Universidad Nacional del Centro
de la Provincia de Buenos Aires

Computación Paralela y Distribuida

Sistema distribuido para el procesamiento de tareas

Realizado por **Martín Cordischi**

UNICEN - Junio 2013

Resumen

Como trabajo final de la cátedra de Computación Paralela y Distribuida, se ha realizado un sistema distribuido para el procesamiento de tareas de cualquier tipo sin dependencia entre ellas. El framework podrá ser utilizado para la realización de sistemas grid, dejando transparente todo el trabajo necesario para conexión de nodos, interacción entre ellos, y ejecución de tareas.

Este informe dará las nociones generales del sistema, notificará las principales decisiones adoptadas y detallará los elementos de interés para el funcionamiento del framework. Este documento se organiza comenzando con una introducción y detallando los componentes principales del sistema; siguiendo por una explicación de la correcta utilización de estos, con un historial de cambios y las tareas a realizar; culminando por una conclusión y discusión.

Introducción

En la actualidad existen grandes sistemas implementados para el procesamiento distribuido de tareas. Un ejemplo de esto el el grid World Community Grid[1], o Seti@Home[4]. Cada uno de ellos tiene sus peculiaridades, dadas por el tipo de tarea a realizar, el propósito de la red y la característica de sus nodos, entre otros.

La cátedra de Computación Paralela y Distribuida ha propuesto realizar un sistema de procesamiento distribuido con control no centralizado y con la capacidad de robos entre los consumidores de tareas. Este proyecto tendrá como objetivo lograr un sistema flexible y con buena usabilidad para el programador que desee utilizarlo para proyectos reales o de investigación.

Desarrollo

Nociones generales del sistema

El objetivo principal del trabajo es la posibilidad de brindar procesamiento distribuido a tareas no dependientes a través de un sistema no centralizado, utilizando un modelo productor/consumidor. Como requerimiento funcional también se ha solicitado la posibilidad de establecer políticas de robo de trabajos entre los nodos consumidores.

Para llegar a cumplir los requerimientos, se ha desarrollado una aplicación en Java[2] la cual es capaz de recibir y ejecutar tareas brindadas por el usuario a través de una interfaz sencilla, utilizando la potencia del procesamiento distribuido.

El sistema desarrollado utiliza las facilidades brindadas por la biblioteca JGroups[3] para abstracción de la comunicación entre nodos. Esta herramienta ha traído importantes ventajas para la correcta implementación y ha satisfecho completamente la necesidad de tener un módulo de comunicación.

Características técnicas

- **Tipo de sistema:** Framework.
- **Propósito:** Procesamiento distribuido de tareas.
- **Lenguaje:** Java[2].
- **Bibliotecas externas:** Jgroups[3].
- **Código fuente:** *github.com/tinchofm/Copay-di/*

Descripción del sistema

El sistema está formado por 3 tipos de componentes principales:

- **Nodos:** principales componentes del sistema, encargados de la conexión, la ejecución de tareas y la interacción con el usuario. Forman en conjunto un clúster.
- **Tareas:** elementos que se sirven de entrada y salida al sistema.
- **Mensajes:** componente completamente interno, encargados de contener la información transmitida entre los nodos.

Estos 3 tipos de componentes definen el sistema. Como soporte a ellos, también se definen

- **Políticas:** empaquetadas en elementos del sistema, sirven para determinar comportamientos de programación (scheduling) y de robos.
- **Eventos:** notificaciones de los nodos.

A continuación, se detallarán los elementos principales del sistema y sus funciones.

Nodos

Como se puede observar en la figura 1, los nodos son los elementos con mayores responsabilidades ya que deben establecer comunicaciones con sus pares, seguir el estado del clúster e insertar y/o ejecutar tareas.

Los nodos pueden corresponderse uno a uno con máquinas en un clúster, pero es necesario destacar que pueden existir y convivir más de un nodo por terminal, siendo así el caso en el que se desee procesar, insertar y/o monitorear tareas, teniendo un tipo de nodo distinto por propósito.

El sistema define 4 tipos de nodos, mostrados en la figura 2, los cuales están implementados a través de la jerarquía de clases mostrada en la figura 3. Las principales responsabilidades de cada tipo de nodo son:

- **Master:** Encargado de insertar tareas al clúster, este tipo de nodo devuelve resultados mediante la interfaz *Future*, siendo así necesaria su existencia siempre que queden tareas pendientes.
- **Slave:** consumidor de las tareas. Una vez conectado al clúster, es capaz de buscar y ejecutar tareas. Las distintas clases implementadas proponen distintas políticas de robo, las cuales serán detalladas más adelante.

