UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

MAPEO E INTERCONEXIÓN DE PROCESOS EN UN CLUSTER DE COMPUTADORAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
ANDRÉS ALDANA GONZÁLEZ

DIRECTOR DE TESIS:
DR. JORGE LUIS ORTEGA ARJONA

2012

Hoja de datos del jurado

1.Datos del alumno
Aldana
González
Andrés
58 79 88 78
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
301316857

2. Datos del tutor
Dr.
Jorge Luis
Ortega
Arjona

3. Datos del sinodal 1
Dr.
Héctor
Benítez
Pérez

4. Datos del sinodal 2
Dr.
José David
Flores
Peñaloza

5. Datos del sinodal 3
Dr.
Sergio
Rajsbaum
Gorodezky

6. Datos del sinodal 4
Mat.
Salvador
López
Mendoza

7. Datos del trabajo escrito
Mapeo e interconexión de procesos en un cluster de computadoras
207 p
2012

A mi Mamá,
por su esfuerzo,
dedicación y paciencia.
Porque esta tesis también es de ella.

A Max,
de quien aprendí
el amor a la ciencia.

A Claudia,
que siempre me dio ánimos.

A los que siempre creyeron en mí.

Índice general

Agradecimientos

La realización de este trabajo ha sido posible gracias a la colaboración y apoyo de muchas personas con las cuales estoy profundamente agradecido.

Quisiera empezar agradeciendo infinitamente a mi asesor, el Dr. Jorge Luis Ortega Arjona, quien propuso este tema de tesis y ha sido una fuente de inspiración a lo largo de mi formación universitaria. Como mi profesor y como mi asesor de tesis, el Dr. Ortega me ha apoyado y ayudado en inumerables ocasiones, compartiendo conmigo su conocimiento y experiencia en temas incluso fuera del ámbito académico.

También quiero agradecer a mis sinodales, el Matemático Salvador López Mendoza y los doctores Héctor Benítez Pérez, José David Flores Peñaloza y Sergio Rajsbaum Gorodezky, cuyos comentarios y observaciones enriquecieron y mejoraron este trabajo, se tomaron el tiempo y paciencia para corregir y explicarme todos los errores que cometí en su elaboración y me permitieron aprender más sobre mi tema de tesis.

Agradezco a los profesores y alumnos de la UNAM, quienes me formaron como profesionista y en quienes pude encontrar apoyo y conocimiento. Son ellos los heroes de la educación, pues siempre están dispuestos a ayudar, corregir y aconsejar a los compañeros y alumnos que forman parte de esta institución, compartiendo su sabiduría y enriqueciendonos tanto en el ámbito académico como en el personal. Formar parte de esta universidad es un orgullo y uno de los logros más grandes en mi vida.

No tengo palabras para agradecer lo suficiente a mi mamá, Catalina González Espinoza, quien siempre me ha querido y apoyado incondicionalmente, tanto económica como anímicamente, y en quien siempre he encontrado admiración y respeto inagotables. Sin su apoyo, este trabajo no hubiera sido posible, pues con su trabajo, fue ella quien me dio el tiempo y espacio para realizar mis estudios durante toda mi trayectoria académica. Su dedicación para trabajar, desvelarse,

levantarse temprano y proveerme con todo lo necesario para hacer una carrera, hacen que esta tesis también sea de ella.

Agradezco mucho a mis hermanos Max y Claudia. Max me enseño desde la infancia el amor por la ciencia, la importancia de realizar una carrera, de formar parte de la UNAM y de dedicar mi vida a crecer tanto a nivel intelectual como personal, transportando los conocimientos aprendidos día a día hacia los demás ámbitos de mi vida. Ha sido mi maestro al enseñarme matemáticas y ayudarme a entender temas científicos, mi amigo al aconsejarme sobre mis errores y mi hermano al quererme y apoyarme cuando lo he necesitado. Claudia siempre ha estado cerca para ayudarme y apoyarme, sobre todo moralmente, pues su calidad humana siempre me ha impulsado a seguir adelante. Incluso en los momentos más difíciles, ella siempre ha estado presente para hacerme crecer como persona.

