Un Algoritmo Distribuido y Cooperativo para Balance de Carga Dinámico

Natalia L. Weinbach Javier Echaiz Alejandro J. García
nlw@cs.uns.edu.ar je@cs.uns.edu.ar ajg@cs.uns.edu.ar

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur.
Av. Alem 1253, (8000) Bahía Blanca, Argentina

Resumen

En este trabajo presentamos un algoritmo de balance de carga, que presenta las siguientes características: es dinámico, distribuido, cooperativo, global, de asignación por única vez, sub-óptimo y heurístico. En nuestra propuesta, cada nodo recibe información de la carga de otros nodos en forma asincrónica, lo que le permite calcular de antemano cuál nodo tiene la menor carga en ese momento. Al arribar un nuevo proceso P a un nodo N, en función de la carga de N y el mínimo previamente calculado, N decide si P se ejecuta localmente o directamente lo transfiere al mejor candidato considerado por N. El algoritmo no requiere información provista por el usuario, ni archivos históricos de ejecución para clasificar los tipos de procesos. Es estable y escalable mediante el ajuste de ciertos parámetros como la cantidad de nodos que puede atravesar durante la migración o transferencia inicial de proceso, y el valor de un *umbral*, que regula la distancia hacia el nodo receptor e influye también en la confiabilidad de la información de carga que se conoce del resto de los nodos. Al ser distribuido, presenta tolerancia a fallas, ya que no se asigna una responsabilidad diferenciada a sólo algunos nodos del sistema.

Palabras clave: Sistemas distribuidos. Paralelismo. Algoritmos de balance de carga

1. Introducción

En sistemas distribuidos o arquitecturas paralelas, una buena distribución de la carga permite obtener sustanciales mejoras en la performance del sistema. Sin embargo, uno de los principales problemas que enfrentan la mayoría de los algoritmos de balance de carga, es poder decidir con rapidez dónde ejecutar un nuevo proceso. El tiempo empleado en obtener la carga de nodos remotos puede retardar notoriamente el inicio de ejecución de un nuevo proceso, aumentando de esta forma el tiempo de ejecución del proceso y degradando la performance global del sistema.

En este trabajo presentamos un algoritmo de balance de carga, que siguiendo la clasificación de Casavant en [1], presenta las siguientes características: es dinámico, distribuido, cooperativo, global, de asignación por única vez, sub-óptimo y heurístico (ver sección 3). Es importante destacar que el balance de carga estático resulta simple y efectivo cuando la carga puede caracterizarse lo suficientemente bien de antemano, pero falla cuando existen fluctuaciones en la carga del sistema [11]. El algoritmo que se ha desarrollado es dinámico, ya que reacciona ante parámetros del entorno que cambian dinámicamente, y balancea la carga del sistema en el momento que los procesos arriban a un nodo.

En nuestra propuesta, cada nodo recibe información de la carga de otros nodos en forma asincrónica, lo cual le permite calcular de antemano qué nodo tiene la menor carga en ese momento. Al arribar un nuevo proceso P a un nodo N, en función de la carga de N y el mínimo previamente calculado, N decide si P se ejecuta localmente o directamente lo transfiere al mejor candidato considerado por N. Al tratarse de un algoritmo distribuido la responsabilidad de la asignación dinámica de procesos no reside en un único nodo, sino que la tarea está físicamente distribuida entre los nodos por el intercambio de información que realizan. Esto provee mayor confiabilidad y tolerancia a fallas.

En un algoritmo de balance de carga pueden identificarse varias políticas a ser desarrolladas [6]: de intercambio de información de estados, de selección del nodo destino (ubicación), de transferencia de procesos, de estimación de carga, de limitación a la migración y de determinación de prioridades en la ejecución. Como se verá más adelante, el algoritmo que se presenta en este trabajo considera fundamentalmente las tres primeras.

Habitualmente se identifican dos categorías de algoritmos de balance de carga [1, 7, 11]: los iniciados por el emisor y los iniciados por el receptor. En los primeros cada nodo al recibir un proceso, toma la iniciativa de balancear la carga decidiendo si lo ejecuta local o remotamente. En los iniciados por el receptor o servidor, los nodos poco cargados piden procesos para ser ejecutados por ellos. El algoritmo aquí presentado cae en la categoría de iniciado por el emisor.

