceso a migrar

7.2.1. Transmisión de datos para la distribución de la carga

Para la transmisión de la información utilizamos el protocolo TCP/IP debido a que este es el protocolo estándar de red por excelencia y además por ser el protocolo natural y de uso mas frecuente en ambientes UNIX para la transmisión de datos.

Las características que ofrece TCP/IP, se adecuan perfectamente a nuestras necesidades de comunicación:

- Comunicación segura mediante conexiones punto a punto.
- Inicialmente todos los nodos son iguales.
- No existe la necesidad de contar con nodos servidores "especiales"

Como todo desarrollo sobre TCP/IP, la comunicación se desarrolla bajo el esquema del modelo Cliente-Servidor, utilizando conexiones sockets para la lectura y emisión de los datos a transmitir. En una comunicación de este tipo, un nodo asume el rol de cliente y otro nodo debe asumir el rol de servidor. Como la comunicación entre los distintos nodos depende del tipo de actividad y estado que se encuentra ejecutando el demonio loadbalanced, a veces un mismo nodo puede cumplir el rol de cliente en la comunicación y a veces puede ser servidor de la comunicación.

El esquema de comunicación de mensajes entre conexiones sockets se puede observar en la figura 3.

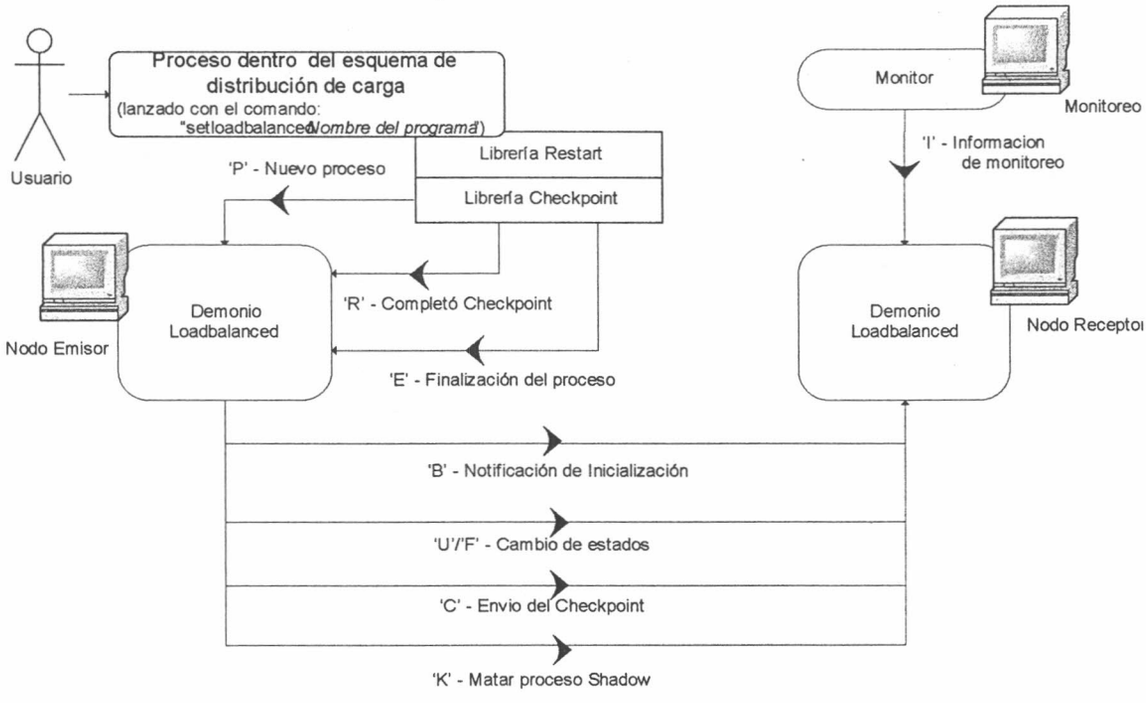


Figura 3: Esquema de transmisión de mensajes utilizado por el demonio "loadbalanced"entre conexiones sockets

7.2.2. La aplicación SetLoadBalanced por dentro

Como expusimos anteriormente, la aplicación SetLoadBalanced es un ejecutable que permite lanzar una aplicación, de una manera comoda, para que esta última pueda ser participe del mecanismo de distribución de la carga, fijando los parámetros que un proceso necesita para incorporar su información a la lista de procesos manejados por el demonio *loadbalanced*.

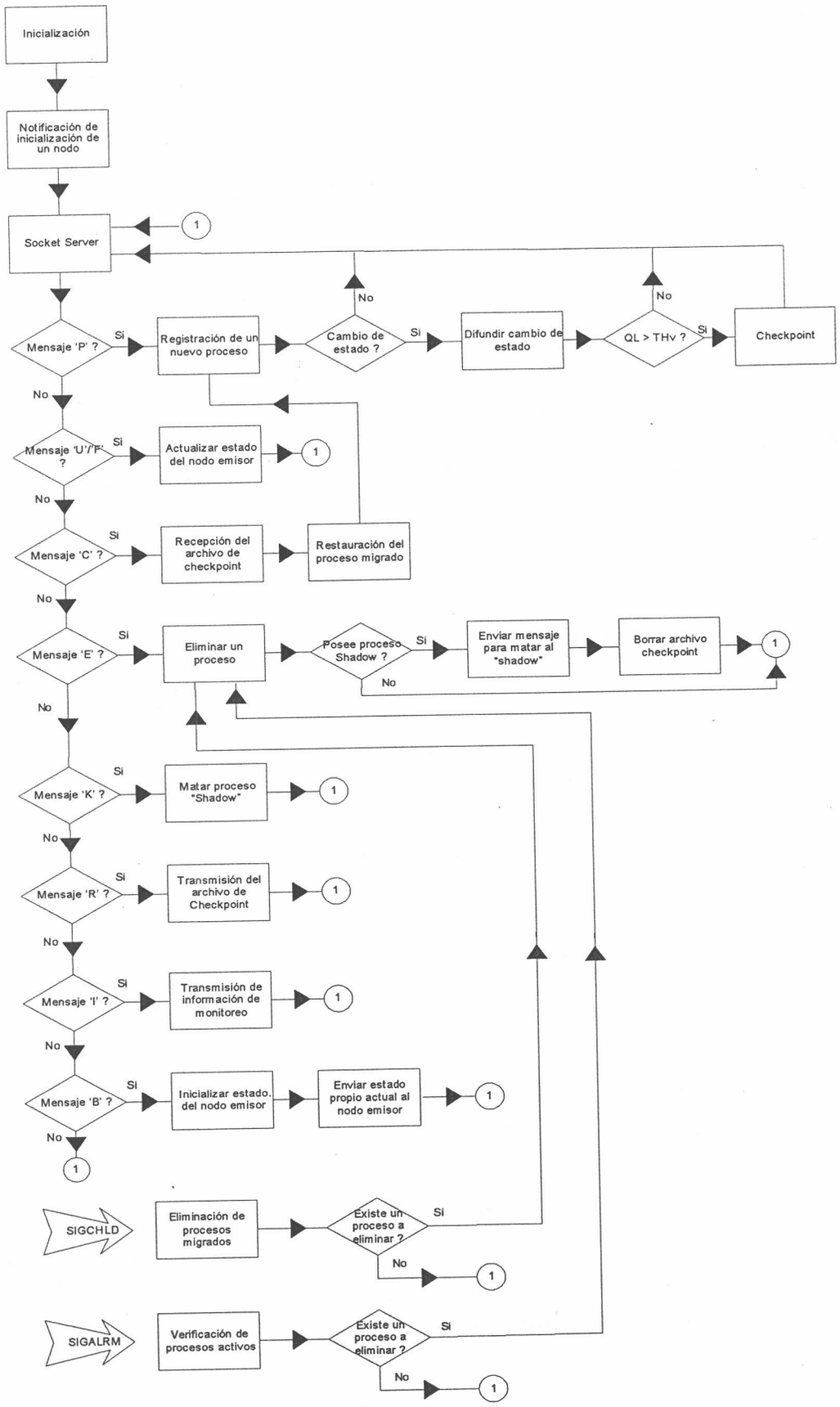
El programa setloadbalanced es invocado pasándole como parámetro el nombre del programa a incluir dentro del mecanismo de distribución de la carga:

Setloadbalanced "Nombre_del_programa"

En su ejecución, la aplicación lee del archivo de configuración loadbalanced.cfg la ubicación de la librería de checkpoint, el port donde conectarse con el demonio loadbalanced y el programa shadow (syscall_rpc_svc). Una vez leídos estos datos, el setloadbalanced invoca al programa solicitado utilizando un fork-execve y pasando los parámetros de entorno que la librería de checkpoint necesita para su ejecución. (La llamada execve permite la ejecución de un programa especificando el nombre del mismo, los parámetros de invocación y sus propias variables de entorno). Una vez lanzada la aplicación solicitada, esta va a cargar la librería de checkpoint y va a solicitar su registración al demonio loadbalanced enviándole un mensaje de tipo 'P'. El demonio loadbalanced, al recibir un mensaje de tipo 'P', incluirá al proceso dentro de la lista de procesos participes del mecanismo de equilibrio de carga.

7.2.3. El demonio Loadbalanced por dentro

Flowchart del demonio *Loadbalanced*



7.2.4. Detalle de cada una de las tareas funcionales del demonio "LoadBalanced"

- Inicialización:

  - "Demonizar" la aplicación para que trabaje en background como si fuese un servicio del sistema.
    Se invoca un fork y se finaliza al proceso padre, de este modo creamos un proceso hijo huérfano, que será entonces hijo del proceso INIT (el padre de todos los procesos) y comenzara de este modo a operar en background. Luego se independiza de cualquier grupo de procesos y se cierran todos los descriptores de archivos heredados del padre. También se fija una nueva máscara de permisos de creación de archivos y el directorio de ejecución. Se establece un archivo de exclusión mutua para que solo una instancia del demonio se ejecute en la máquina. Finalmente, se ignoran las señales SIGTSTP (señal Ctrl-Z), SIGTTOU (señal de escritura del terminal para un proceso en 2° plano) y SIGTTIN (señal de lectura del terminal para un proceso en 2° plano), y se atrapan las señales SIGCHLD (señal de finalización de un hijo), SIGHUP (señal de suspensión) y SIGTERM (señal de finalización normal).

  - Lectura de los parámetros de configuración del demonio.
    Para la lectura de los parámetros se abre el archivo de configuración llamado "loadbalanced.cfg", el cual se deberá encontrar en el directorio "/etc" (directorio que habitualmente se lo utiliza en Linux para depositar los archivos de configuración de las aplicaciones), y posteriormente se procede a la lectura de la información allí almacenada, registrándola en variables globales que serán utilizadas a lo largo del programa como referentes de comportamientos y tomas de decisiones. El archivo de configuración es un archivo de texto, por lo que el proceso de lectura del mismo se efectúa simplemente a través de un recorrido secuencial por el mismo, verificando en cada línea el tipo de información que esta posee.