Finalmente, deseo agradecer al resto de mis compañeros y amigos que han participado de forma indirecta en la elaboración de este trabajo. Particularmente, a Noemí, cuyos consejos siempre fueron tomados en cuenta y ayudaron a tomar decisiones importantes tanto en la elaboración de este trabajo de tesis como en la forma de abordar problemas cotidianos, así como a Felipe, quien siempre está dispuesto a compartir su tiempo y conocimiento y con quien he descubierto nuevas técnicas para resolver problemas.

A todas estas personas que hicieron este trabajo posible: ¡¡Muchas gracias!!

Andrés Aldana González

Introducción

Contexto

En esta tesis nos enfocamos en el "Procesamiento paralelo en un ambiente de memoria distribuida" en una red local (cluster) de computadoras. Dado que un proceso se basa en un algoritmo y un conjunto de datos [17, 20], en el procesamiento paralelo un proceso completo puede ser dividido en varios sub-procesos ejecutados en varias computadoras conectadas entre sí. Esto se logra dividiendo al algoritmo completo en sub-algoritmos ejecutados en distintas computadoras o dividiendo el conjunto de datos entre varios algoritmos similares distribuidos entre las computadoras de la red [12].

La aproximación depende del problema particular que se desea resolver. Sin embargo, para llevar a cabo estas tareas, han surgido dos problemas fundamentales [3]:

- Distribuir (mapear) los procesos a las distintas computadoras del cluster
- Transmitir datos entre los procesos mapeados

Estos dos problemas representan una parte muy importante en el diseño de algoritmos paralelos distribuidos y se han construido diversas técnicas para abordarlos. Estas técnicas van desde la implementación de dispositivos de hardware como los "Transputers" hasta la implementación de utilidades de software como "MPI" y "PVM".

Nosotros abordamos estos dos problemas mediante la implementación de una biblioteca a la que llamamos J-MIPS (Java - Mapeo e Interconexión de Procesos de Software) desarrollada en el lenguaje de programación Java, version 1.6. A lo largo de este trabajo, describimos principalmente los aspectos de diseño de esta biblioteca y algunos aspectos de programación paralela distribuida importantes que permitan la comprensión y uso de J-MIPS.

Planteamiento del problema

Como describimos antes, en este trabajo vamos a solucionar el siguiente problema:

> Dado un conjunto de procesos que residen en una sola computadora de un cluster y un mapeo de los procesos hacia otras computadoras del mismo cluster, distribuir los procesos a las computadoras de acuerdo al mapeo y permitir la transferencia de información entre ellos para poder ser ejecutados paralelamente en el cluster.

La solución de estos problemas facilitará la implementación de programas paralelos distribuidos.

Solución: Biblioteca J-MIPS

La solución al problema planteado en la sección anterior consiste en desarrollar una biblioteca en el lenguaje de programación Java, que permita el mapeo de procesos en una red de computadoras, así como la transmisión de datos entre dichos procesos. Esta biblioteca debe permitir al usuario especificar la topología virtual de la red, independientemente de la topología física implementada. Además las siguientes consideraciones deben ser cubiertas:

- Las computadoras deben representarse de manera univoca en la red.
- El usuario debe ser capaz de especificar el proceso que cada computadora ejecuta (mapeo de procesos).
- El usuario debe ser capaz de definir las conexiones entre procesos.
- El usuario debe ser capaz de especificar conexiones síncronas y asíncronas entre los procesos.
- Se debe permitir la transmisión de cualquier objeto finito a través de la red.

Llamamos a esta biblioteca J-MIPS (Java Mapeo e Interconexión de Procesos de Software). El diseño intenta facilitar al programador el mapeo de procesos entre computadoras, así como la transmisión de datos entre dichos procesos. La solución que proponemos permite que el programador ejecute un sistema paralelo en dos pasos:

- Implementación de la topología virtual de la red.
- Especificación de los procesos (y transmisión de datos entre ellos) para cada computadora de la red.

La independencia de estos dos puntos, permite al programador enfocarse en una sola tarea a la vez.

El primer punto habilita al usuario para definir las conexiones entre procesos por medio de software, independientemente de la topología física implementada para la red. Esto permite la implementación virtual de cualquier topología de red posible montada sobre la red física real. Más aún, está permitida la implementación de una red virtual que contiene más computadoras de las que existen físicamente. Entre otras cosas, esta característica facilita las pruebas de ejecución y depuración de los programas implementados. Además, es posible implementar más de una red virtual sobre una sola red física.