El trabajo está organizado de la siguiente manera. En la sección siguiente, se describirá el algoritmo propuesto, las políticas mencionadas, y se ilustrará el funcionamiento del algoritmo en un ejemplo. En la sección 3 se evaluarán las características y propiedades del algoritmo de acuerdo a clasificaciones conocidas. En la sección 4 se hará una comparación con otros algoritmos citados en la literatura. Finalmente se incluyen las conclusiones y el trabajo futuro.

2. Algoritmo propuesto

El algoritmo trabaja sobre un conjunto de nodos que están conectados entre sí y que pueden tener cualquier topología de conexión. En forma abstracta, este conjunto de nodos puede verse como un grafo no dirigido etiquetado $G = (V, A)$, que consta de un conjunto finito de vértices V que representan los sitios de procesamiento o nodos, y un conjunto A de aristas o arcos que representan las conexiones entre sitios. Estas conexiones son lógicas y son fundamentales para el intercambio de información de estado. En la realidad pueden existir más conexiones físicas de las que se muestran en el grafo, pero para no disminuir la performance del sistema, es deseable que las conexiones lógicas que se establezcan cuenten con una conexión física subyacente (ver figura 1).

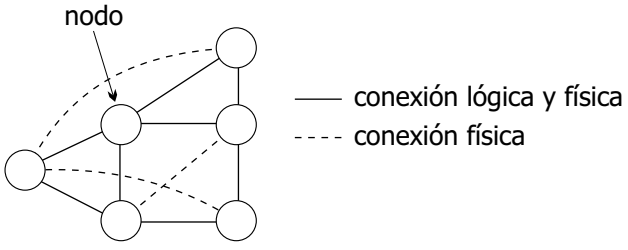

Figura 1: Tipos de conexiones

Cada nodo tiene una etiqueta que representa la información sobre su carga actual, que puede ser calculada, por ejemplo, utilizando el número de procesos que están en la cola de listos. Nuestro algoritmo asume una función $c(N)$ que calcula la carga de un nodo N. El lector interesado puede encontrar información sobre este tipo de funciones en [4, 10].

Cada nodo N mantiene además una tabla interna compuesta por tuplas de la forma $< nodo, carga, longitud, destino >$. Cada tupla indica el *nodo* vecino que la genera, la *carga* mínima informada por dicho vecino, el nodo *destino* que posee dicha carga y la *longitud* del camino que separa a ambos nodos (ver figura 2).

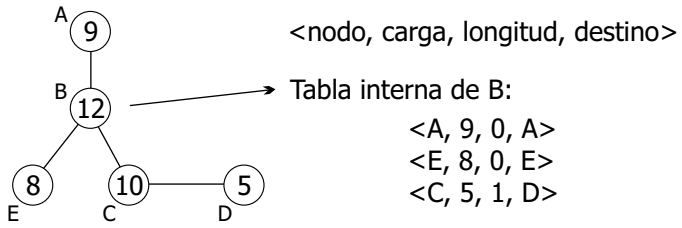

Figura 2: Tuplas para el intercambio de datos

Como puede verse en la figura 2, por cada arco que llega al nodo se tiene la mínima carga que hay en alguno de los nodos alcanzables por esa vía. Esta información es provista directamente por un vecino en forma asincrónica. Al llegar esta información, el nodo puede calcular el mínimo entre su propia carga y la proveniente de los nodos vecinos. Una vez calculado, propaga este mínimo a los sitios vecinos, que a su vez harán el mismo procedimiento.

De este modo, la carga de los nodos y la carga mínima se encuentra en actualización continua y asincrónica. Cuando un nodo recibe de un vecino un nuevo mínimo, o cuando su carga cambia, recalcula el mínimo y envía un mensaje a los nodos adyacentes para hacerles conocer la modificación. Así, la información del nodo menos cargado se propagará por el grafo, y cada nodo conocerá como acceder a dicho mínimo (por el nodo que le transmitió la información).