  - Notificar a los nodos asociados acerca de su inicialización.
    Recorre la lista de nodos asociados y le envía un mensaje de tipo 'B' (notificación de inicio) informándole a cada uno de los nodos asociados que se inicializo el servicio de balanceo de carga en el nodo que esta emitiendo el mensaje y obteniendo como respuesta el estado actual de cada uno de los nodos que recibieron el mensaje.

  - Establecer un servidor socket para la recepción de datos provenientes de otros nodos.
    Se crea y se le da el control del programa a un servidor socket, el cual una vez creado, se mantendrá a la espera de mensajes que solicitarán la ejecución de las distintas tareas/responsabilidades del demonio de distribución de carga del nodo.

- Notificación de inicialización de un nodo

Cuando el demonio loadbalanced de un nodo se inicializa, envía un mensaje de tipo 'B' a cada uno de sus nodos asociados. Al recibir esta notificación de inicialización, el demonio receptor registra con estado 'U' al nodo emisor y le envía su propio estado, para que el nodo emisor que esta inicializándose registre en su tabla de nodos asociados el estado actual de cada uno de ellos.

- Socket Server

  - Creación de un socket de recepción:
    Se crea un socket, estableciendo un puerto de lectura cuyo número estará determinado en el archivo de configuración, desde donde se van a recepcionar las solicitudes emitidas por otros nodos y/o aplicaciones que desean enviar u obtener información.

  - Mantenerse en espera de la recepción de datos provenientes de otros nodos.
    Una vez creado el socket, el mismo se queda a la espera de la llegada de un pedido de conexión, empleando la técnica de "Select" que mantiene en estado de espera la aplicación hasta que se detecta actividad en el socket. Cuando llega un pedido, se crea la conexión y se la registra en un vector de descriptores de socket, permitiendo de este modo que el demonio "loadbalanced" pueda atender más de una conexión, sin necesidad de crear nuevos procesos de atención de sockets ante cada nueva conexión solicitada.

  - Invocar acciones de acuerdo al tipo de mensaje recibido.
    Cuando un mensaje es recibido, se chequea el primer byte/caracter de la transmisión, que nos permitirá conocer el tipo de mensaje que se ha pretendido transmitir.
    Los valores que puede tener el primer byte/caracter de transmisión de un mensaje son los siguientes:

    - 'P': Lanzamiento de un proceso
    - 'U'/'F': Cambios de estados ('U' (Under): Infrautilizado - 'F'(Full): Ocupado)
    - 'C': Recepción del checkpoint
    - 'R': Envío del checkpoint
    - 'K': Matar proceso *shadow*
    - 'I': Solicitud de información de monitoreo.
    - 'B': Notificación de inicialización de un nodo.

    Cada uno de estos mensajes tienen asociados acciones o procesos que el demonio "Loadbalanced" deberá atender.

- Registración de un nuevo proceso local

  Al leerse un mensaje de tipo 'P' (Lanzamiento de un proceso), se lee a continuación el PID del proceso que se lanzó. Utilizando el PID leído, se lo registra en la lista de procesos que participan del mecanismo de distribución de carga.

Al incorporar el nuevo proceso a la lista de procesos del nodo, se debe verificar si se produjo un cambio de estado dentro del nodo, comparando las cotas con la longitud de la lista de procesos, y en caso de producirse un cambio de estado se deberá difundir el mismo al resto de los nodos asociados, enviando mensajes 'U'/'F' según corresponda al estado en que se encuentre el nodo. Si al difundir un cambio de estado no es posible efectuar la conexión con algunos de los nodos asociados, a este último se lo considerará "caído" y se lo inhabilitará como nodo asociado, asignándole al estado un valor indefinido (' ').
Además, si al incorporar el nuevo proceso a la lista de procesos, la longitud de la misma supera la cota THv, se deberá realizar un checkpoint sobre algún proceso seleccionado para ser migrado, según el tipo de método de selección definido en el archivo de configuración.

- Actualización de cambios de estados de otros nodos

  Al leerse un mensaje de tipo 'U' (Under) o 'F' (Full), significa que un nodo realizo un cambio de estado y se lo difundió al resto de sus nodos asociados, por lo tanto, al recibir este tipo de mensaje, se deberá actualizar el estado del nodo dentro de la lista de nodos asociados. Para ello, el demonio loadbalanced recorre la lista de nodos asociados hasta encontrar al nodo que envió el mensaje y le cambia el estado al mismo.

- Matar un proceso "*shadow*"

  Cuando se necesita matar al proceso *shadow* en la máquina donde originalmente se lanzó un proceso que fue migrado y que ahora terminó, se envía un mensaje de tipo 'K' con el PID del *shadow* que se desea matar. El demonio receptor del mensaje, desregistra el servidor RPC asociado al proceso utilizando el comando *rpcinfo –d* y luego le enviará una señal SIGTERM al proceso *shadow* correspondiente al PID referenciado.

- Recepción y restauración de un proceso con checkpoint

  - Obtención de los datos del checkpoint
    Cuando el servidor socket recibe un mensaje de tipo 'C' (Envío del checkpoint), inmediatamente después se lee el nombre del programa ejecutable migrado, el nodo donde se encuentra el proceso *shadow* asociado al proceso migrado, el PID del *shadow* y el archivo de checkpoint.
    Para leer el archivo, primero se recibe el nombre del archivo, luego el tamaño del mismo y finalmente el archivo completo, utilizando para esto último, un ciclo de lecturas en buffers de lo que le llega a la conexión del socket hasta completar el tamaño del archivo.
    Si el proceso migrado es la primera vez que lo hace, el nodo donde reside el *shadow* será el nodo que transfirió el archivo de checkpoint, por lo que se registrará como dirección IP del shadow la dirección IP de la conexión socket que se utilizó para la migración.

- Restauración del proceso migrado
  Una vez recibido el archivo de checkpoint, se procede a la restauración del proceso migrado, invocando al programa restart, pasándole como parámetro el nombre del archivo de checkpoint recibido.
  Para registrar este proceso recién migrado a la lista de procesos del nodo, obtenemos el PID del proceso recién lanzado, el cual, junto con la dirección del nodo que posee el *shadow* del proceso y el PID del *shadow* serán los datos de un nuevo ítem de la lista de procesos del nodo.

- Transmisión del archivo de checkpoint

  El mecanismo de checkpoint genera un archivo (que llamaremos archivo de checkpoint) y el proceso seleccionado para la migración es transformado en un proceso *shadow* que mantendrá el estado externo del proceso atendiendo los requerimientos de entrada/salida del proceso migrado.
  Para completar la migración de un proceso se debe transmitir el archivo de checkpoint generado y eliminar el proceso de la lista de procesos del nodo.
  Una vez que el mecanismo de checkpoint completo su tarea, envía un mensaje de tipo 'R' al demonio loadbalanced para informarle a este que puede completar la migración del proceso.
  Para enviar el archivo de checkpoint primero se establece cual es el nodo *más preferido* de la lista de nodos asociados, esto se realiza efectuando un recorrido secuencial por la lista de nodos asociados hasta obtener el primer nodo con estado en 'U'. Luego, se establece una conexión socket con el nodo seleccionado para recibir el proceso a migrar y se le envía un mensaje de tipo 'C' donde se transferirá el nombre de la aplicación migrada, la identificación del nodo donde se encuentra el proceso *shadow* del proceso a migrar, el PID del proceso *shadow*, el nombre del archivo de checkpoint, la longitud del archivo de checkpoint y por último los datos del archivo de checkpoint, recibiendo como respuesta el estado actual del nodo receptor del mensaje una vez que este incluyó el proceso migrado en su lista de procesos a administrar.
  Si la transmisión del checkpoint resultó exitosa, se elimina el proceso de la lista de procesos del nodo original, de lo contrario, se vuelve atrás con el checkpoint realizado, restituyendo el proceso en el estado en que se encontraba antes de su intento de migración.

- Envío de datos de monitoreo

  Para monitorear los datos y el comportamiento de un demonio, se implementó un mensaje de tipo 'I' que obliga al demonio a enviar información propia a través de la conexión socket que solicitó el pedido.

- Chequear estado local del sistema

  Cada vez que un nuevo proceso es insertado y/o eliminado de la lista de procesos, se deberá verificar la longitud de la lista, ya que la misma nos

permitirá conocer cual es la carga de trabajo del nodo, estableciendo un estado de carga en el nodo.
La relación entre la longitud de la lista de procesos y el estado del mismo se determina de acuerdo con las cotas establecidas (y previamente leídas) en el archivo de configuración del demonio loadbalanced.
El siguiente cuadro muestra cual es la relación entre la longitud de la lista de procesos y el estado de carga de un nodo:

Longitud de la lista de procesos (QL)	Estado
QL <= THu	'U' (Under)
THf < QL <= THv	'F' (Full)
THv < QL	'V' (Over)

Un detalle de nuestra implementación es lo siguiente: ya que cuando un nodo esta en estado 'F' o en estado 'V' no puede recibir procesos externos, no es necesario informar al conjunto de nodos asociados que un nodo se encuentra en estado 'V', ya que simplemente con estar en estado 'F' alcanza para no permitir que ese nodo reciba procesos migrados de otros nodos.

- Checkpoint
  Cuando la carga de un nodo (medida por la longitud de su lista de procesos) supera la cota THv, se hace necesaria la migración de un proceso. Para ello se invoca el mecanismo de checkpoint.
  Si existen nodos infrautilizados (estado 'U') hacia quien enviar algún proceso propio, el demonio loadbalanced elegirá un proceso victima de la migración de acuerdo al método de selección establecido en el archivo de configuración (menor tiempo de consumo de procesador, mayor tiempo de consumo de procesador, menor consumo de memoria o mayor consumo de memoria).
  El checkpoint sobre un proceso es iniciado enviándole al mismo la señal SIGTSTP.
  Como resultado de la ejecución del checkpoint, se genera un archivo que contiene datos del estado interno del proceso a migrar, además, transforma el proceso migrado en un proceso *shadow* que mantendrá el estado externo del proceso, interactuando con el proceso migrado y atendiendo solicitudes de llamadas al sistema (system calls) de este último.

- Eliminación de procesos migrados
  (Funcionalidad asociada a la señal SIGCHLD)

  Cuando un proceso es migrado, es lanzado en el nodo remoto por el demonio loadbalanced, lo cual lo convierte a ese proceso en hijo del loadbalanced. Por lo tanto, cuando un proceso migrado finaliza, el demonio recibe una señal SIGCHLD que le permite reconocer esa situación y eliminar ese proceso de la lista de procesos propios, enviando además un mensaje de solicitud de muerte del proceso *shadow* asociado (mensaje de tipo 'K'), al nodo origen del proceso que acaba de terminar.