El segundo punto requiere que cada computadora virtual de la red "conozca" el trabajo que debe realizar. Es decir, el usuario debe especificar el proceso que cada computadora ejecuta en la red. J-MIPS intenta abstraer al programador de la complejidad de las primitivas de comunicación, permitiéndole enviar y recibir datos (Objetos) hacia y desde las computadoras de la red virtual, únicamente especificando el dato que debe ser transmitido y la localización de la computadora a la que debe ser enviado o de la que debe ser recibido. Este intercambio de datos se lleva a cabo mediante llamadas a métodos simples, similares a los populares "set" y "get"[6].

Contribuciones

Este trabajo está orientado hacia el desarrollo de programas paralelos distribuidos. Creemos que la biblioteca J-MIPS puede ser muy útil para este propósito. En secciones posteriores profundizaremos en el estudio del trabajo relacionado con este tema. Sin embargo, podemos adelantar que hasta donde sabemos, ninguna biblioteca o API diseñada con el mismo propósito combina al mismo tiempo el manejo lógico de las conexiones entre procesos (implementación de redes virtuales) y el nivel de portabilidad y flexibilidad que alcanza J-MIPS. En parte, este nivel de portabilidad se debe a un aspecto muy importante en la arquitectura del lenguaje Java: La Java Virtual Machine[6].

Elegimos Java como el lenguaje de programación para desarrollar J-MIPS debido a dos características del lenguaje: Su creciente expansión en la comunidad de desarrollo, lo cual da lugar a un cada vez mas amplio uso de este lenguaje, y su portabilidad [6].

El criticado uso de la máquina virtual [1] para ejecutar instrucciones de programación tiene la gran ventaja de ser portable. De esta forma, para ejecutar programas que utilicen J-MIPS basta con instalar en la computadora huesped el entorno de ejecución de Java: el JRE, a partir de la versión 1.5. La heterogeneidad de la máquina virtual de Java, permite ejecutar J-MIPS en cualquier sistema operativo de cualquier arquitectura sin necesidad de recompilar la biblioteca, siempre y cuando se encuentre disponible el entorno de ejecución JRE.

Creemos también que el enfoque del diseño de J-MIPS (el cual discutiremos en la sección 5) permite un desarrollo de aplicaciones más orientado a la ejecución local de cada computadora de la red virtual (la tarea que cada una debe realizar) que a la ejecución de la red como un todo. Esto presenta ventajas y desventajas dependiendo del problema a solucionar.

Estructura de la tesis

- El capítulo 1 (Antecedentes) profundiza en algunos temas relacionados con el trabajo realizado y contiene los conceptos e ideas necesarias para entender por completo el problema a tratar.

- En el capítulo 2 se abordan algunos trabajos relacionados como PVM y MPI, los cuales contienen soluciones similares a la propuesta en esta tesis.

- En el capítulo 3 se tratan todos los aspectos de diseño de la biblioteca J-MIPS. Se inicia describiendo el problema a resolver, así como una idea general de la solución que proponemos a dicho problema. Después se detalla el diseño de todos los componentes que intervienen en nuestra solución, así como las interacciones entre ellos.

- El capítulo 4 contiene el desarrollo de la aproximación propuesta en esta tesis, utilizando el lenguaje de programación Java. Podemos decir que este capítulo contiene la implementación de los componentes vistos en el capítulo 4. Además, la sección 4.8 (Puesta en marcha) puede ser vista como un manual de referencia para quien se interese en utilizar J-MIPS.

[1] La creencia popular es que *Java es un lenguaje interpretado y por lo tanto resulta demasiado lento para aplicaciones serias*. En [6] se señala esta creencia como un error.

- El capítulo 5 ilustra el uso de la solución propuesta mediante la implementación de dos ejemplos concretos de ejecución paralela. A su vez, esta sección puede ser vista como un tutorial de la biblioteca.

- Finalmente, se encuentran las conclusiones del trabajo realizado, donde se mencionan algunos problemas y limitaciones de la biblioteca diseñada y que pueden considerarse como la continuación natural del desarrollo de este trabajo.