Es claro que la propagación de información lleva tiempo. Por lo tanto, la información de un mínimo en un nodo muy distante puede estar desactualizada. Para solucionar este problema, el algoritmo tiene definido un *umbral* que establece la distancia máxima a la que se propaga un mínimo en el grafo. Si el *umbral* es 0, entonces cada nodo sólo conocerá su carga actual y no propagará información. Si el *umbral* es mayor o igual al camino más largo en el grafo que representa al sistema, entonces el mínimo se propagará a todos los nodos. Como se mostrará en el ejemplo de la sección 2.4, si el *umbral* es menor al camino más largo, entonces cada nodo tendrá la información del mínimo a una distancia menor o igual al *umbral*. El valor del *umbral* podrá adecuarse al sistema en forma manual o automática.

Cuando se crea un proceso nuevo en el sistema, el nodo que lo origina, debe decidir si el proceso se ejecutará localmente o a cuál nodo transferirlo. Si decide migrar el proceso, utiliza el mínimo previamente calculado para elegir el nodo destino. Puede ocurrir que mientras un proceso esté en camino hacia un nodo ligeramente cargado, este nodo aumente su carga debido a que todos los procesos nuevos del sistema quieran ejecutarse allí. Sin embargo, el nodo receptor puede volver a realizar una transferencia del proceso si ahora se encuentra muy cargado. Además, si el proceso no viaja directamente hacia el nodo destino sino que pasa por algunos nodos intermedios, dinámicamente se modificará su trayectoria en base a las actualizaciones de las etiquetas.

2.1. Política de intercambio de información

El intercambio de información se realiza únicamente cuando se hace necesaria una actualización, esto es, cada vez que la carga del nodo (representada por la función $c(N)$) cambia, o se recibe un nuevo mínimo de algún vecino. Cada nodo N utiliza el siguiente algoritmo de actualización.

Algoritmo Actualización:

1. Esperar hasta que $c(N)$ cambie o arribe una nueva tupla de un vecino.
2. Si arribó una nueva tupla entonces reemplazar la tupla anterior de la tabla.
3. Recalcular el mínimo utilizando el algoritmo Calcular Mínimo.
4. Hacer un multicast a los vecinos.
5. Ir a 1.

Algoritmo Calcular Mínimo:

1. Recolectar los datos que le llegan desde los vecinos (las etiquetas de los arcos) que están almacenados en una tabla interna.
2. Quitar de esta lista los valores que refieren a sí mismo (el propio nodo es quien tiene datos más actualizados).
3. Armar una sublista $L_{vecinos}$ con los valores $< nodo, carga, longitud, destino >$, donde $carga$ representa la carga mínima que conoce el $nodo$; y además se cumple que $longitud < umbral$, donde $umbral$ es un número que afecta a la localidad del algoritmo. Es decir, si $umbral = 0$ cada nodo siempre toma su carga propia como la mínima, si $umbral =$ [máxima longitud de camino en el grafo], todos los nodos consideran los valores de carga de todo el resto.
4. Con $L_{vecinos}$ armar una sublista $L_{min-carga}$ con las tuplas $< nodo, carga, longitud, destino >$ con los valores que tengan carga mínima (puede haber más de un nodo con el mismo valor).
5. Con $L_{min-carga}$ armar una sublista $L_{min-carga-cercanos}$ con los valores que tengan longitud de camino mínima (serían los de mínima carga y que están más cercanos).
6. Asignar a Min una tupla cualquiera de $L_{min-carga-cercanos}$.
7. Si $c(N) \leq Min$ entonces asignar a Min la tupla $< N, c(N), 0, N >$, ya que la carga local es la mínima.

La tupla Min que se obtiene en el último paso, es la que se propaga en el multicast. Además, si el nodo sabe que este mínimo que obtuvo es el mismo que propagó la vez anterior, no es necesario que lo envíe en multicast. Sólo es necesario enviar los valores cuando cambian. Esta optimización disminuye los costos de comunicación y procesamiento, pues no obliga a los nodos que reciben la información a calcular nuevamente el mínimo.