- Verificación de procesos activos
  (Funcionalidad asociada a la señal SIGALRM)

  El mecanismo de distribución de carga, requiere de un mecanismo encargado de verificar si algún proceso, dentro de la lista de procesos del nodo y que nunca ha sido migrado ha finalizado.
  Para ello se establece una alarma dentro del demonio que, con una frecuencia definida en el archivo de configuración del demonio loadbalanced, consultará el directorio /proc del sistema, verificando la existencia de cada proceso registrado en la lista de procesos del nodo. Si no se encuentra la identificación de un proceso (PID) en el directorio /proc, se solicitará inmediatamente la eliminación del proceso de la lista de procesos del nodo.
  Junto con la verificación de procesos activos, se invoca una reexaminación del aquellos nodos que se encontraban en inactivos para ver si se han activado o no.

- Finalización de un proceso

  Cuando un proceso termina su ejecución sobre un nodo, ya sea por haber sido migrado o porque finalizó de manera normal o abrupta, se lo debe eliminar de la lista de procesos del nodo. Si además el proceso recientemente eliminado había sido migrado con anterioridad, se debe enviar un mensaje de tipo 'K' (matar al *shadow*) al nodo origen del proceso, el cual deberá poseer el *shadow* asociado al proceso que acaba de finalizar, y también se borra el archivo de checkpoint transferido durante la migración del proceso, para que no queden archivos "basura" dentro del sistema.

7.3. Migración de Procesos

7.3.1. Introducción

Checkpoint de procesos es un mecanismo básico para proveer la infraestructura necesaria para efectuar la migración de un proceso. Si bien existen mecanismos de migración de procesos que no realizan checkpoint, nosotros hemos optado por implementar la migración de procesos a través de checkpointing, debido a que esto nos proveía una mayor independencia entre el mecanismo de distribución de carga, y el mecanismo de migración. El mecanismo de checkpoint y migración implementado, está basado en el utilizado por el sistema Condor [ref. 2].

A diferencia de la implementación realizada para el sistema Condor, en donde se privilegia tanto la tolerancia a fallos, como la distribución de la carga, en nuestra implementación sólo estamos interesados en la distribución de la carga, lo cual nos permitió realizar una serie de modificaciones a la implementación original del mecanismo utilizado por Condor, para ganar en transparencia y versatilidad.

Para hacer un checkpoint , y reiniciar un proceso es necesario considerar todos los componentes que hacen al estado de un proceso. Podemos dividir el estado de un proceso, entre su **estado interno** (dividido en código, datos, stack, y librerías dinámicas cargadas, así como el contenido de los registros) y su **estado externo** (manejadores de señales, descriptores de archivos abiertos)

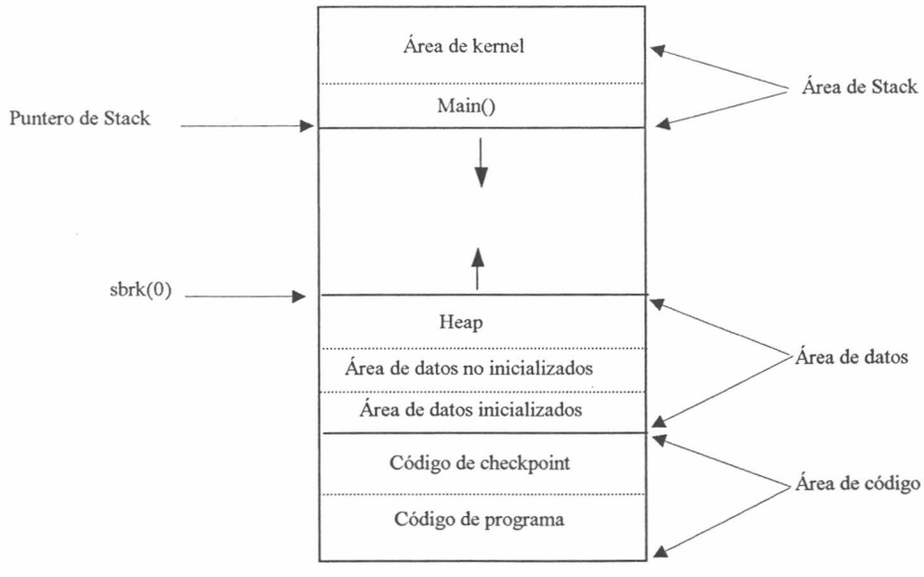


Fig. 4: Estructura interna del proceso en memoria.

7.3.2. Implementación de checkpoint y migración de procesos en el sistema Condor

Preservación del estado interno

Área de código y área de datos

Los procesos nacen con su área de código completamente cargada en su memoria virtual, y comienza generalmente en la posición 0, con lo cual no necesariamente tiene que ser salvada (estamos dando por supuesto que el área de texto no cambia).

El área de datos se puede dividir en área de datos inicializada, no inicializada y el heap. El área de datos inicializada son aquellos datos asignados por el programador en tiempo de compilación, la no inicializada, es espacio de memoria reservado, pero no inicializado, y el heap es el área de datos asignada en tiempo de corrida por llamadas al sistema como brk() y sbrk(), utilizadas habitualmente a través de la función de biblioteca standard malloc().

Entonces, resumiendo, el área de datos inicializada es la siguiente al área de código, la no inicializada le sigue le sigue a ésta, y a ésta el heap. Por lo tanto para salvar el área de datos, solo es necesario conocer su comienzo (fin del área de código), y su fin (que está dado por la llamada a sbrk(0)).

Stack

El área de stack, es la parte del espacio de direcciones asignada en tiempo de ejecución en donde se almacena la información que necesita el mecanismo de llamadas a procedimientos, los argumentos de las llamadas a procedimientos, etc.

El tamaño del stack varía en tiempo de ejecución, con cada llamada a un procedimiento. En Linux, el stack comienza en una dirección fija y crece hacia direcciones mas bajas.

Preservar el estado del stack requiere salvar dos piezas de información, el stack context (que contiene el stack pointer) y el contenido del stack propiamente dicho.

Para salvar (y restablecer) el stack context, Condor utiliza las llamadas al sistema setjmp() y longjmp(). Cuando el proceso recibe la señal (usualmente SIGTSTP) que iniciará el proceso de checkpoint, el manejador que atiende señal, lo primero que hará es hacer una llamada a setjmp() con una variable de tipo JMP_BUF como parámetro, en donde se almacenará el stack context y devolverá 0, para luego, cuando el proceso es restablecido, una llamada a longjmp con la misma variable (habiendo restablecido con anterioridad el área de datos) restablecerá el stack context , y, por lo tanto, la ejecución del proceso continuará en donde había sido interrumpida. Para hacer todo esto, el único cuidado que se tiene es que, al reiniciar proceso, este tendrá su propia área de stack, con lo cual, mientras estamos restableciendo, no podemos sobrescribir dicha área. Para salvar esta dificultad, lo que se hace es ejecutar las funciones de reestablecimiento del proceso en un nuevo stack dentro del área de datos

(nuevamente utilizando setjmp/longjmp y modificando el stack pointer de JMP_BUF entre la llamada a setjmp y longjmp).

Contenido de los registros

Debido a que el proceso de checkpoint es invocado a través de señales enviadas al proceso, cuando el manejador de esta señal retorna, reestablece el estado de los registros. Por lo tanto el mecanismo de manejo de señales provee todo lo necesario para salvar y reestablecer el estado de la CPU.

Preservación del estado externo

Archivos abiertos

Debido a que en el sistema Condor, los procesos fueron vinculados con las librerías de dicho sistema, y todas las llamadas al sistema relacionadas al manejo de archivos (open, close, write, etc) fueron rescritas en dichas librerías (técnica denominada *interposición*) para guardar toda la información relacionada con los archivos abiertos (el modo en que fueron abiertos, su posición, etc.), al momento de reestablecer el proceso, se cuenta con toda la información para reabrir dichos archivos y reposicionarlos.

Por ejemplo, las librerías del sistema Condor, implementan su propia versión de de la llamada al sistema open(), la cual guarda un registro de todos los archivos abiertos, así como también su modo de apertura, y luego invoca a syscall(SYS_OPEN,...,), para permitirle al proceso continuar con su tarea. Esta interposición se realiza solamente a nivel de llamadas al sistema (system calls), ya que cualquier otra llamada a la librería "C" standard (por ejemplo en el caso de write(), podría ser ptrintf()), invocará a la llamada al sistema interpuesta.

Señales

El conjunto de señales bloqueadas, es obtenido con sigprocmask(), el manejador de cada una de las señales es obtenido con sigaction(), y el conjunto de señales pendientes es obtenido con sigispending(), con lo cual se puede salvar con estas llamadas al sistema el estado, manejadores y señales pendientes.

Migración

La migración de procesos exige que un proceso pueda acceder al mismo conjunto de archivos abiertos de manera consistente. Mientras esta facilidad es provista en muchos ambientes a través de sistemas de archivos de red (como NFS) [ref. 17], el sistema Condor no utiliza estas facilidades, sino que lo hace a través de un proceso que queda en el *nodo origen*, al cual denomina "*shadow*", y que básicamente es un proceso

que se inicia luego del checkpoint y atiende (vía RPC) las llamadas al sistema remotas cuando el proceso original es migrado a otro nodo de la red. Como se indicó anteriormente, el estado de los archivos abiertos se restaurará por intermedio de llamadas al sistema en forma remota, con la información salvada en el archivo de checkpoint acerca del estado de los descriptores de archivo, su modo de apertura (que restablecerá a través de open()), posición (a través de lseek()), etc.

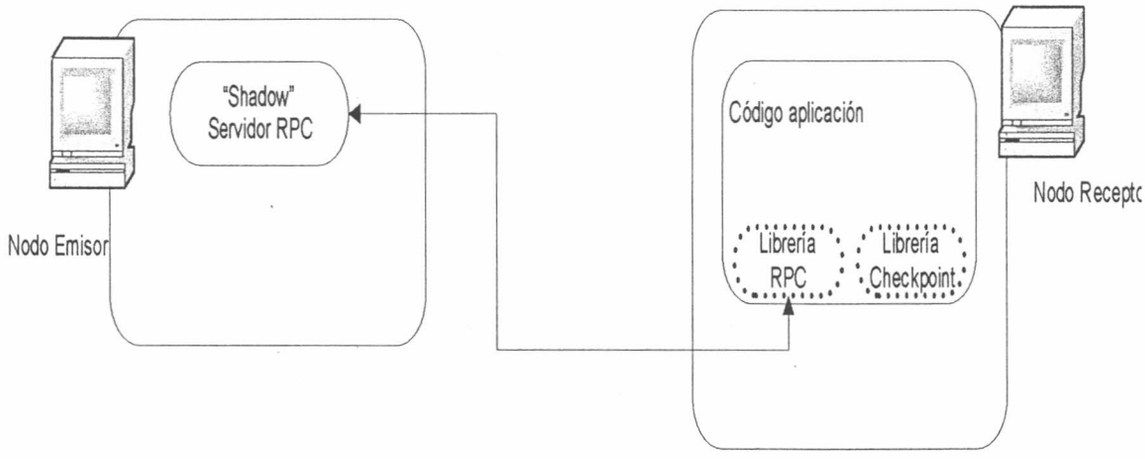


Figura 5: El estado externo del proceso migrado es reconstruido haciendo llamadas remotas al sistema, al proceso "*shadow*", que es ejecutado en la máquina origen.