Capítulo 1

Antecedentes

La biblioteca que desarrollamos en este trabajo está pensada para ejecutar aplicaciones paralelas en un ambiente de memoria distribuida. Para ello, es necesario introducir al lector en las características de los sistemas distribuidos y las redes de computadoras, así como en las técnicas y problemas relacionados con el diseño de programas paralelos. No profundizaremos en el hardware que hace posible la comunicación entre las computadoras de una red, los protocolos utilizados o los algoritmos de enrutamiento, pues como mencionamos, nuestro interés es desarrollar una herramienta que sirva para implementar aplicaciones de software que se ejecuten en un clúster de computadoras. Sin embargo, es necesario hablar sobre algunos aspectos de la red más utilizada en la actualidad para el computo paralelo con memoria distribuida: La Ethernet [15]. Describimos estos aspectos en las siguientes secciones.

También abordamos algunos aspectos teóricos relacionados con el diseño de programas paralelos, como los patrones arquitectónicos y los problemas que surgen al intentar mapear e interconectar procesos paralelos en un clúster. La comprensión de estos temas, ayuda al lector a entender el diseño de la biblioteca J-MIPS, tratado en el capítulo 3, así como los ejemplos de aplicación mostrados en el capítulo 5.

1.1. Sistemas distribuidos

En las primeras décadas de los sistemas de cómputo (1945-1985), éstos eran altamente centralizados. Generalmente, el "sistema de cómputo" consistía en una sola computadora que ocupaba todo un cuarto acondicionado especialmente para ella. En la mayoría de los casos, las instituciones tenían solamente algunos cuantos sistemas de este tipo que cumplían por separado todas las necesidades de la organización [23, 22].

Actualmente, la fusión entre las computadoras y las telecomunicaciones ha permitido reemplazar este esquema por uno en el cual un gran número de computadoras independientes conectadas entre sí comparten información para resolver una tarea en común. A este esquema se le llama *"Red de computadoras"*[22, 23].

Dos avances tecnológicos hicieron esto posible: El primero fue el desarrollo de microprocesadores poderosos y baratos. Hasta antes de la segunda mitad de la década de 1980, los microprocesadores eran máquinas de 8 bits, eran muy costosos y ocupaban mucho espacio. Actualmente, los microprocesadores son tan poderosos como varios sistemas mainframe, pero a una fracción de su precio [23].

El segundo avance importante fue la mejora en las redes de comunicación. Las *Redes de área local* o *LAN* (acrónimo de Local Area Network) [19], son redes de alta velocidad que permiten la interconexión de cientos de máquinas ubicadas dentro de un mismo edificio o complejo de edificios, de manera que es posible intercambiar información entre todas ellas en una fracción de segundo [23].

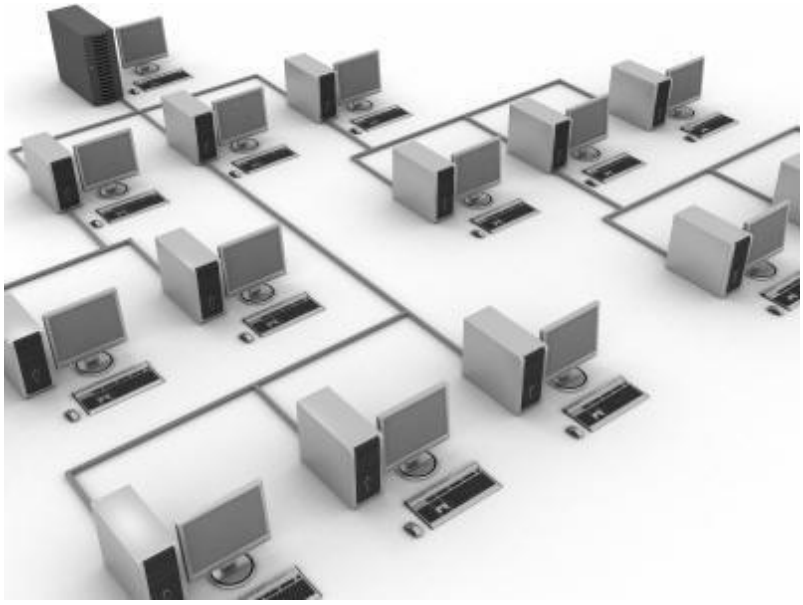

Figura 1.1: *Diagrama de una red de computadoras*

Por su parte, un *Sistema de Cómputo Distribuido* se define como

> *Una colección de computadoras independientes que dan al usuario la impresión de constituir un único sistema coherente.* [23]

La definición involucra que cada computadora independiente debe colaborar con el resto del sistema. Esta colaboración se lleva a cabo mediante una red de computadoras. Es común confundir a un sistema distribuido con una red; sin embargo, la diferencia principal radica en que

> *El usuario de un sistema distribuido no está consciente de que haya múltiples procesadores en el sistema. El usuario ve al sistema como una única computadora virtual.* [22]

A esta característica se le llama *Transparencia del sistema*[23].