Por ejemplo, supongamos que para el nodo A, con carga propia 4, tenemos las siguientes entradas en su tabla interna: $< C, 3, 1, E >$; $< B, 4, 2, A >$; $< D, 3, 4, E >$; $< F, 7, 3, Z >$; $< T, 3, 1, W >$; $< G, 3, 2, K >$.

En el segundo paso se quita la segunda tupla ya que es información del propio nodo. Con un valor de $umbral = 4$ se tiene:

$L_{vecinos}$= $[< C, 3, 1, E >; < F, 7, 3, Z >; < T, 3, 1, W >; < G, 3, 2, K >]$;

$L_{min-carga}$= $[< C, 3, 1, E >; < T, 3, 1, W >; < G, 3, 2, K >]$;

$L_{min-carga-cercanos}$= $[< C, 3, 1, E >; < T, 3, 1, W >]$;

Luego, el nodo E o W serían los posibles receptores de un proceso del nodo A que necesite ser migrado.

2.2. Política de Transferencia

Cuando algún nodo N del sistema cuenta con un nuevo proceso, ya sea porque se originó allí o porque lo recibió de algún otro nodo, éste nodo utiliza el valor de $c(N)$ para decidir si conviene ejecutarlo localmente o es preferible encontrar alguna ubicación alternativa. Si el nodo N está muy cargado, esto es, $c(N)$ es mayor que un cierto *valor límite* (*'threshold value'*), se decide ejecutar el proceso en el sitio de menor carga. Para cada nodo N se utiliza el siguiente algoritmo de balance de carga.

Algoritmo Balance de Carga

1. Supongamos que el valor del mínimo previamente calculado está representado por la tupla Min=$< nodo, carga, longitud, destino >$
2. Si $destino = N$ entonces P se ejecuta localmente.
3. Sino, se inicia la transferencia de P a *destino* para su ejecución remota.

La *política de transferencia* que determina si el proceso debe ejecutarse localmente o remotamente está dada por el valor del mínimo que conoce el nodo N. Es decir, si el valor del mínimo es el que corresponde a la carga local (indica que N es el que está menos cargado en el sistema), el proceso se ejecuta localmente. Si el valor del mínimo nos indica una ubicación remota, intentaremos transferir el proceso para su ejecución en ese lugar. Por lo tanto, la *política de ubicación* ya está resuelta por el mecanismo de intercambio de información. En el momento

de buscar una ubicación no es necesario determinar a qué nodo tenemos que enviar el proceso, ya que se conoce de antemano ese lugar por la información del mínimo.

La tupla $Min{=}< nodo, carga, longitud, destino >$ es utilizada para transferir el proceso. Los identificadores de *nodo* y *destino* pueden contener información sobre el host del sitio correspondiente. En esta etapa, si el nodo de origen conoce una conexión directa con *destino*, directamente inicia la transferencia. Caso contrario, el proceso necesita primero ser enviado hacia *nodo* para luego ser transferido al destino (forwarding).

2.3. Política de limitación a la migración

Un posible problema de este algoritmo es que un proceso quede ciclando sin encontrar un nodo adecuado para ejecutarse. Para evitar esto, se propone implementar una política de limitación a la migración que entre otros factores se verá afectada por el costo que produce la transferencia de un proceso desde su nodo actual hasta uno vecino.

Antes de comenzar la transferencia, se le agrega al proceso un parámetro M para limitar su migración, que es decrementado cada vez que el proceso viaja de un nodo a otro. Si M llega a cero entonces el proceso se ejecuta en el nodo donde se encuentra en ese momento sin ser transferido nuevamente. Esto puede ocurrir, por ejemplo, porque la carga del nodo mínimo aumentó mientras se intentaba transferir el proceso, y el proceso fue transferido a un nuevo destino.

2.4. Ejemplo

En la figura 3 se muestra un ejemplo de la transferencia de información en un sistema con 7 nodos. El grafo de la izquierda representa el sistema antes que se inicie el intercambio de información de las tuplas. Las etiquetas de los nodos representan la carga local. Para simplificar el dibujo, las etiquetas de los arcos representan la carga mínima que se va transmitiendo en las tuplas. El grafo central y el de la derecha muestran como van evolucionando las etiquetas de los arcos del grafo mientras el mínimo se va propagando.