Es claro que este mecanismo de checkpoint/migración tiene como limitaciones el hecho de no ser apropiado para migrar procesos que se intercomuniquen , procesos que utilicen las llamadas al sistema fork() y exec(), procesos multi-threads, procesos que utilicen descriptores de archivos sobre dispositivos que no soporten lseek(), o que tengan una semántica especial al realizar la operación close() (por ejemplo rewind o eject cuando se cierran). Por otro lado, otra importante limitación, es que el código que soporta el mecanismo de checkpoint/migración se debe vincular con el código de los programas de usuarios, cuando esto no siempre es posible.

7.4. Migración de Procesos: Nuestra implementación

Las modificaciones introducidas se pueden dividir en: aquellas relacionadas a como los programas de usuario se vinculan con las librerías de checkpoint/migración, como se realiza la interposición de llamadas al sistema y, por último, en como preservamos el estado, tanto interno como externo de un proceso. Cabe aclarar que debido a la implementación elegida, perdemos la cualidad de tolerancia a fallos que el sistema Condor posee.

7.4.1. Vinculación de librería de checkpoint/migración

Una manera directa, pero a la vez tediosa y compleja de vincular el programa de usuario con nuestras librerías de checkpoint/migración, sería hacer las modificaciones necesarias en la librería C estándar, recompilarla, e instalarla.

Para evitar lo anteriormente mencionado, pudimos valernos de que la mayoría de los sistemas modernos estilo-Unix (como Linux), soportan el concepto de precargar librerías definidas por el usuario (asignando el/los path correspondientes a la variable de ambiente LD_PRELOAD). Estas librerías pueden ser reemplazos completos (una librería "C" privada), o un subconjunto de ella (por ejemplo la llamada al sistema write()).

Entonces haciendo uso de este concepto, nosotros hacemos la interposición de las llamadas al sistema rescribiendo aquellas que nos interesan [ref. 18][ref. 21], agregándoles la funcionalidad que deseamos (en nuestro caso RPC al "*shadow*", que luego explicaremos, cuando hablemos de preservar el estado externo de un proceso), compilándolas en una librería dinámica, y asignando a LD_PRELOAD el path a nuestra librería. Cabe aclarar que esta variable es controlada por el linker/loader para objetos ELF. Este usa el primer símbolo que encuentre dentro de una librería que coincida con cualquier nombre simbólico dentro del programa de usuario, y por lo tanto al precargar nuestra librería, estos sobrescribirán cualquier otro símbolo definido en la librería "C" estándar.

Ahora bien, para realizar esto, nosotros tuvimos que vincular estáticamente nuestra librería con los objetos de la librería "C" estándar, que nosotros deseamos interponer. Esto se hace fácilmente extrayendo de la librería "C" dichos objetos (utilizando el comando ar –x write.o libc.a) y luego vinculamos estos objetos con nuestra librería. Luego de hacer esto, también tuvimos que tener el cuidado de que cuando nosotros necesitamos hacer llamadas al sistema, lo hicimos a través de la interface syscall(), para no entrar en un ciclo recursivo.

Por ejemplo, para interponer la llamada al sistema write(), lo hicimos de la siguiente manera :

```
ssize_t write(int fd, void const  *buf, ssize_t size)
{
if (!LocalExec)
//variable global que indica si estamos ejecutando luego de un checkpoint
        {
                // funcionalidad agregada para hacer llamadas remotas
                write_rpc_1(original_host,original_process_id,fd,buf,size);
        }
        else
        {
                //llamada local
                syscall(SYS_WRITE,fd,buf,size);
        }
}
```

Entonces, luego de hacer esto, cualquier llamada a write(), ya sea directa o indirectamente (por ejemplo a través de printf()), pasará por nuestro write().

Es claro que este modo de interponer llamadas funciona sólo con programas que realicen las llamadas al sistema a través de la librería "C" standard, y que ésta sea vinculada dinámicamente con dicho programa.

Por ejemplo, si el programa sobre el cual queremos hacer interposición, hiciera una llamada a write() a través de syscall(SYS_WRITE,...), este mecanismo no funcionaría. Luego, cuando hablemos de cómo preservamos el estado externo del proceso, explicaremos el significado de write_rpc y sus argumentos.

Ahora bien, con esto solucionamos el problema de la interposición de llamadas al sistema, pero todavía tenemos el problema de cómo hacer para instalar el manejador de señales (en nuestro caso SIGTSTP), que lanzará el mecanismo de checkpoint. El propio mecanismo de precargar librerías nos soluciona este problema, ya que en las librerías dinámicas se pueden definir dos funciones _init() y _fini(), que se ejecutarán cuando la librería se carga, y cuando se descarga, respectivamente. Entonces, definimos en la función _init(), la instalación del manejador de señales, y la inicialización de todas las estructuras que nos van servir para realizar el checkpoint, como podemos ver en el siguiente fragmento de código

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <dlfcn.h>
#include <syscall.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "checkpoint.h"
#include "syscall_svr.h"
```

```
extern struct shadow shadow_proc;
void write_rpc_1(char *host,char *buf, size_t nbytes);
void strcopy(char *ori, char *des, size_t len);

void _init() {
struct hostent *host;
struct in_addr **paddr;

      //Inicializa las estructuras para migración
      // preservando el process id y la dirección IP

      shadow_proc.pid=getpid();
      host=gethostbyname("localhost");
      paddr=(struct in_addr **)host->h_addr_list;
      sprintf(shadow_proc.addr,"%s",inet_ntoa(**paddr));
      shadow_proc.execution_mode=LOCAL_EXEC;
      shadow_proc.number_of_restarts=0;

      // instalando el manejador de señal para SIGTSTP

      if (SIG_ERR == signal(SIGTSTP, checkpoint_signal_handler))
      fprintf(stderr, "No puedo instalar el manejador señales \n");


}
```

La función checkpoint_signal_handler() será la encargada de asignar el punto de retorno (mediante setjmp()) para cuando el proceso sea reestablecido, crear el archivo de checkpoint, y ejecutar el "*shadow*", como veremos posteriormente.

7.4.2. Preservación del estado de un proceso

Estado interno

La preservación del estado interno de un proceso se hace básicamente del mismo modo en que lo hace Cóndor, utilizando la información del estado almacenada en los archivos /proc/self/maps (donde se encuentra la estructura de los segmentos de memoria utilizados por el proceso) y /proc/self/stat (donde encontramos la ubicación en memoria de los segmentos de datos y de código). Una vez obtenidos estos datos, lo que hacemos cuando realizamos el checkpoint, es básicamente, leer dichos segmentos de memoria (incluyendo librerías dinámicas), y salvarlos a un archivo que será nuestro archivo de checkpoint, el cual podremos utilizar luego para iniciarlo en otro nodo de la red.

Por ejemplo, si vemos para un proceso determinado su archivo /proc/self/maps, observaremos :

/cat /proc/self/maps

```
08048000-0804c000 r-xp 00000000 03:05 372494     /bin/cat
0804c000-0804d000 rw-p 00003000 03:05 372494     /bin/cat
0804d000-0804e000 rwxp 00000000 00:00 0
```

```
40000000-40012000 r-xp 00000000 03:05 64793      /lib/ld-2.2.93.so
40012000-40013000 rw-p 00012000 03:05 64793      /lib/ld-2.2.93.so
40024000-40025000 rw-p 00000000 00:00 0
40025000-40225000 r--p 00000000 03:05 469582
/usr/lib/locale/locale-archive
42000000-42126000 r-xp 00000000 03:05 518164     /lib/i686/libc-
2.2.93.so
42126000-4212b000 rw-p 00126000 03:05 518164     /lib/i686/libc-
2.2.93.so
4212b000-4212f000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

En el ejemplo presentado, podemos ver que las tres primeras regiones pertenecen al programa ejecutado (segmento de código, segmento de datos inicializados y segmento de datos no inicializados), las tres regiones siguientes corresponden al cargador de programas ld-2.2.93.so (nuevamente segmento de código, segmento de datos inicializados y segmento de datos no inicializados). Las regiones siguientes corresponden a la librería "C" compartida, y, finalmente, la última región pertenece al área de stack. Cabe aclarar que se salva también toda la información relacionada con los segmentos de memoria, como ser su modo de protección y sus direcciones de carga, de modo que cuando se realice el restablecimiento del proceso en un nuevo nodo se puedan levantar todos los segmentos en la misma dirección y con la misma protección.

Es claro que esta manera de salvar el estado interno de un proceso, nos obliga salvar no solamente el código y los datos asociados al programa original, sino también todas las librerías dinámicas enlazadas con este proceso, lo que tiene tanto ventajas como desventajas.

Como ventaja, podemos decir que nos independizamos del entorno de ejecución del proceso en el nodo destino, ya que no es necesario que las mismas versiones de las librerías dinámicas estén instaladas en él, y, en caso de que estas existan y sean las mismas versiones, como no podemos asegurar que estas sean cargadas en el mismo orden y en las mismas direcciones, no podríamos garantizar que el proceso se reiniciará correctamente.

Como desventaja, podemos señalar el tamaño desproporcionado que el archivo de checkpoint podría tener, así como también que este proceso quedaría "vinculado estáticamente", aunque originalmente no lo haya sido (esto es, existirán cargadas en memoria tantas copias de las librerías compartidas como procesos migrados haya, aunque estos compartan algunas de ellas). Esto implica un mayor consumo de memoria y tiempo de transferencia hacia el nodo destino.

Estado externo

Aquí, básicamente cambiamos el concepto "en donde salvamos" el estado externo de un proceso a migrar. Como habíamos visto anteriormente, el sistema Cóndor salva el estado externo del proceso en el propio archivo de checkpoint, haciendo un seguimiento de todas las llamadas al sistema que le interesan, y almacenando esa información en el propio proceso. Esta información será salvada cuando se haga el

checkpoint del área de datos, para luego, cuando el proceso se reinicia, dicha información será utilizada para recrear el estado de los archivos abiertos.

En nuestra implementación, lo que hacemos para salvar el estado externo del proceso es, básicamente no salvarlo, sino mantenerlo, ya que nunca matamos el proceso original, sino que reemplazamos su imagen por la del proceso *shadow*. Esto nos permite mantener el estado de los archivos abiertos (debido a que la llamada al sistema exec() preserva el estado de los descriptores de archivo).

Esta implementación nos permite salvar algunas de las falencias que tenía el desarrollo original, como ser manejar dispositivos que tuvieran una semántica especial de cerrado, o trabajar con descriptores de archivos que no soportaran lseek(), así como también abre la posibilidad de trabajar con procesos que se intercomuniquen por intermedio de sockets (o simplemente que se intercomuniquen, si tomamos a los sockets como una generalización de la comunicación entre procesos).

Por otro lado perdemos en tolerancia a fallos, ya que si llegara a suceder que debido a una falla en el nodo de origen, nuestro proceso *shadow* finalizara antes de que el proceso migrado finalice , dicho proceso se cancelaría, sin posibilidad de reiniciarlo.