En una red de computadoras, el usuario debe ingresar en un nodo y explícitamente definir el destino y contenido de los mensajes que serán enviados, así como las tareas que cada nodo de la red debe realizar. En un sistema distribuido, nada está definido explícitamente. Las tareas son asignadas automáticamente a las computadoras involucradas y el contenido y destino de los mensajes que circulan entre los nodos del sistema son definidos y especificados por estas tareas [22].

En un sistema distribuido, las diferencias entre las distintas computadoras y la forma en que se comunican entre sí deben permanecer ocultas al usuario. Lo mismo sucede con la organización interna del sistema. Además, los usuarios y las aplicaciones del sistema distribuido pueden interactuar con este de manera consistente y uniforme, sin importar dónde y cuándo tenga lugar esta interacción [23].

De esta forma, los sistemas distribuidos son sistemas de software construidos sobre una red de computadoras. En otras palabras, cuando hablamos de *Redes de Computadoras*, nos referimos al conjunto de computadoras y conexiones físicas que la conforman, mientras que cuando hablamos de *sistemas distribuidos*, hacemos referencia a un sistema de software[22].

Existen tres tipos de sistemas distribuidos: Sistemas distribuidos de cómputo, sistemas distribuidos de información y sistemas distribuidos embebidos. En este trabajo, nos enfocamos en los sistemas distribuidos de cómputo, de los cuales existen dos tipos: Sistemas de cómputo en clúster y sistemas de cómputo en malla [23]. Para el desarrollo de J-MIPS, nos centramos en los sistemas de cómputo en clúster.

1.1.1. Sistemas de cómputo en clúster

Para hablar del cómputo en clúster, es necesario definir una *Red de Área Local* o *LAN* [1]. Esta se define de la siguiente forma:

[1] Del ingles Local Area Network.

> *Una LAN consiste de un conjunto de dispositivos de software y hardware y un medio de transmisión de datos compartido entre los dispositivos, el cual permite la interacción (intercambio de datos) entre ellos.* [19]

En términos simples, una LAN es una red de computadoras cuya principal característica es ser de alta velocidad y cuyos componentes se encuentran alojados en el mismo edificio o en un complejo pequeño de edificios [22].

Así, en el computo en clúster, el hardware consta de una colección de computadoras similares, conectadas por medio de una red de área local, en donde cada computadora, generalmente ejecuta el mismo sistema operativo [23].

Estos sistemas crecieron cuando las computadoras personales se convirtieron en recursos relativamente baratos. La idea era construir una supercomputadora mediante la simple conexión de un conjunto de computadoras sencillas ubicadas dentro de una red LAN. En general, la computación en clúster se utiliza para la programación en paralelo con memoria distribuida, en donde un solo programa se ejecuta paralelamente en múltiples máquinas [23].

En un clúster, el acceso a los nodos se realiza mediante un único nodo maestro. Por lo general, el nodo maestro controla la ubicación de los nodos restantes, mantiene una cola de trabajos enviados y proporciona al usuario un mecanismo de interacción con el sistema. [23].

Generalmente, los clusters de computadoras se ejecutan en redes *Ethernet*. Ethernet es la tecnología de redes de área local más exitosa de los últimos 20 años. Consiste en una red de acceso múltiple con topología de bus, lo cual significa que un conjunto de nodos envían y reciben mensajes a través de un medio compartido. Cualquier mensaje enviado a la Ethernet es transmitido a todas las computadoras de la red, pero únicamente la computadora destinataria atenderá el mensaje [15, 19].