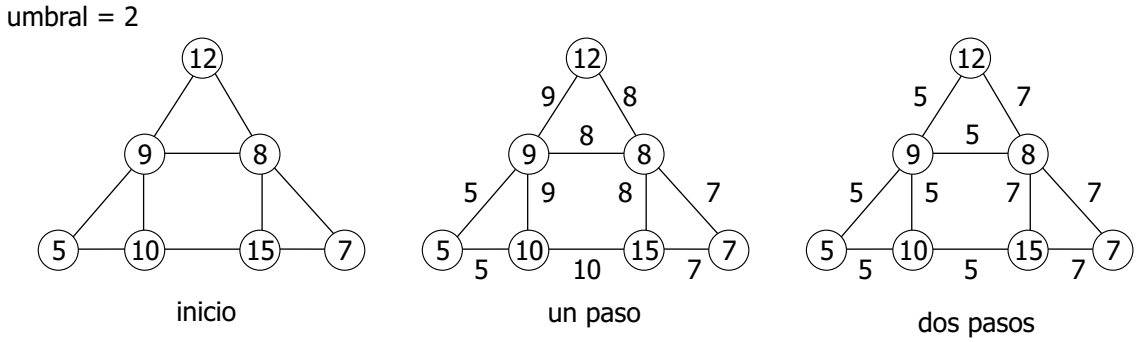

Figura 3: Propagación del mínimo

Partiendo de un estado inicial en donde los nodos todavía no tienen información de sus vecinos, se inicia el intercambio de mensajes con tuplas que contienen un valor en el campo *longitud* igual a cero, ya que se propaga sólo la carga local como el mínimo (las tablas internas están vacías).

Se puede observar que, luego de un paso, los arcos tienen como etiqueta al mínimo entre los nodos de los extremos. Como el valor de *umbral* es igual a dos, es de esperar que en dos pasos el intercambio de información se estabilice. A esa altura, los nodos ya tendrían la información del mínimo, suponiendo que no hay cambios en el sistema. Si sucediera algún cambio en la carga de alguno de los nodos, llegar a un estado estable podría requerir más pasos.

Es importante destacar cómo en el último paso de la figura existe una diferenciación entre los nodos que tienen un valor 5 de mínimo y los que tienen un valor 7, el límite lo impone el valor de *umbral*.

3. Evaluación del Algoritmo

Según la taxonomía propuesta en [1] el algoritmo propuesto cuenta con las siguientes características:

- *Global*: la planificación global se encarga de decidir dónde ejecutar un proceso. La planificación local (la asignación de time slices para ejecutar en entornos de tiempo compartido) es tarea del sistema operativo que posee el procesador que se eligió para la asignación. Esto permite a los procesadores en un multiprocesador incrementar su autonomía mientras se reduce la responsabilidad (y el overhead) de los mecanismos de planificación globales.

- *Dinámico*: una solución dinámica reacciona ante parámetros del entorno que cambian dinámicamente y que describen al estado actual del sistema, cuando debe tomar sus decisiones.

- *Sub-Óptimo y Heurístico*: se buscará que la solución sea sub-óptima para minimizar la sobrecarga en el mecanismo que toma las decisiones, y así reducir el tiempo de cómputo. Además incluye parámetros para limitar la migración y parámetros de longitud de camino, que califican en las posibilidades de elección de un nodo destino.

- *Distribuido*: la responsabilidad de la asignación dinámica de procesos no reside en un único procesador, sino que la tarea está físicamente distribuida entre los procesadores por el intercambio de información que realizan. Provee mayor confiabilidad y tolerancia a fallas.

- *Cooperativo*: se estudia un mecanismo que involucra la cooperación entre componentes distribuidos. En este caso cada procesador tiene la responsabilidad de llevar a cabo su propia porción de la tarea de planificación, pero todos los procesadores trabajan para lograr un objetivo común a lo largo del sistema (no afectar la performance global). Provee estabilidad.