Como en el mecanismo que implementa Cóndor se lleva registro del estado de los descriptores de archivo, sólo sería necesario en este caso reiniciar el proceso *shadow*, y luego reiniciar el proceso original con el último checkpoint salvado.

La única información referente al estado externo del proceso que se está salvando en el propio archivo de checkpoint es la dirección IP del nodo origen y el identificador de proceso del proceso original. Esta información es utilizada para realizar las llamadas remotas al *shadow*, que se encontrará ejecutando en el nodo que tiene la dirección IP salvada, con el identificador de proceso del proceso original (su propio identificador de proceso, antes de realizar el checkpoint).

Ahora detallaremos la sucesión de pasos que conforman la migración de un proceso, desde el nodo origen al nodo receptor.

Para ello consideraremos lo siguiente: Del conjunto de procesos que se encuentran ejecutando en el "Nodo_Emisor", se selecciona el proceso con identificador de procesos = pid_n como el proceso a migrar. El proceso pid_n está vinculado con las librerías de checkpoint y de llamadas al sistema remotas (RPC). El pid_n tiene asociados "n" descriptores de archivos abiertos, que en nuestro caso llamaremos File_descriptor(1..n), tal como lo muestra la Fig. 6.

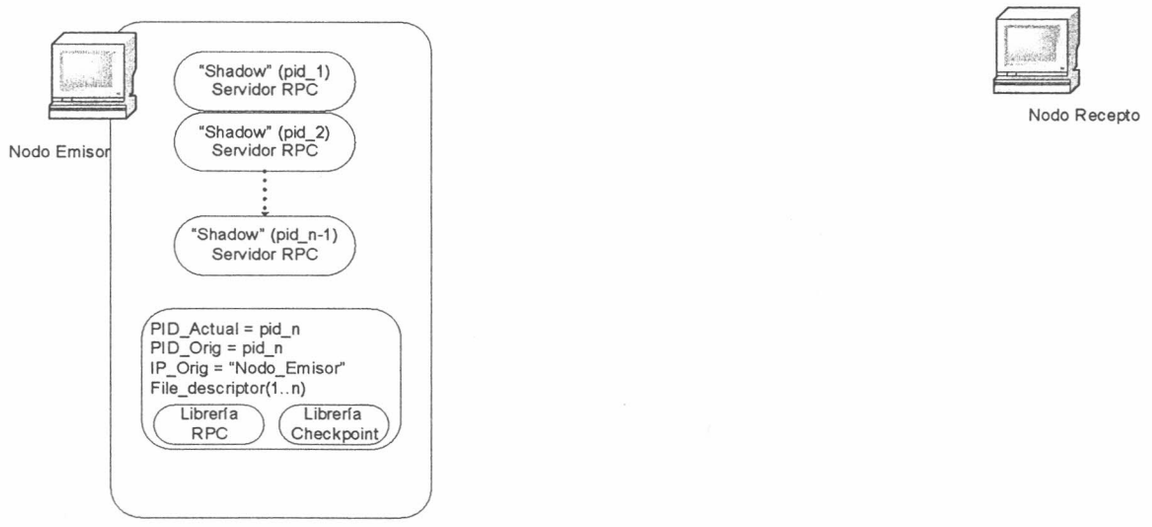


Figura 6: Proceso pid_n ejecutando en el nodo emisor.

Nuestro proceso pid_n, al recibir la señal SIGTSTP, activa el mecanismo de checkpoint/migración (Fig. 7)

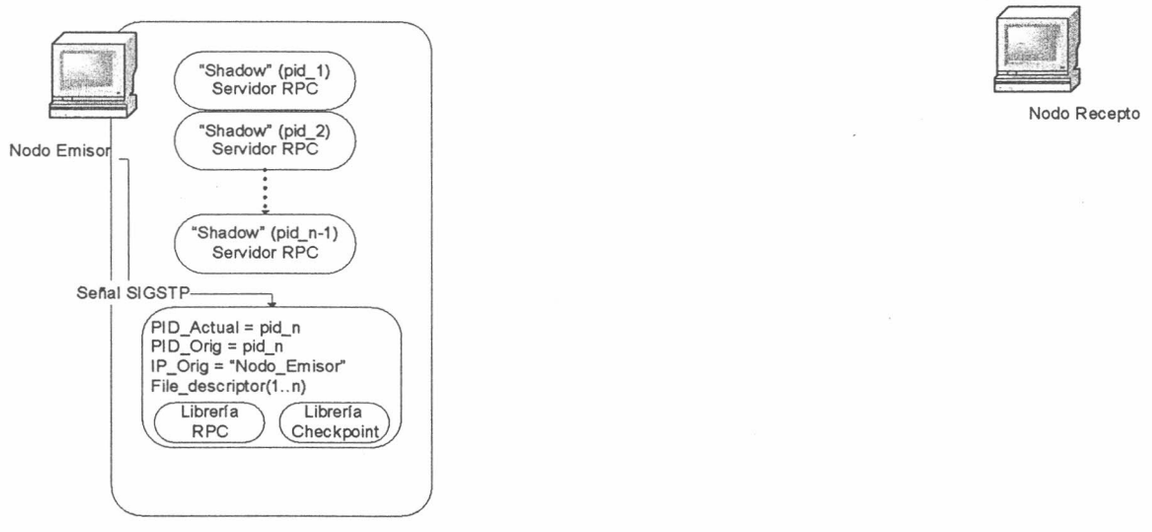


Figura 7: el proceso a migrar recibe la señal SIGSTP, la cual activa el mecanismo de checkpoin/migración.

Este mecanismo salvará en un archivo de checkpoint el estado interno del proceso, el cual también contendrá, como información del estado externo, la dirección IP del "Nodo_Emisor", y el identificador de proceso pid_n (Fig. 8).

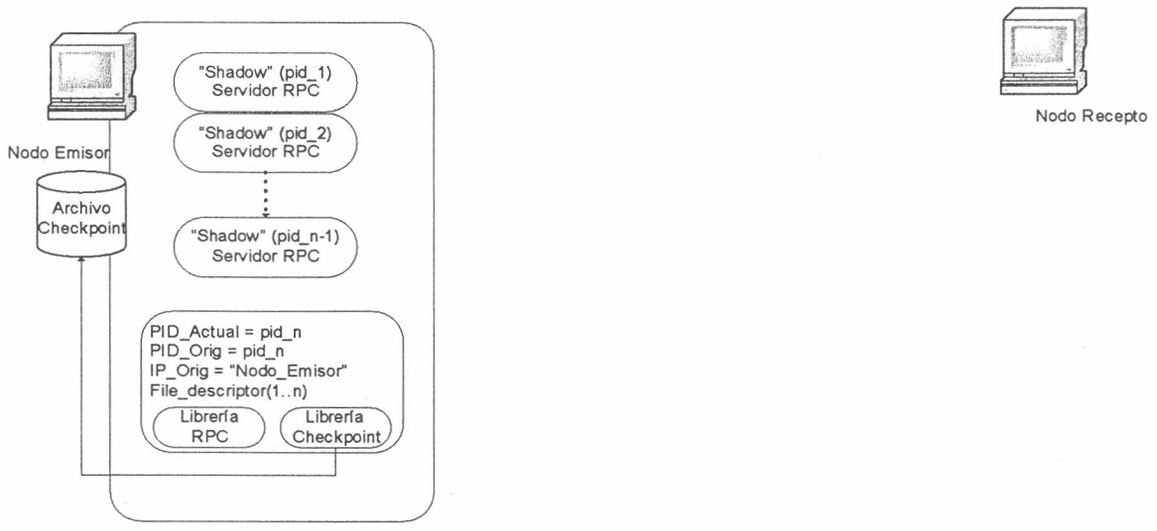


Figura 8: el estado interno se almacena en un archivo de checkpoint.

Una vez que el estado fue salvado, el propio mecanismo de checkpoint/migración, se encarga de reemplazar (por intermedio de la llamada al sistema exec()) la imagen del proceso pid_n, por la imagen del *shadow*. Este heredará el mismo identificador de proceso y el estado de los descriptores de archivo.

Luego, la imagen del proceso es transferida al "Nodo_Receptor", y allí será reiniciado como un nuevo proceso. Este nuevo proceso pid_x interactuará con su entorno de origen, a través de llamadas remotas al *shadow* correspondiente al proceso pid_n, en el nodo emisor (Fig. 9).

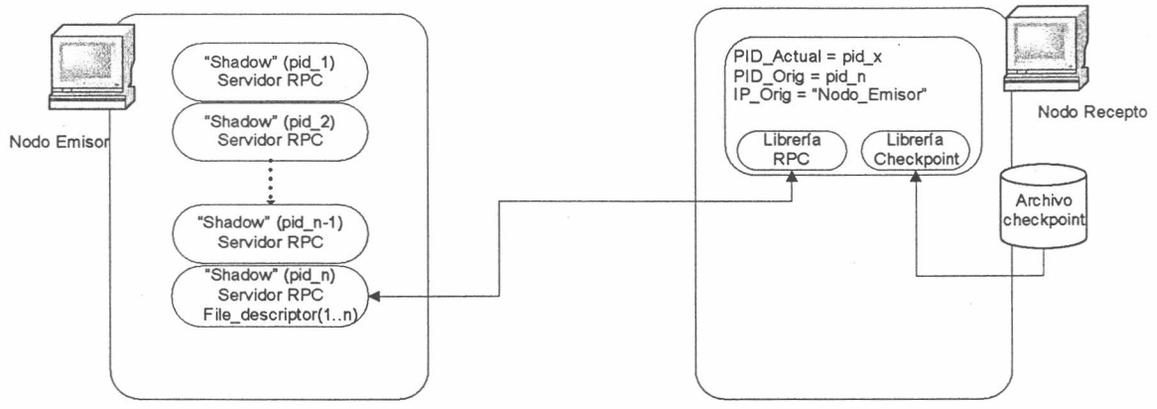


Figura 9: El proceso pid_n es migrado al "Nodo_Receptor", siendo reemplazada su imagen en el "Nodo_Emisor" por el *shadow*.

7.4.3. Implementación del proceso *Shadow*

Como vimos anteriormente, el proceso "*Shadow*" atenderá las llamadas al sistema remotas realizadas por el proceso migrado.

Este proceso es en realidad un servidor RPC (Remote Procedure Call), el cual fue implementado utilizando la versión de RPC incluida en las distribuciones de Linux que está basada en la versión RPC de ONC (Open Network Computing) del grupo Sun Microsystems.

Se utilizó el compilador rpcgen para generar los stubs de las llamadas al sistema. Por ejemplo para la implementación de la llamada al sistema write() se definió la siguiente especificación en lenguaje RPC (el cual es interpretado por rpcgen):

```
/*
 * Este programa realiza un syscall remoto
 */
      struct write_input {
      int fd;      //file descriptor
      string buf; //string a escribir
      int count; //cantidad de bytes a escribir
      };

      typedef struct write_input write_input;

      program SYSCALL_SVR
      {
        version SYSCALL_VERS
        {
         int WRITE(write_input) = 1;
        } = 1;
      } = 30000;
```

Como sabemos, una aplicación RPC es formalmente empaquetada en un "programa" con uno o más procedimientos. En este caso solo definimos, a modo de ejemplo, el procedimiento write, que recibirá como parámetro, una estructura write_input.