Los clusters de computadoras pueden ser implementados con diferentes tecnologías y topologías; sin embargo, en este trabajo supondremos que el clúster en el cual se ejecutan las aplicaciones que utilizan J-MIPS está basado en Ethernet, únicamente por ser esta última la tecnología más popular para los clusters de computadoras[23]. No obstante, podemos asegurar que J-MIPS puede ser ejecutado en cualquier sistema de red con la interfaz apropiada y para el cual exista

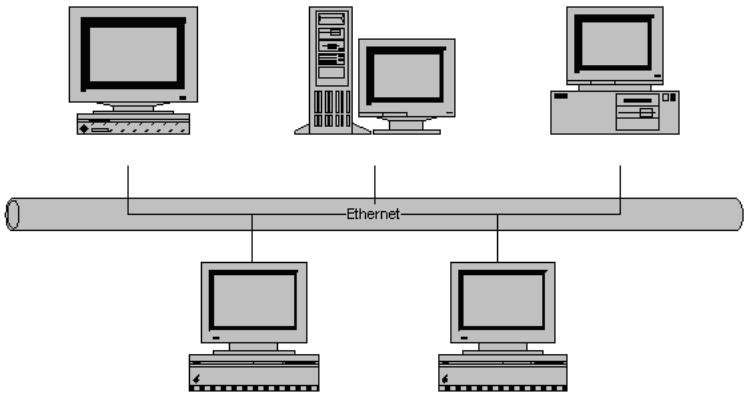

Figura 1.2: *Red Ethernet basada en topología de bus*

una Java Virtual Machine.

1.1.2. Características de un sistema distribuido

Como todos los sistemas de cómputo, los sistemas distribuidos poseen características que les permiten ser evaluados y comparados con otros. Estas características permiten a los usuarios elegir entre uno u otro sistema, dependiendo de las necesidades que requiera. Aunque existen muchas, nosotros consideramos un conjunto de características que permiten evaluar al sistema J-MIPS, así como compararlo con otros sistemas que cumplen el mismo objetivo.

Portabilidad

En un sistema distribuido, la portabilidad caracteriza la medida en la que una aplicación desarrollada para un sistema distribuido A puede ser ejecutada sin modifiaciones en un sistema distribuido diferente B que implemente las mismas interfaces que A [23].

Esta característica es importante en un sistema distribuido porque permite al programador ejecutar aplicaciones paralelas en diferentes sistemas sin la necesidad de redefinirlas. Además, la arquitectura de un sistema distribuido es en general mucho más dinámica que en un sistema secuencial debido a su alto grado de escalabilidad. Es importante entonces, que las aplicaciones que se ejecutan en sistemas distribuidos sean portables para que puedan adaptarse a cambios en la arquitectura física del sistema [23].

Comunicación

La comunicación entre procesos de software se define como *la capacidad que tienen dos procesos para intercambiar información* y representa uno de los principales temas de estudio en los sistemas distribuidos; en particular, porque los procesos residen en diferentes computadoras [23].

La comunicación entre procesos en un sistema distribuido esta basada en paso de mensajes (sección 1.4.1) y es el software de la red subyacente que ofrece las interfaces necesarias para lograr la comunicación. Expresar la comunicación de dos procesos por medio de paso de mensajes es más difícil que utilizar primitivas basadas en sistemas de memoria compartida. Los sistemas distribuidos modernos consisten en miles de procesos esparcidos a través de redes de poca fiabilidad, lo cual dificulta el desarrollo de aplicaciones distribuidas y requiere de primitivas de paso de mensajes eficientes.

Control de procesos

En sistemas distribuidos, el control de procesos se refiere a la capacidad de iniciar y detener tareas, de saber cuáles tareas se ejecutan en un momento determinado, en donde lo hacen y posiblemente, de realizar una migración de procesos de una computadora del sistema a otra [4, 23].

En la práctica, un nivel alto en el control de procesos puede degradar seriamente el rendimiento de un sistema distribuido. Sin embargo, el uso de técnicas *multithreading*, en las cuales se sustituye un proceso pesado de granularidad gruesa por un conjunto de procesos ligeros de granularidad fina, ayuda a mejorar el rendimiento del sistema, así como facilitar el desarrollo de aplicaciones distribuidas [23].

Control de recursos

El control de recursos de un sistema distribuido se refiere a la capacidad de éste de administrar los recursos de cómputo de los que dispone, en particular de la habilidad de añadir o eliminar computadoras del sistema, así como de reorganizar las conexiones entre ellas. Permitir a las aplicaciones paralelas interactuar y manipular su entorno de cómputo provee un poderoso paradigma para abordar problemas como balanceo de carga, migración de tareas y tolerancia a fallos [4].

Topología

En una red de computadoras, se llama topología al patrón de conexión entre sus nodos, es decir, a la forma en que los nodos de la red están interconectados[22].