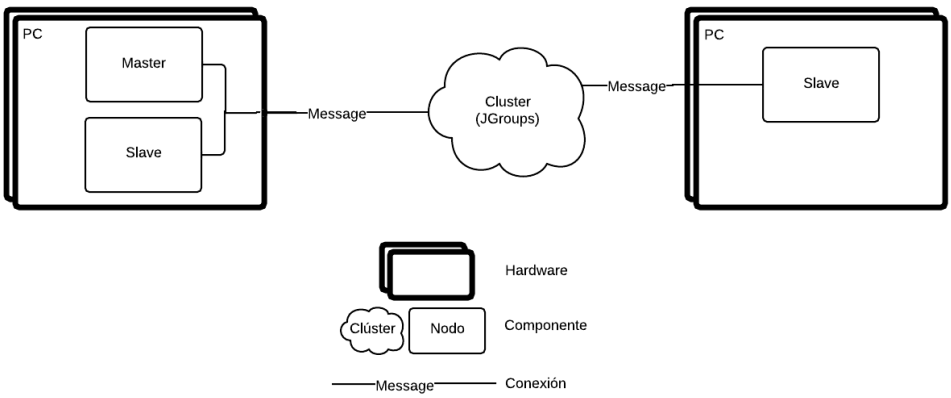

Figura 1: *Ejemplo de deployment del sistema. Una misma computadora puede contener varios nodos, la comunicación entre nodos se realiza mediante mensajes utilizando el clúster provisto por JGroups.*