A cada programa se le asigna un número de programa (SYSCALL_SVR, en este caso 30000), que será conocido por los procesos que lo invoquen. También se le asigna un número de versión, para el caso en que se modifique la interface del programa (SYSCALL_VERS, en este caso 1), a la cual nosotros no le daremos uso, y, por último un número de procedimiento, que se corresponderá con cada llamada al sistema implementada. Estos números serán utilizados por el programa **portmap** para asignarle a cada programa RPC, un puerto. Cuando cada proceso *shadow* se inicie, se registra a sí mismo utilizando el programa **portmap** que le asignará un puerto, donde el *shadow* esperará y aceptará conexiones.

Tal como indicamos anteriormente, dentro del estado interno del proceso salvamos la dirección IP del nodo origen, así como también el identificador de proceso del proceso a migrar. Esta información es la que nos va a servir para que cuando el proceso sea migrado, pueda comunicarse con su *shadow*. Para ello, lo que hacemos es que cuando el proceso a migrar reemplaza su imagen con el *shadow* a través de la llamada al sistema exec(), a esta se le pasa como parámetro el identificador de proceso

del proceso a migrar, y el *shadow* utiliza este identificador para registrarse por intermedio de **portmap**, con lo cual, el proceso migrado, sabrá que en el nodo origen (cuya dirección IP salvó) y con el número de programa RPC igual al identificador de proceso que tenía el programa migrado en el nodo origen, estará esperando conexiones su *shadow*. La siguiente pieza de código nos muestra la función main() del *shadow*

```
Int main (int argc, char **argv)
{
      register SVCXPRT *transp;
      char buf[36];

      // Se instala el manejador para la señal SIGTERM
      _install_signal_handler(SIGTERM,(SIG_HANDLER)catchterm);

      if (argc != 3) //si no recibe parámetros, se registra con
                     // valores de programa-versión por defecto
      {
            sprintf(buf,"usar: syscall_svc programa version\n");
            write(0,buf,36);
            write_prog = WRITEPROG;
            write_vers = WRITEVERS;
      } else //sino, se registra con los valores
             //pasados como parámetro, que será el identificador de
             //proceso del proceso a migrar como número de programa
             //y 1 como versión
      {
            write_prog = atol(argv[1]);
            write_vers = atol(argv[2]);
      }


      pmap_unset (write_prog, write_vers);

      transp = svcudp_create(RPC_ANYSOCK);
      if (transp == NULL) {
            fprintf (stderr, "%s", "cannot create udp service.");
            exit(1);
      }
      if (!svc_register(transp, write_prog, write_vers, writeprog_1,
IPPROTO_UDP)) {
            fprintf (stderr, "%s", "No puedo registrar (write_prog,
write_vers, udp).");
            exit(1);
      }

      transp = svctcp_create(RPC_ANYSOCK, 0, 0);
      if (transp == NULL) {
            fprintf (stderr, "%s", "no puedo crear servicio tcp .");
            exit(1);
      }
      if (!svc_register(transp, write_prog, write_vers, writeprog_1,
IPPROTO_TCP)) {
            fprintf (stderr, "%s", "no puedo registrar (write_prog,
write_vers, tcp).");
            exit(1);
      }

      svc_run (); //ejecuto el servicio
      fprintf (stderr, "%s", "svc_run returned");
```

```
        exit (1);
        /* NOTREACHED */
}
```

Por ejemplo, supongamos que deseamos migrar el proceso 11450 (el cual fue vinculado con las librerías de checkpoint/migración). Cuando este reciba la señal SIGTSTP, creará el archivo de checkpoint y luego lanzará el proceso *shadow* (a través de execl("shadow","shadow","11450","1",null)) cuyo número de programa será 11450, su versión 1, y el cual estará esperando llamadas en el puerto que le asignó el programa **portmap.** Entonces, luego de enviar la señal SIGTSTP al proceso 11450, si nosotros ejecutamos el siguiente comando:

$ rpcinfo –p localhost

obtendremos la siguiente salida :

program	vers	proto	port	
100000	2	tcp	111	portmapper
100000	2	udp	111	portmapper
11450	1	tcp	1221	
11450	1	udp	1223	

Este proceso tendrá un manejador para la señal SIGTERM, que, cuando la intercepte, se encargará de desregistrarse como servicio RPC, y luego terminará.

8. Resultados Comparativos

Se realizaron distintas pruebas que involucraron dos conjuntos de procesos, uno de ellos con un uso intensivo de procesador y bajos requerimientos de entrada/salida y el otro con procesos de uso intensivo de procesador, así como también con altos requerimientos de entrada/salida.

8.1. Descripción del ambiente de pruebas:

Nodo 1 (192.168.0.1):

Procesador: Athlon 2000 - 2.0Ghz
Memoria RAM: 512 Mb
Conexión de red: Ethernet 100 Mbps
Sistema Operativo: Red Hat 8.0

Nodo 2 (192.168.0.3):

Procesador: Pentium III – 1.0 Ghz
Memoria RAM: 512 Mb
Conexión de red: Ethernet 100 Mbs
Sistema Operativo: Red Hat 8.0

Ambos nodos se encuentran conectados mediante un cable cruzado, lo que ofrece una conexión de red absolutamente pura, sin tráfico de red que afecte el intercambio de información entre los nodos.

8.2. Pruebas realizadas

Todas las pruebas se efectuaron, lanzando en simultaneo varias tareas sobre el nodo 1, para que el servicio de distribución de carga, de acuerdo con la configuración definida para cada nodo, realice la distribución de los procesos sobre ambos nodos o manteniendo los mismos en el nodo origen, para poder comparar los tiempos de respuesta final desarrollados con distribución de carga o sin distribución de carga. Cabe aclarar que los tiempos tomados son totales, esto es, incluyen los tiempos de procesamiento propio de cada tarea mas los tiempos de transferencia de los archivos de checkpoint , cuando esto aplique. Por otro lado, cuando especificamos que una prueba se realizó con procesos que hacen uso de entrada/salida, esta se hizo sobre archivos en disco.

Las pruebas realizadas, junto con sus tiempos y la evaluación de los resultados se detallan a continuación:

8.2.1. Primera Prueba:

Seis procesos de calculo de matrices sin entrada/salida y sin migración

Descripción de los procesos ejecutados durante la prueba:

Cada proceso calcula la multiplicación de dos matrices con un tamaño de 1000 x 1000 cada una.

Configuración de cotas:

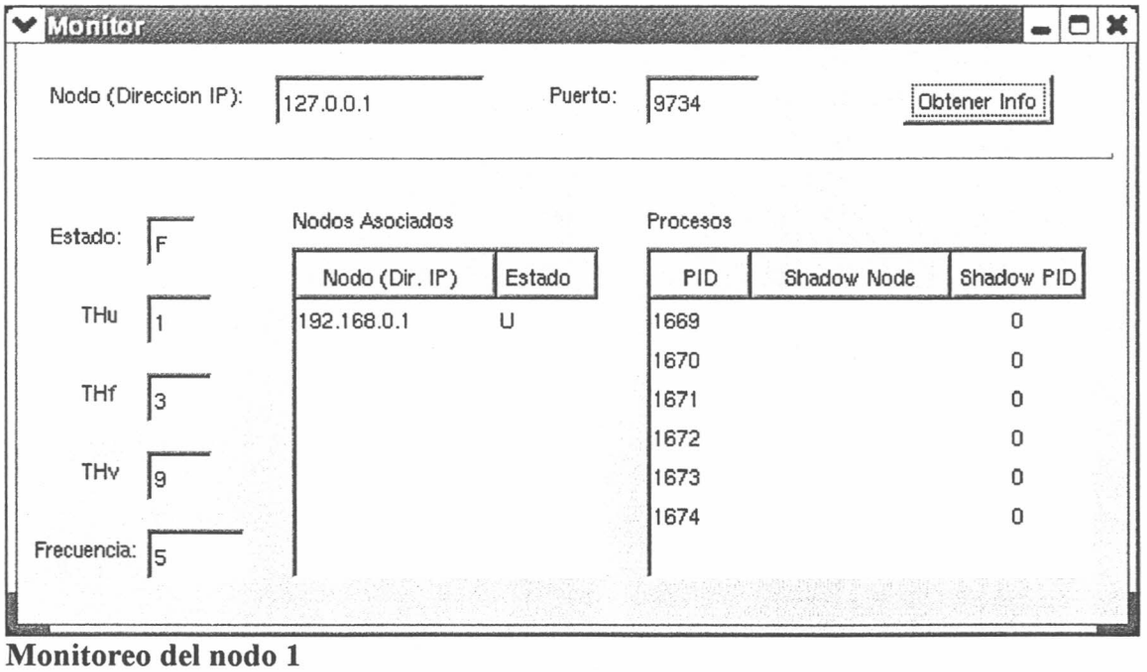
Nodo 1:

THu: 1
THf: 3
THv: 9

Nodo 2:

THu: 1
THf: 3
THv: 4

La elección de la cota THv = 9 en el nodo 1 va a evitar que los seis procesos enviados al nodo 1 puedan llegar a ser migrados, debido a que recién en un potencial noveno proceso se desencadenaría la migración de un proceso hacia otro nodo, por lo tanto, los seis procesos estarán trabajando sobre un mismo nodo.



Monitoreo del nodo 1

Tabla de resultados:

PID	Tiempo Total (en segundos)	Migrado
1669	140.10	No
1670	140.24	No
1671	140.47	No
1672	140.35	No
1673	140.57	No
1674	140.51	No

Resumen de resultados obtenidos:

Promedio:	140.37
Menor Tiempo:	140.10
Mayor Tiempo:	140.57

8.2.2. Segunda Prueba:

Seis procesos de calculo de matrices sin entrada/salida con migración de procesos.

Descripción de los procesos ejecutados durante la prueba:

Cada proceso calcula la multiplicación de dos matrices con un tamaño de 1000 x 1000 cada una.