- *Asignación por única vez*: a veces también referido como migración estática, en cuanto toma la decisión de dónde ejecutar un proceso cuando éste es creado. Luego, al comenzar su ejecución en un determinado nodo, el proceso ya no podrá ser migrado. La desventaja es que no se pueden corregir las asignaciones que se hayan hecho, aún cuando el entorno sea muy cambiante y se presenten mejores oportunidades. Por otro lado, una migración dinámica de procesos es muy costosa ya que hay que transferir también el estado del proceso y se debe hacer un cuidadoso análisis de cómo se va a llevar a cabo [3, 5].

En [6] se identifican una serie de propiedades deseables que debe poseer un algoritmo de balance de carga. Describimos a continuación estas propiedades analizando si nuestro algoritmo las satisface.

- *Prescindir de información 'a priori'*: los algoritmos de planificación que operan basados en la información sobre las características y utilización de recursos del proceso, normalmente imponen una responsabilidad extra en los usuarios que deben especificar esta información al remitir sus procesos para ejecución. En este algoritmo no se necesita que los usuarios ingresen información adicional sobre los procesos, sólo es necesaria la configuración inicial del grafo que puede ser realizada por los administradores del sistema.

- *Rapidez en la toma de decisiones*: un buen algoritmo de planificación de procesos debe tomar decisiones rápidas sobre la asignación de los procesos a los nodos. Esta característica vuelve inadecuadas a muchas soluciones potenciales. Los modelos heurísticos requieren un menor esfuerzo computacional y proveen resultados cercanos al óptimo. En este algoritmo no es necesario buscar repetidamente un nodo que esté dispuesto a aceptar un proceso, ya se conoce cuál es el nodo destino por los datos provistos por el mecanismo de intercambio de información.

- *Performance del sistema y overhead de planificación balanceados*: los algoritmos de planificación recolectan información del estado global y utilizan esta información para las decisiones de asignación. Mientras mayor sea la cantidad de información con la que se cuente, más inteligente podrá ser la planificación. Sin embargo, mientras mayor sea el overhead impuesto en la recolección de esta información, menor será su utilidad debido a su envejecimiento y a la menor frecuencia de las planificaciones (costo extra para buscar y procesar la información).

- *Estabilidad*: un algoritmo es inestable si puede entrar en un estado en el cual los nodos del sistema pasan su tiempo migrando procesos sin realizar ningún trabajo útil (thrashing de procesador). Esto se solucionó limitando la migración.

- *Escalabilidad*: debe ser capaz de manejar sistemas pequeños o grandes. Una aproximación simple es probar con m nodos destino posibles de los N existentes en el sistema. El valor de m puede ser ajustado dinámicamente dependiendo de N. Esto se ve reflejado en el algoritmo propuesto por la conformación del grafo y las conexiones lógicas establecidas. También influye en la escalabilidad el valor de *umbral* elegido.

- *Tolerancia a fallas*: el algoritmo debe seguir funcionando a pesar de las posibles fallas en uno o más nodos del sistema. Si los nodos se particionan en dos o más grupos por fallas en los enlaces, el algoritmo debería ser capaz de seguir funcionando correctamente dentro de cada grupo, degradando únicamente su performance. Los nodos pueden ser capaces de detectar si algún nodo vecino falló, por ejemplo, si no reciben ninguna información (mensaje de propagación de mínimo) dentro de un determinado tiempo, entonces descartan el valor de carga que conocen del nodo caído ya que éste no está disponible.

Para realizar una prueba del algoritmo propuesto decidimos utilizar Jinni (Java INference engine and Networked Interactor), un lenguaje multithreaded de programación lógica [8, 9]. Los threads de Jinni están coordinados a través de blackboards, locales a cada proceso y que fácilmente pueden ser utilizados como áreas de memoria compartida. Cada proceso Jinni que

se ejecuta emula un nodo del sistema. En esta simulación se observó que el intercambio de información se realiza de la manera esperada y que rápidamente los nodos capturan la información del mínimo. El algoritmo presenta una buena reacción ante los cambios de las cargas de los nodos. Se realizaron pruebas variando el parámetro *umbral* para cada nodo en forma independiente (cada nodo puede trabajar con valores distintos, presenta mayor flexibilidad) y se mostró estable.