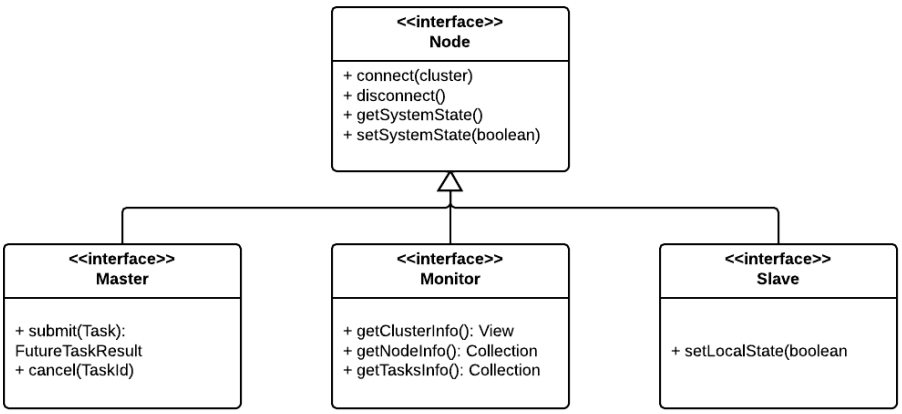

Figura 2: *Jerarquía de interfaces de Nodos*

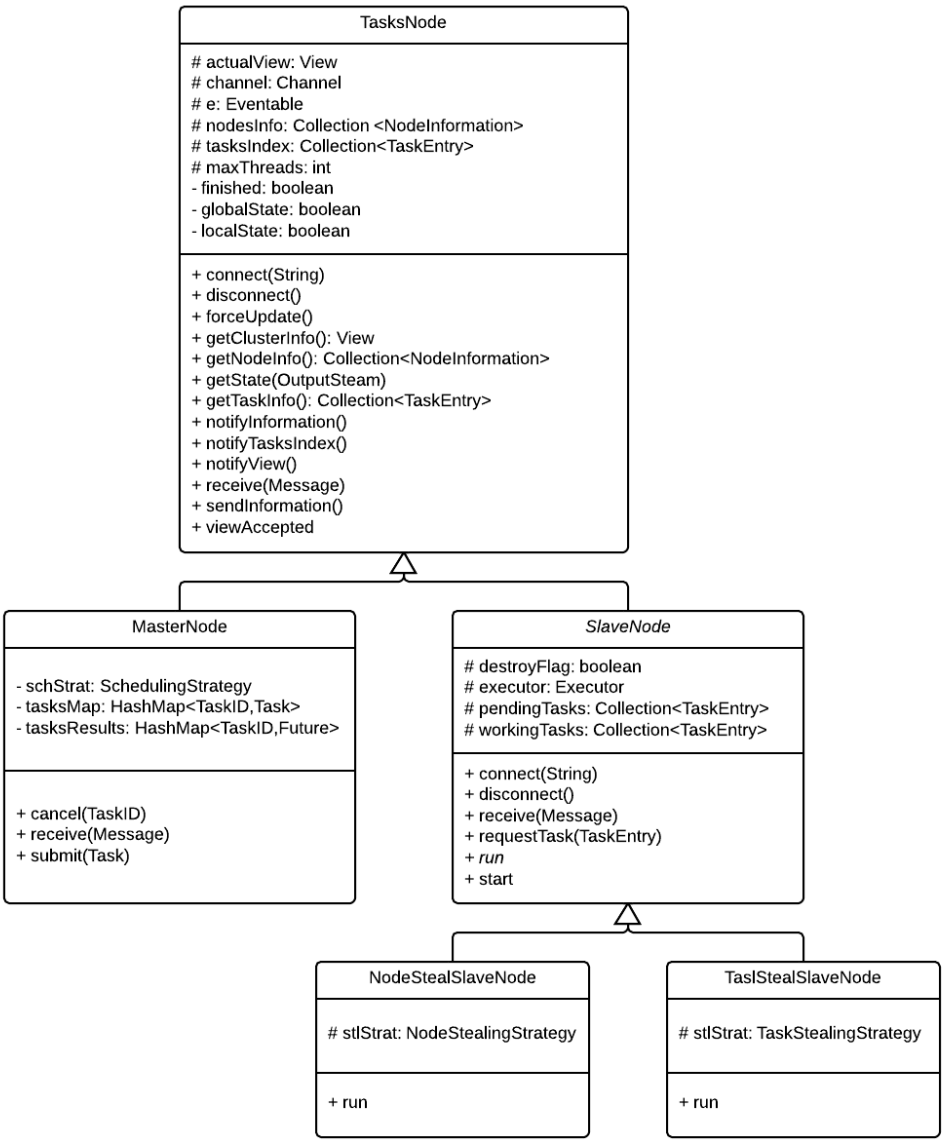

Figura 3: *Jerarquía de clases de Nodos. Se muestran solo métodos públicos, ignorando getters y setters.*

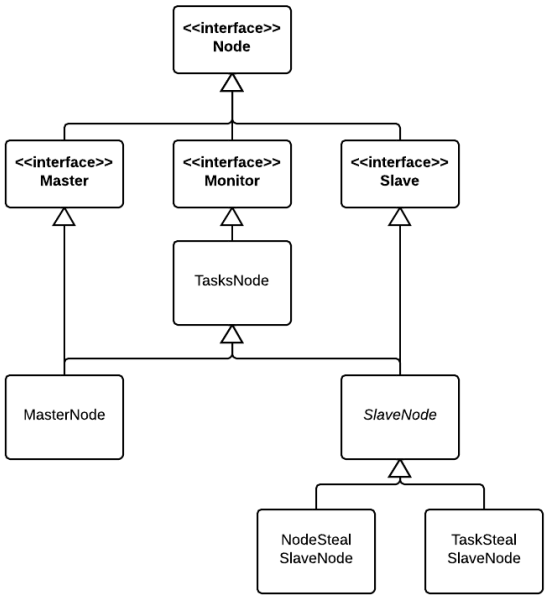

Figura 4: *Jerarquía de clases e interfaces de Nodos. Se ignoran elementos externos al sistema*

- **Monitor:** interfaz no implementada en su totalidad, pero diseñada para el acceso a la información detallada sobre el clúster. Esta entidad provee la posibilidad de obtener datos detallados sobre las tareas y los nodos conectados.

La jerarquía completa de clases y interfaces de nodos se puede observar en la figura 4.

Todos los nodos poseen un estado global del clúster, el cual incluye la lista de todos los nodos y su información, a través de la interfaz *NodeInformation*; y los datos básicos pertenecientes a todas las tareas que están trabajando, pendientes o finalizadas a través de la clase *TaskEntry*.

Tareas

Las tareas están provistas como una interfaz contenedora llamada *Task*, cuya ejecución será a través del método *call()*. Estas devolverán un resultado que se podrá acceder a través del *Future* generado por el *Master*. Cualquier tipo de excepción o error será notificado a través del mismo medio.