Configuración de cotas:

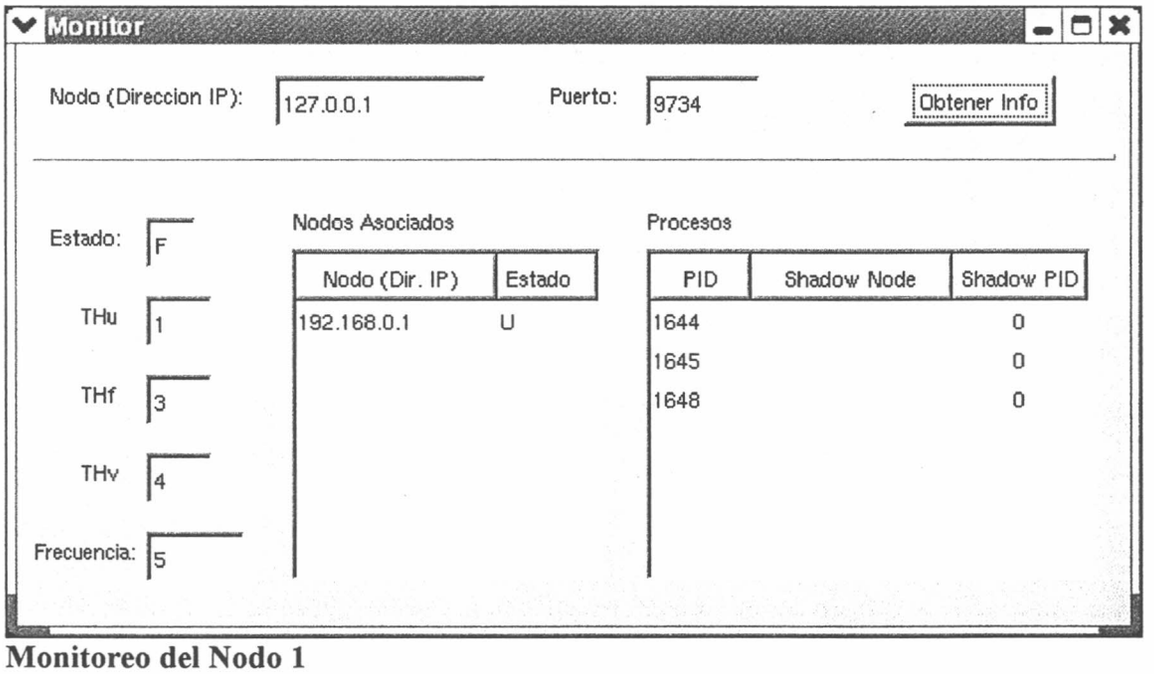
Nodo 1:

THu: 1
THf: 3
THv: 4

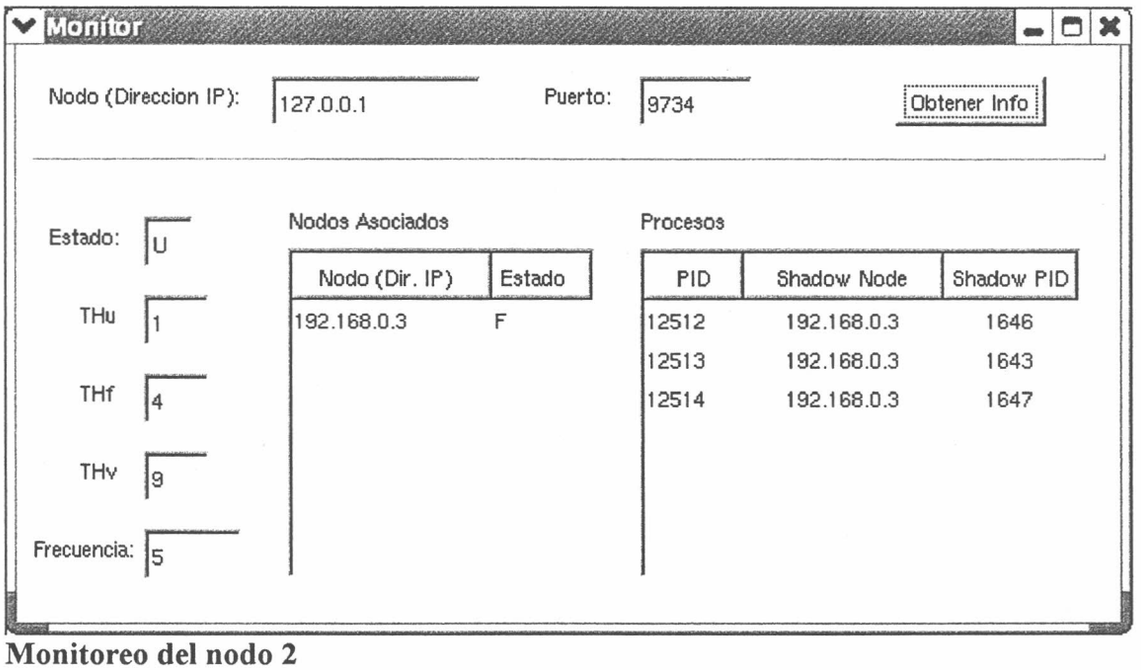
Nodo 2:

THu: 1
THf: 3
THv: 4

La elección de esta cotas permite una distribución simétrica en la carga de cada uno de los nodos (Tres procesos se mantendrán trabajando en el nodo 1 y los otros tres procesos serán migrados hacia el nodo 2).



Monitoreo del Nodo 1



Monitoreo del nodo 2

Tabla de resultados:

PID	Tiempo Total (en segundos)	Migrado
1644	71.39	No
1645	71.42	No
1648	71.26	No
1646	84.70	Si
1643	84.72	Si
1647	84.79	Si

Resumen de resultados obtenidos:

Promedio:	78.05
Menor Tiempo:	71.26
Mayor Tiempo:	84.79

8.2.3. Análisis de los resultados de la primera y segunda prueba

	Primera Prueba (sin distribución de carga)	Segunda Prueba (con distribución de carga)
Promedio	140.37	78.05
Menor Tiempo	140.10	71.26
Mayor Tiempo	140.57	84.79

Como podemos observar, los resultados obtenidos en la segunda prueba, donde se hace uso del mecanismo de equilibrio de carga sobre procesos con intenso uso de procesador ofrecen mejores tiempos de respuesta, verificándose una mejora de casi el 45 % en el tiempo de respuesta promedio, una mejora del 50 % sobre los procesos en el nodo origen y una mejora del 40 % en los procesos migrados hacia otro nodo. A partir de este resultado podemos concluir que la mejora es significativa cuando se aplica la distribución de la carga en procesos que no realizan entrada/salida o estas son mínimas.

8.2.4. Tercera Prueba:

Seis procesos de calculo de matrices con envío del resultado de cada cálculo realizado a un archivo de salida y sin utilización de la migración de procesos.

Descripción de los procesos ejecutados durante la prueba:

Cada proceso calcula la multiplicación de dos matrices con un tamaño de 1000 x 1000 cada una y en cada operación realizada se envía su resultado hacia un archivo de salida.

Configuración de cotas:

Nodo 1:

THu: 1
THf: 3
THv: 9

Nodo 2:

THu: 1
THf: 3
THv: 4

La elección de la cota THv = 9 en el nodo 1 va a evitar que los seis procesos enviados al nodo 1 puedan llegar a ser migrados, debido a que recién en un potencial noveno proceso se desencadenaría la migración de un proceso hacia otro nodo, por lo tanto, los seis procesos estarán trabajando sobre un mismo nodo.

Tabla de resultados:

PID	Tiempo Total (en segundos)	Migrado
1734	188.63	No
1735	189.10	No
1736	190.62	No
1737	187.83	No
1738	189.44	No
1739	187.62	No

Resumen de resultados obtenidos:

Promedio:	188.87
Menor Tiempo:	187.62
Mayor Tiempo:	190.62

8.2.5. Cuarta Prueba:

Seis procesos de calculo de matrices con envío del resultado de cada cálculo realizado a un archivo de salida y con migración de procesos.

Descripción de los procesos ejecutados durante la prueba:

Cada proceso calcula la multiplicación de dos matrices con un tamaño de 1000 x 1000 cada una y en cada operación realizada se envía su resultado hacia un archivo de salida.

Configuración de cotas:

Nodo 1:

THu: 1
THf: 3
THv: 4

Nodo 2:

THu: 1
THf: 3
THv: 4

La elección de esta cotas permite una distribución simétrica en la carga de cada uno de los nodos (Tres procesos se mantendrán trabajando en el nodo 1 y los otros tres procesos serán migrados hacia el nodo 2).

Tabla de resultados:

PID	Tiempo Total (en segundos)	Migrado
1834	241.57	No
1835	240.64	No
1836	716.87	Si
1837	433.42	Si
1838	243.00	No
1839	696.51	Si

Resumen de resultados obtenidos:

Promedio:	428.67
Menor Tiempo:	240.64
Mayor Tiempo:	716.87

8.2.6. Análisis de los resultados de la tercera y cuarta prueba

	Tercera Prueba (sin distribución de carga)	Cuarta Prueba (con distribución de carga)
Promedio	188.87	428.67
Menor Tiempo	187.62	240.64
Mayor Tiempo	190.62	716.87

En esta ocasión, observamos que los resultados de la cuarta prueba, en donde se produjo la migración de procesos con intenso uso de procesador e intenso uso de entradas/salidas, muestran un peor rendimiento general y particular en los procesos. Cuando los procesos no migraron fueron 55% más rápidos en promedio que cuando distribuimos la carga con migración de procesos. Como se puede observar con estas pruebas, el rendimiento global de los procesos intervienientes empeoró significativamente al ser migrados algunos de dichos procesos. Este resultado, tal como habíamos previsto, se debe principalmente a que las entradas/salidas que realizaron los procesos migrados, se hicieron por intermedio de llamadas RPC a sus procesos *"shadow"* asociados.

9. Conclusiones y Trabajos Futuros

9.1. Conclusiones

En ésta tesis hemos implementado una herramienta para efectuar distribución de carga y migración de procesos sobre una red de computadoras con sistemas operativos de tipo Unix dentro del espacio de usuario (sin necesidad de modificar el kernel del sistema operativo). Dicha herramienta implementa la distribución de la carga de manera dinámica y descentralizada. La migración de procesos se realizó utilizando una técnica de checkpoint basada en la que se utiliza en el sistema de procesamiento distribuido Condor, al cual se le efectuaron cambios que implementan mejoras en cuanto a funcionalidad, pero que no tienen impacto, positivo o negativo, sobre la performance del mecanismo de checkpoint que posee la implementación original.

La distribución de la carga, al ser descentralizada, evita tener un solo nodo central que provea de la información de estado del sistema al resto de los nodos, con la consiguiente ganancia en cuanto a tolerancia a fallos. Además, la utilización de una lista de nodos asociados que se conocen mutuamente, favorece la disminución del trafico de la red, ya que solo se intercambia información entre grupos reducidos de nodos (los integrantes de la lista de nodos asociados). El dinamismo generado por la difusión de cambios de estados utilizando cotas de infrautilización y sobrecarga ayuda a mantener, de manera consistente y real, la imagen del estado del sistema en cada nodo con muy poca carga extra sobre la red.

Las mejoras funcionales que posee nuestra implementación de checkpoint/migración con respecto a la original son:

- Los programas no necesitan ser enlazados ni compilados con las librerías que implementan este mecanismo, por lo tanto, podríamos utilizar este mecanismo con programas sobre los cuales no contamos con el código fuente.

- Permite que los procesos interactúen con dispositivos y/o tipos de archivos que no soporten reposicionamiento o tengan una semántica especial cuando se cierran (por ejemplo: dispositivos que realicen operaciones de rebobinado cuando se cierran).

- Los procesos migrados con esta técnica podrían ser procesos que se intercomuniquen vía sockets, ya que se mantiene el estado de los descriptores de archivos luego de que el proceso ha sido migrado.