4. Comparación con otros algoritmos

En [11] se presentan varios algoritmos de balance de carga dinámico. Al igual que en nuestro algoritmo, los llamados GLOBAL, CENTRAL y DISTED, conocen de antemano la carga de los demás nodos, y utilizan esta información para decidir dónde ejecutar un nuevo proceso que arriba al sistema. Los otros tres, THRHLD, LOWEST y RESERVE, consultan por la carga de los demás nodos en el momento que deciden ejecutar un proceso en forma remota. Esto condiciona la velocidad de la toma de decisiones, ya que debe esperarse hasta recibir la información de todos los nodos consultados antes de transferir un proceso.

A continuación se describen brevemente tres algoritmos de balance de carga dinámico presentados en [11]. Para cada uno de ellos se indican sus principales ventajas y desventajas, realizando una comparación con el algoritmo presentado en este trabajo.

En el algoritmo GLOBAL, cada P segundos, uno de los nodos, designado como "centro de información" recibe la información de carga de todos los nodos y la almacena en un "vector de carga", el cual es transmitido a todos los nodos. Cada nodo entonces, para elegir dónde ejecutar un proceso, busca el mínimo dentro del vector de carga. Al ser centralizado, no es tolerante a fallas y puede presentar una congestión en la red alrededor del nodo central por la frecuencia de las comunicaciones.

El algoritmo llamado DISTED, es similar a GLOBAL, pero en lugar de existir un sólo nodo como "centro de información", cada nodo realiza un broadcast con su carga a todos los demás, para que cada uno actualice su vector de carga. En este caso no hay un punto central de falla, ni un nivel extra de indirección en la distribución de la información de carga, pero la sobrecarga de transferencia de información es mayor y crece linealmente con el tamaño del sistema.

En el algoritmo llamado CENTRAL, el nodo que funciona como "centro de información", no sólo recibe la carga de los demás nodos, sino que actúa como planificador central para los demás nodos. Cuando un nodo N decide ejecutar remotamente un proceso P, N consulta al "centro de información" quien decide dónde debe ejecutarse P. El problema aquí sigue siendo la tolerancia a fallas. Pero además, existe una sobrecarga de ejecución en el nodo "central" porque debe tomar las decisiones de todas las transferencias de procesos del sistema. Según los resultados de [11] esto representa un 35 % del uso de CPU para 49 nodos.

Los algoritmos anteriores presentan dos extremos: en DISTED la información es transmitida de cada nodo a todos los demás, y en CENTRAL o GLOBAL, existe un nodo central que recibe y envía información. El algoritmo que hemos presentado en este trabajo presenta un equilibrio entre estos dos extremos: al no existir un nodo central como en GLOBAL o CENTRAL, el algoritmo es más tolerante a fallas (si un nodo se cae, el resto del sistema no depende de él), además, como la información es solamente transmitida a los nodos vecinos, no existe la sobrecarga de comunicación que presenta DISTED.

Es importante destacar, que los tres algoritmos DISTED, CENTRAL y GLOBAL, al tener acceso al vector de carga completo del sistema, pueden calcular efectivamente el nodo con la mínima carga. Sin embargo, como ya fue explicado en la sección 2.1, en nuestro algoritmo esto depende del *umbral* elegido. Si el *umbral* es 0, entonces cada nodo sólo conocerá su carga actual y no propagará información. Si el *umbral* es mayor o igual al camino más largo en el grafo que representa al sistema, entonces el mínimo se propagará a todos los nodos. Como se mostró en el ejemplo de la sección 2.4, si el *umbral* es menor al camino más largo, entonces cada nodo tendrá la información del mínimo a una distancia menor o igual al *umbral*. Como propagar la información lleva tiempo, el *umbral* garantiza que la información no se propague mas allá de una distancia a la cual llegue desactualizada.