Como se puede observar en la figura 5, una vez que se le solicita una tarea al *Master*, este solo notificará a todo el clúster sobre la novedad, pero no enviará la tarea en sí hasta no recibir una solicitud de ejecución de un

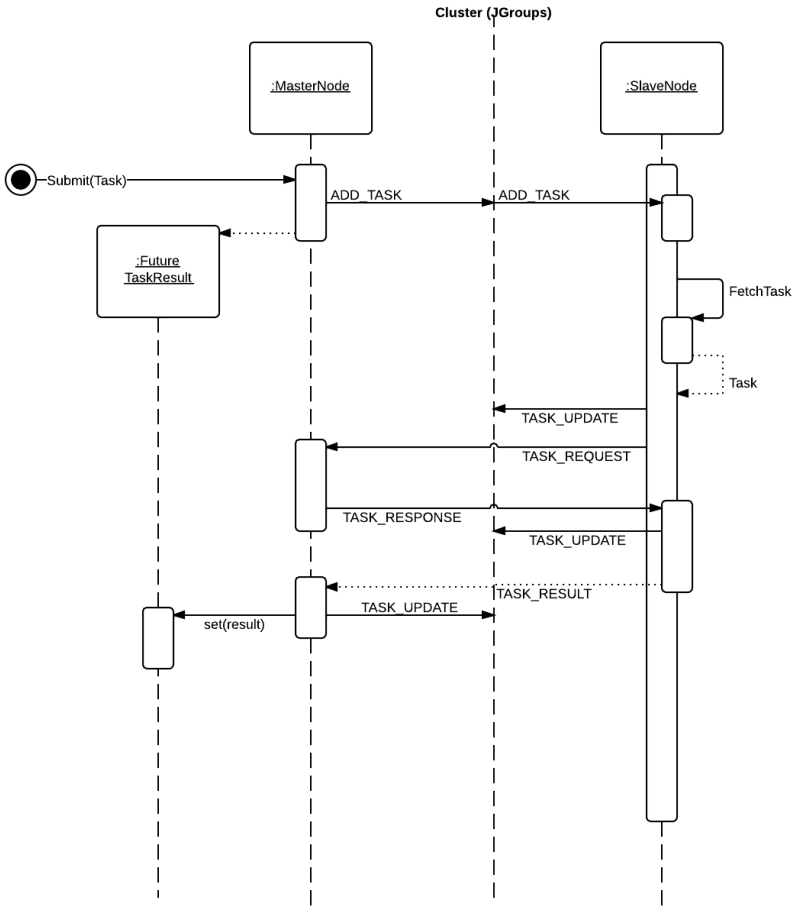

Figura 5: *Diagrama de secuencia mostrando el funcionamiento normal del clúster*

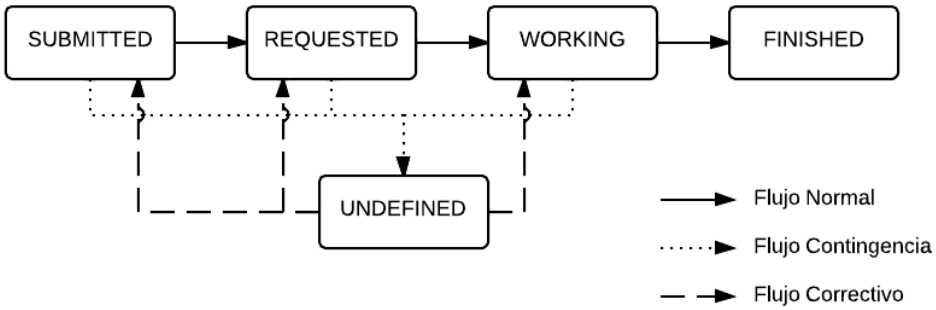

Figura 6: *Posibles estados de una tarea*

Slave. Una vez enviada la tarea al nodo responsable, solo quedará ejecutarla y devolver el resultado o excepción.

La información que poseen todos los nodos sobre cada tarea está encapsulada dentro de *TaskEntry*, la cual contiene el estado actual de la tarea, el identificador único, el dueño y el actual *Handler*. Estos estados por los que pasa cada tarea se puede observar en la figura 6.