Los resultados obtenidos durantes las pruebas realizadas con el mecanismo de distribución de carga propuesto, demostraron que esta implementación produce un incremento notable del rendimiento sobre procesos con uso intensivo del procesador y con bajos requerimientos de entrada/salida, sin embargo, frente a procesos con altos

requerimientos de entrada/salida, este mecanismo generó resultados más pobres que los obtenidos sin distribución de la carga, esto se debe a la sobrecarga adicional que se genera cada vez que se envía una petición de entrada/salida de un proceso migrado hacia el shadow que se encuentra en el nodo origen.

9.2. Trabajos Futuros

Algunas mejoras a la herramienta que podemos mencionar son las siguientes:

- El algoritmo de distribución de carga implementado trabaja y toma decisiones sobre una métrica de carga basada en la longitud de la lista de procesos. Para una mejor toma de decisiones, se podría llegar a pensar en otras métricas que se basen no solo en la cantidad de procesos cargados sobre un nodo, sino también en las características de cada proceso y los recursos tomados por ellos.

- Implementación de todas la llamadas al sistema que interactúen con el sistema de archivos (actualmente están implementadas las llamadas al sistema open(),close(),write() y read()).

- Implementar un mecanismo de checkpoint/migración que evite que el código de las librerías compartidas utilizadas por el proceso a migrar, formen parte del archivo de checkpoint, y que garantice que, si dichas librerías existen en el nodo destino, estas sean cargadas en la misma dirección y con la misma protección que tenían estas cuando el proceso se encontraba ejecutando en el nodo origen. Esto ahorraría tiempo de transferencia, y carga de memoria.

- Implementar un nivel de abstracción de procesos virtuales, que permitan unificar la referencia a un proceso determinado en el conjunto de los nodos intervinientes en el *dominio de equilibrio*, de esta manera podríamos hacer participar en el esquema de distribución de la carga a procesos que se intercomuniquen (mas allá de la comunicación por intermedio de sockets que nuestra implementación estaría en condiciones de soportar) y procesos que puedan generar procesos hijos.

10. Instalación y Ejecución del Servicio

10.1. Instalación

El servicio de distribución de carga y migración de procesos esta conformado por los siguientes archivos:

- loadbalanced: demonio responsable del servicio de la distribución de la carga.
- setloadbalanced: ejecutable encargado de lanzar un proceso dentro del mecanismo de distribución de la carga.
- loadbalanced.cfg: archivo de configuración local del demonio loadbalanced.
- libckpt_wrapper.so: librería de checkpoint.
- lib_restart.so: librería de restauración de un proceso
- syscall_rpc_svc: proceso *shadow* que reemplazará a un proceso cuando este es migrado hacia otro nodo y que será el responsable de atender las llamadas al sistema remotas.
- restart: ejecutable encargado del mecanismo de restauración de un proceso migrado.

Para la instalación del servicio es necesario realizar los siguientes pasos en cada uno de los nodos participes del mecanismo de distribución de la carga:

Copiar los archivos loadbalanced, setloadbalanced, syscall_rpc_svc y restart a cualquier directorio del sistema, aunque seria preferible instalarlos en un directorio que pertenezca al PATH del sistema, permitiendo de este modo que estos ejecutables puedan ser invocados sin necesidad de especificar su camino. Además, dichos archivos deben tener habilitados el permiso de ejecución para los usuarios que cuenten con la posibilidad de lanzar aplicaciones que utilicen el servicio de distribución de carga.

Copiar el archivo loadbalanced.cfg al directorio /etc (directorio utilizado de manera habitual por la aplicaciones para obtener sus datos de configuración).

Copiar las librerías libckpt_wrapper.so y lib_restart.so a cualquier directorio del sistema y fijarlas a las variables de ambiente LD_PRELOAD y LIB_RESTART respectivamente. Por ejemplo, se podrían copiar ambas librerías al directorio "/usr/lib/" y luego fijar los valores de LD_PRELOAD y LIB_RESTART del siguiente modo:

export LD_PRELOAD=/usr/lib/libckpt_wrapper.so
export LIB_RESTART=/usr/lib/lib_restart.so

Si además se llegará a desear lanzar a los procesos participes del esquema de distribución de la carga sin utilizar el programa setloadbalanced, se deberán fijar las siguientes variables de ambiente:

export CKPT_LOADBALANCED_PORT= {número de port de atención del demonio loadbalanced }
export CKPT_SHADOW={ejecutable syscall_rpc_svc *}*
export CKPT_RESTARTLIB={ejecutable restart}

10.2. Inicialización del servicio

Antes de iniciar el servicio es necesario registrar en el archivo de configuración /etc/loadbalanced.cfg los parámetros de ejecución que utilizará el demonio loadbalanced. Dicha información puede ser editada con cualquier editor de texto.

Ejemplo de loadbalanced.cfg:

```
# Lista de nodos asociados
192.168.0.3

# Cotas
THu 1
THf 5
THv 8

# Puerto Socket
Port 9734

# Frecuencia de examinación de procesos activos, en segundos
Frequency 5

# Aplicacion que inicia la restauración de un proceso migrado
Restart /dos/Tesis/bin/restart

# Aplicacion shadow
Ckpt_Shadow /dos/Tesis/bin/syscall_rpc_svc

# Librería de restauración de procesos migrados
Lib_Restart /dos/Tesis/bin/librestart.so

# Libreria de checkpoint
Lib_Ckpt /dos/Tesis/bin/libckpt.so

# Tipo de selección del proceso a migrar
# 1 : Menor tiempo de consumo de procesador.
# 2 : Mayor tiempo de consumo de procesador.
```

3 : Menor consumo de memoria.
4 : Mayor consumo de memoria.
SelectProcessToMigrate 1

Para iniciar el servicio de distribución de carga, simplemente se debe invocar la ejecución del demonio loadbalanced, el cual comenzará a ejecutarse en background dentro del sistema. Dicha invocación podría efectuarse de manera automática al inicializarse el sistema, ingresando la invocación del demonio loadbalanced en los scripts de /etc/rc.d, los cuales son ejecutados durante el proceso de inicialización del sistema operativo Linux.

Cuando se pretende que una aplicación sea participe del servicio de distribución de carga, se deberá invocar dicha aplicación utilizando el ejecutable setloadbalanced. El comando de invocación de una aplicación es el siguiente:

setloadbalanced {nombre_de_la_aplicacion}

Esto hará que la aplicación pueda ser registrada dentro del mecanismo de distribución de carga, con la posibilidad de poder ser migrado a otros nodos del sistema. (Atención: las aplicaciones, para que puedan ser migradas a distintos nodos, deberán estar instaladas sobre directorios que pertenezcan al PATH del sistema, para que de este modo puedan ser invocadas sobre diferentes nodos, sin la necesidad de que estas se encuentren instaladas sobre una misma ruta de acceso en cada nodo).

11. Bibliografía

[1] Kang Shin y Yi-Chieh Chang. "Load sharing in distributed real-time systems with state-change broadcasts". IEEE Transactions on computers VOL.38, 1989.

[2] M. Litzkow, T. Tannenbaum, J. Basney y M. Livny. "Checkpoint and migration of Unix processes in the Condor distributed processing system". Usenix, 1992.

[3] Cheng Zhong Xu y Francis C. M. Lau. "Load balancing in parallel computers, theory and practice". Kluwer Academic Publishers, 1997.

[4] T. Kunz. "The influence of different workload descriptions on a heuristic load balancing scheme". IEEE Transactions on Parallel and Distributed Systems, 1993.

[5] D. Ferrari y S. Zhou. "An empirical investigation of load indices for load balancing applications". Proceedings of Performance'87, 1987.

[6] T. F. Znati, R. G. Melhem y K. R. Pruhs. "Dilation-based bidding schemes for dynamic load balancing on distributed processing systems". Proceedings of 6th Distributed Memory Computing Conference, 1991.

[7] D. M. Nicol y H. Saltj. "Dynamic remapping of parallel computations with varyng resource demands". IEEE Transactions on Computers, 1988.

[8] F. C. H. Lin y R. M. Keller. "The gradient model load balancing method". IEEE Transactions on Software Engineering, 1987.

[9] L.M. Ni, C.W. Xu y T.B.Gendreau, "A distributed drafting algorithm for load balancing". IEEE Transactions on Computers, 1988.

[10] D. M. Nicol y H. Saltj, "Dynamic remapping of parallel computations with varyng resource demands". IEEE Transactions on Computers, 1988.

[11] R. Chowkwanyun y K. Hwang. "Multicomputer load balancing for concurrent lisp execution". Parallel processing for Supercomputers and Artifical Intelligence, 1989.

[12] W. Shu y L. V. Kale. "A dynamic scheduling strategy for the Chare kernel systems". Proceedings of Supercomputing, 1989.

[13] P. Krueger y M. Livny. "A comparision of preemptive and non-preemptive load distribution". Proceedings of International Conference on Distributd Computing Systems, 1998.

[14] C. Z. Xu y F. C. M. Lau. "Iterative dynamic load balancing in multicomputers". Journal of Operational Research Society, 1994.

[15] A.S. Tanenbaum. "Distributed Operating Systems". Prentice Hall, 1995.

[16] A.Goscinski. "Distributed Operating Systems (The logical design)". Addison-Wesley Publishing Company, 1992

[17] S. Petri y H. Langendorfer. "Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes". Operating Systems Review, 1995, vol.29 nro. 4.

[18] Yannis Smaragdakis."Layered Development with (Unix) Dynamic Libraries". Georgia Institute of Technology, College of Computing.

[19] P.Mehra y B.W. Wah. "Adaptive Load-Balancing Strategies for Distributed Systems". IEEE Computer Society, 1992.

[20] Manish Mehta. "Intelligent Load Balancing" University of Texas, 1999.

[21] M.B. Jones. "Interposition Agents: Transparently Interposing User Code at the System Interface". Microsoft Corporation.

[22] J.S. Plank, M. Beck y G. Kingsley. "Libckpt: Transparent Checkpointing under Unix". Usenix Conference Proceedings, January 1995.

[22] Brian Kernighan y Dennis Ritchie. "El lenguaje de programación C". Prentice-Hall, 1991.

[23] Brian Kernighan y Rob Pike. "El entorno de programación Unix". Prentice-Hall, 1987.

[24] Mark Mitchell, Jeffrey Oldham y Alex Samuel – "Advanced Linux Programming". New Riders Publishing, 2001

[25] Andrew Tanembaum. "Modern operating systems". Prentice-Hall, 1992

[26] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pirre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Systems. Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto, 1992.

[27] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: A Global Layer Unix for a Network of Work-stations. Technical report, Computer Science Division, University of California, Berkeley, 1997

[28] MOSIX Homepage. http://www.cs.huji.ac.il/mosix/ .

[29] Y. Khalidi, J. Bernabeu, V. Matena, K. Shirri_, and M. Thadini. Solaris MC: A Multicomputer Operating System. USENIX 1996 Annual Technical Conference, pages 191{203, January 1996.