5. Conclusiones y Trabajo Futuro

En este trabajo se desarrolló un algoritmo de balance de carga dinámico, distribuido y cooperativo que presenta varias características que son deseables en sistemas distribuidos o en paralelismo. El algoritmo no requiere información provista por el usuario, ni archivos históricos de ejecución para clasificar los tipos de procesos. Es estable y escalable mediante el ajuste de ciertos parámetros como la cantidad de nodos que puede atravesar durante la migración o transferencia inicial de proceso, y el valor del *umbral*, que regula la distancia hacia el nodo receptor e influye también en la confiabilidad de la información de carga que se conoce del resto de los nodos. Al ser distribuido, presenta tolerancia ante las fallas en los nodos o enlaces de la red, ya que no se asigna una responsabilidad diferenciada a sólo algunos nodos del sistema.

Cada nodo cuenta en todo momento con la información del nodo menos cargado que se encuentra dentro de un radio *umbral*. Al transferir sólo información relativa al nodo con mínima carga se reduce el tamaño de los mensajes enviados y recibidos. La política de selección de un nodo destino o política de ubicación, está resuelta principalmente por el intercambio de información, lo que permite reducir el tiempo que consume la toma de decisiones en cuanto a la ubicación de un proceso para ejecución remota.

Para sistemas con un gran número de nodos, el algoritmo puede adaptarse sin la necesidad de agrupar los procesadores en clusters, mediante la configuración del grafo. Las conexiones lógicas son las que deben establecerse de antemano por los administradores del sistema y pueden ser conocidas por cada nodo a través de algún archivo particular. Información sobre las conexiones físicas disponibles, o los nodos alcanzables, también puede encontrarse en tablas o en archivos de configuración del sistema.

La funcionalidad del algoritmo fue estudiada en base a sistemas homogéneos o sistemas heterogéneos en configuración, esto es, los nodos pueden diferir en su poder de procesamiento, capacidad de memoria, espacio de disco, etc. Para sistemas que cuenten con recursos compartidos disponibles únicamente en ciertos nodos, habría que hacer consideraciones sobre los recursos que necesite cada proceso antes de ser migrado a un sitio particular.

En un trabajo futuro se planea realizar simulaciones para evaluar el comportamiento del algoritmo presentado en diferentes escenarios y con distintas topologías de conexión. Se podría realizar una serie de observaciones sobre la distribución inicial de los nodos y la conformación del grafo. Si la forma del grafo es circular, los nodos que estadísticamente tienen cargas menores se ubicarían cercanos al centro, para que sean alcanzables por nodos vecinos atravesando pocos

arcos. Los nodos que están siempre más cargados, o que representan a máquinas con escasas capacidades de cómputo (memoria, CPU, ...), estarían ubicados sobre la periferia. También se pueden analizar los casos de mesh, torus, hipercubo y otros.

Referencias

[1] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[2] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1986.

[3] M. R. Eskicioglu. Design Issues of Process Migration Facilities in Distributed Systems. In *IEEE Computer Society Technical Committe on Operating Systems and Application Environments Newsletter*, volume 4, pages 3–13, 1990.

[4] D. Ferrari. A Study of Load Indices for Load Balancing Schemes. Technical Report UCB/CSD 85/262, Computer Science Division, Univ. California, Berkeley, Oct. 1985.

[5] D. Milojicic, F. Douglis, Y. Panedeine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[6] P. K. Sinha. *Distributed Operating Systems: Concepts and Design.* IEEE Press, 1997.

[7] A. S. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall, Inc., 1995.

[8] P. Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.

[9] P. Tarau. *Jinni: A Prolog Interpreter in Java for Mobile Agent Scripting and Internet Programming, USER GUIDE*, July 1999.

[10] S. Zhou. An Experimental Assessment of Resource Queue Lenghts as Load Indices. In *Proc. Winter USENIX Conf.*, pages 73–82, Washington DC, Jan. 1987.

[11] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, 1988.

[12] S. Zhou, J. Wang, X. Zheng, and P. Delisle. UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software - Practice and Experience*, 23(12):1305–1336, 1993.