Las tareas serán ejecutadas en un nodo, el cual será determinado inicialmente por las políticas de scheduling del *Master*, con posibles variaciones debido a los robos. Estas características deberán ser establecidas y desarrolladas por el programador que desee utilizar el sistema.

Cabe destacar que también se ha implementado una subclase especial de tarea, la cual permite que la misma se pause ante necesidades especiales del *Slave*. De todas formas, hasta el momento el sistema no aprovecha esta característica.

Mensajes

Internos al sistema y completamente transparente para el usuario, los datos enviados a través del clúster son realizados a través del paquete de clases de *Message*.

JGroups otorga una comunicación confiable y multicast, por lo que no se tuvo que trabajar sobre los envíos de mensajes en sí, dejando solo las preocupaciones en los datos enviados.

Políticas

Tanto el scheduling inicial del *Master* como la elección de víctimas del *Slave* que roba necesitarán una política determinada. Mediante el patrón de diseño **Strategy**, las políticas podrán ser definidas por el programador y cambiadas en tiempo de ejecución. Los tipos de política son:

- **SchedulingStrategy:** Utilizado por el *Master* y ejecutado cada vez que se inserte una nueva tarea al clúster, el algoritmo de scheduling deberá determinar un nodo que ejecute la nueva tarea en base a información sobre estos.
- **StealingStrategy:** cuando un *Slave* termina sus tareas, puede aplicar un algoritmo para seleccionar una tarea o nodo como víctima. La información utilizada por estas políticas será información sobre nodos o tareas. Se debe aclarar que si no se define políticas de robo, el *Slave* nunca robará, convirtiéndose así en un consumidor reactivo.

La forma de utilizar las políticas será detallada en la sección *Utilización y Aplicaciones*.

Robo de Nodos

La clase *NodeStealSlaveNode* aplicará la política de robos a nivel de nodos. Cuando se quede sin tareas por procesar, el *Slave* elegirá una víctima de la lista de nodos, y le enviará una solicitud de robo junto a un número de tareas deseadas (el cual es igual a la cantidad máxima de tareas que se pueden ejecutar en simultáneo por éste). Cuando un *Slave* recibe una solicitud de robo, analizará sus tareas pendientes y directamente cambiará el *Handler* de las tareas, notificando de ésto a todo el clúster. Esta situación se puede observar en la figura 7.

Robo de Tareas

El robo de tareas ocurrirá bajo la misma situación que el robo de nodos: cuando en este caso el *TaskStealSlaveNode* se quede sin tareas por procesar. A diferencia con el robo de nodos, el nodo seleccionará una tarea víctima a través del algoritmo dentro de *TaskStealingStrategy* y la robará sin solicitud, cambiando el *Handler* por él mismo y notificándoselo al clúster. El funcionamiento de robo de tareas está en la figura 8

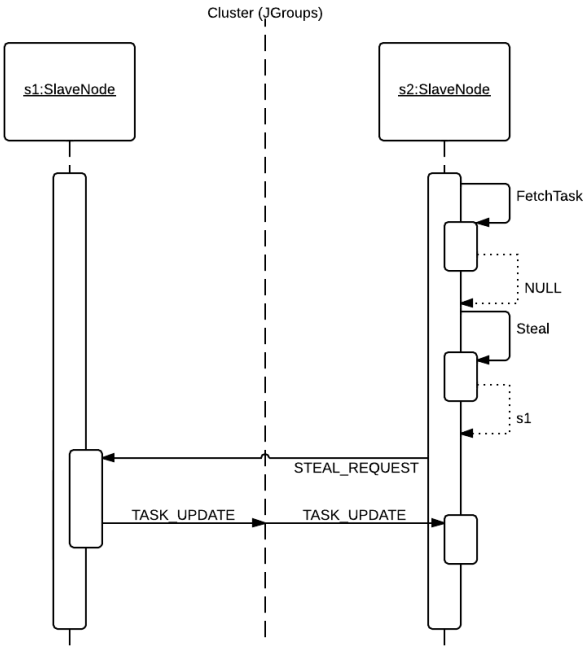

Figura 7: *Diagrama de secuencia mostrando el robo a nivel nodo.*

Eventos

Los nodos informarán cambios en los estados de las tareas y del clúster en general a través de la interfaz *Eventable*, la cual podrá ser utilizada si se requiere adquirir información detallada.

Utilización y aplicaciones

Para poder utilizar este sistema se deberá tener en cuenta ciertos aspectos, los cuales permitirán el desarrollo de un grid personalizable en cuanto a topología, comunicación, tipo de tareas, scheduling y robo de tareas.

Topología

Como ya se explicó, el clúster podrá ser dinámico y ya incluye un soporte ante eventuales desconexiones. Mediante los métodos *connect(cluster)* y *disconnect()* de la interfaz *Node*, el sistema manejará todas las responsabilidades ligadas con la conexión; y las caídas de los nodos de la red serán notificada a través de la interfaz *Eventable*.

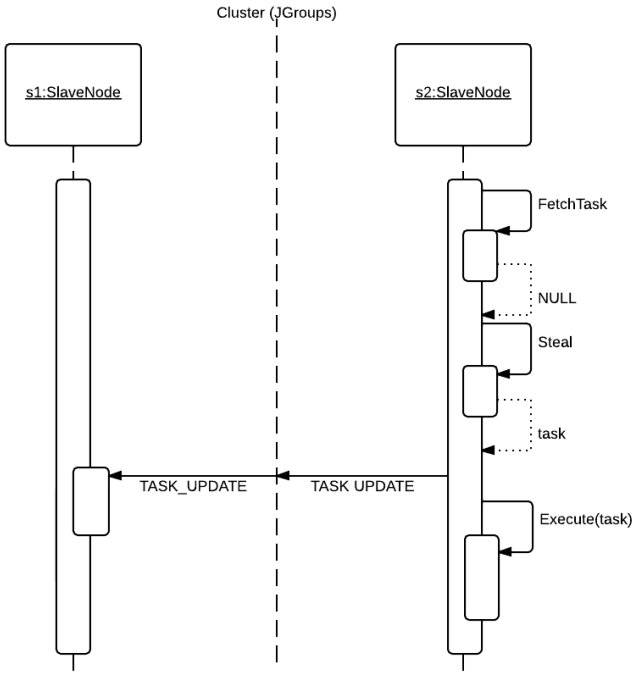

Figura 8: *Diagrama de secuencia mostrando el robo a nivel tarea.*

Comunicación

JGroups[3] otorga la posibilidad de configurar la pila de protocolos de comunicación a necesidad del usuario. El sistema actualmente utiliza los valores por default, para cambiarlos leer la documentación dentro de la página web de JGroups.

Tipos de tareas

Las tareas dentro del grid se manejan a través de una interfaz sencilla llamada *Task*, por lo que cualquier sistema que se implemente deberá utilizar esta interfaz para poder ingresar exitosamente dentro del clúster las tareas, dejándole así total libertad al programador sobre el tipo de tarea a implementar.

Al ser transmitida a través del clúster, *Task* hereda de la interfaz *Serializable*, por lo que la clase implementada deberá contener elementos serializables.

Scheduling

El *Master* se encargará de asignar un *Handler* a las tareas cada vez que se inserte una, o que haya una caída de un nodo *Slave*. El algoritmo de selección se realizará a través de la interfaz *SchedulingStrategy*, por lo que el programador deberá implementar una clase de scheduling.

Se deberá tener en cuenta que la lista pasada por parámetro **NO** se deberá editar, y que la lista de nodos contiene a **todos** los nodos del clúster, mientras que el retorno deberá ser de un nodo *Slave*; de no respetarse esto, las tareas nunca se ejecutarán.

Robo de tareas

Como ya se detalló en la sección *Descripción del Sistema*, existen dos tipos de *Slave* con estrategias distintas en cuanto a la forma de robo.

- **Robo a nivel Nodo:** utiliza el algoritmo dentro de *NodeStealingStrategy* para seleccionar una víctima y enviarle un pedido de robo. Al necesitar pedidos, este tipo de robo es completamente seguro y no presenta problemas con la sincronización.
- **Robo a nivel Tarea:** utiliza *TaskStealingStrategy* para seleccionar una tarea víctima y empezar la ejecución de la misma, notificando al clúster. A través de este método se pueden producir problemas de sincronización dentro del clúster, provocando cálculos innecesarios. Por ejemplo cuando dos nodos roban la misma tarea al mismo tiempo, provocará que la misma tarea se ejecute dos veces. Posibles soluciones se propondrán en la sección *Tareas a realizar*.

Distintos tipos de *Slaves* pueden convivir bajo el mismo clúster. Para utilizarlos, se deberá tener en cuenta que los datos pasados como parámetros de los *Strategy* **NO** deberán ser alterados, y que las estructuras contienen información sobre **todos** los nodos o tareas del clúster. Los algoritmos deberán retornar un nodo *Slave* para el caso de robo de nodos, y una tarea no finalizada para el caso de robo de tareas.

En los códigos fuente se encuentran una política implementada para cada tipo: *FirstNodeStealingStrategy* y *RandomStealingStrategy*.

Historial de cambios

Esta sección explicará las grandes modificaciones realizadas a lo largo de la vida del proyecto junto a las debidas justificaciones, hasta llegar al sistema

actual.

El objetivo de esta sección es poder comprender el proceso realizado, logrando así un mayor entendimiento del estado actual.

Una sola clase: *Peer*

En un principio, el sistema se valió solamente de una clase implementada: *Peer*, la cual contenía las responsabilidades de un *Master-Slave* y recomendando el uso a través de las interfaces provistas. De esta forma se lograba que todos los nodos dentro del clúster puedan procesar tareas, y simplicaba el entendimiento del código fuente gracias a métodos comunes y extendidos.

Pero el crecimiento de la clase, y el agregado de funcionalidades específicas para cada tipo de nodo hizo que se necesitara rediseñar el sistema, llegando a la jerarquía de clases actual.

Robos

Al iniciar el desarrollo, se decidió implementar solo el robo de tareas. Cuando se comenzó el testeo, se descubrió el problema ligado con la sincronización ya explicado, lo cual ocurría con frecuencia cuando la cantidad de tareas era pequeña y los *Slaves* tenían la misma potencia de procesamiento.

A partir de este problema, se decidió cambiar el tipo de robo a uno más seguro en cuanto a sincronización, desarrollando así el de nodos. De todas formas se mantuvo el robo de tareas dentro del sistema para otorgar mayor flexibilidad al programador.

Los cambios realizados en el tipo de robos llevó a la necesidad de implementar un sistema de información sobre nodos más detallado que el brindado por JGroups. De allí surgió la clase *NodeInformation*.

Consumidor reactivo a consumidor activo

Dentro de la clase *Slave*, la búsqueda de tareas se implementó en un principio de forma reactiva: cada vez que el estado de tareas se cambiaba se iniciaba la ejecución de búsqueda de tareas. Este método podía resultar pesado para la creación de hilos cuando el estado de tareas cambiaba continuamente, y podía derivar en problemas de sincronización, por lo que se decidió cambiar a un consumidor que continuamente chequee ante posibles nuevas tareas.

De esta forma, se evitó el bloqueo de las llamadas asincrónicas generadas por JGroups, lo cual podía terminar en el bloqueo de todo el clúster. También se solucionaron posibles problemas de sincronización y se logró que el nodo esté obligado a robar cada un tiempo determinado.

Tareas a realizar

Como todo proyecto, el proceso aún se encuentra inconcluso. En esta sección se nombrarán los próximos pasos a realizar para seguir en la optimización del sistema.

- **Utilización de *NodeInformation*:** actualmente la información de nodos no se actualiza de forma automática, sino que solo se realiza ante un pedido forzado por parte de un nodo o del usuario. Se puede mejorar este aspecto brindando una actualización regularmente, y aprovechando este componente para adquirir datos estadísticos y detallados sobre los nodos.
- **Parámetros sobre políticas de robo y scheduling:** por el momento, los nodos delegan las responsabilidades de selección a las estrategias, pero lo realizan brindándoles información sensible y completa sobre el clúster. Se deberá limitar los datos brindados.
- **Evitar problemas de sincronización en robos:** se deberá buscar una solución ante posibles problemas en los robos de tareas. Por el momento, alternativas analizadas como utilización de *timestamp* en los mensajes o control centralizado de las tareas son difíciles de controlar.

Conclusión y discusión

La realización de este trabajo me ha sido de gran utilidad para la comprensión del funcionamiento de sistemas distribuidos y las políticas de robos. Aunque es un trabajo que requerirá continuas mejoras, espero que este sistema sea de utilidad para futuras investigaciones o aplicaciones.

Bibliografía

[1] World community grid. http://www.worldcommunitygrid.org/.

[2] Java. http://java.com/.

[3] JGroups. http://www.jgroups.org.

[4] SETI@Home. http://setiathome.berkeley.edu/.