En sistemas distribuidos, consideramos además el concepto de topología de procesos, que tratamos como el patrón de conexión entre los procesos que se ejecutan en el sistema por medio de paso de mensajes. Esta topología de procesos puede corresponder o no con la topología física de la red.

Tolerancia a fallas

Una característica de los sistemas distribuidos que los distingue de los sistemas consistentes de una sola computadora es la noción de *falla parcial*. Esta puede ocurrir cuando un componente de un sistema distribuido falla. Dicha falla puede afectar la operación de algunos componentes y al mismo tiempo dejar otros sin ninguna alteración en su funcionamiento. En sistemas no distribuidos, una falla es prácticamente total en el sentido de que afecta a todos sus componentes y puede fácilmente impedir el funcionamiento del sistema completo[23].

En sistemas distribuidos, un objetivo importante es construir el sistema de tal forma que pueda recobrarse de una falla parcial sin afectar al sistema completo. Particularmente, siempre que una falla ocurra en un sistema distribuido, éste debe seguir operando mientras se repara la falla. Esto quiere decir que el sistema **tolera la falla** y continua su operación incluso ante su presencia[23].

1.1.3. Cuellos de botella

Un problema en el procesamiento paralelo en memoria distribuida, en particular en clústers de computadoras, es la saturación del trabajo. En algunas ocasiones, existen áreas del sistema desproporcionadamente lentas respecto a otras áreas debido a una concentración de trabajo en esa área. Estas zonas ocasionan una degradación del rendimiento general del sistema y se les llama *"Cuellos de botella"*. Es necesario disminuir o eliminar los cuellos de botella reestructurando el programa para repartir el trabajo de forma que todas las zonas tengan una cantidad de trabajo similar [1].

1.2. Procesos y Threads en sistemas distribuidos

Generalmente, el término *Proceso* o *Proceso pesado* hace referencia a un programa en ejecución en un momento dado, el cual se compone de dos elementos[20, 21, 17]:

- *El código del programa*. Que describe el conjunto de instrucciones que deben ser ejecutadas, y que puede compartirse con otros procesos que estén ejecutando el mismo programa.

- *Conjunto de datos* asociados a dicho código.

La definición anterior, hace a los procesos poseedores de dos características:

- *Propiedad de recursos.* Un proceso incluye un espacio de direcciones virtuales para el manejo de la imagen del proceso; que es la colección de programa, datos, pila y atributos definidos en el bloque de control del proceso. A los procesos se les puede asignar propiedad de recursos tales como la memoria principal archivos y dispositivos de E/S por un cierto tiempo. El sistema operativo se encarga de evitar interferencias o bloqueos entre procesos con relación a los recursos [20].

- *Planificación.* Un proceso puede estar en diferentes estados en cada momento dado; puede estar en ejecución, esperando para ser ejecutado, terminado, etc. El sistema operativo es quien determina el estado de los procesos en todo momento, así como su orden de ejecución[20].

Un *hilo*, *Thread* o *Proceso ligero* se define como una unidad de utilización básica del CPU. Comprende un identificador, un contador de programa, un conjunto de registros y una pila. proceso pesado se compone de uno o más procesos ligeros, los cuales comparten los recursos asignados al proceso principal. Estos recursos pueden ser las direcciones de memoria, dispositivos de E/S, datos y archivos. Esta característica es inexistente entre procesos pesados distintos [16, 17].

El hecho de que cada thread posea sus propios datos, pero además comparta los recursos asignados a un proceso pesado(en particular el espacio de direcciones de memoria) con otros threads, evita replicar la asignación de recursos del sistema a cada thread, así como el tiempo utilizado en los cambios de contexto entre procesos[2] [1, 20]. Sin embargo, debe tenerse cuidado con los threads, pues el sistema operativo no evitará que un thread modifique un dato mientras otro lo está leyendo. Esta tarea es responsabilidad del programador [16].

Igual que un proceso, un hilo ejecuta su propio segmento de código, independientemente de otros hilos. Sin embargo, en contraste con los procesos, los hilos no hacen ningún intento por lograr un alto grado de transparencia de concurrencia si esto resulta en una degradación del rendimiento. Por lo tanto, los threads mantienen generalmente solo un conjunto de información mínima para permitir que el procesador sea compartido por varios hilos[23, 21].

[2]El cambio de contexto se refiere a las actividades que deben llevarse a cabo cuando el sistema operativo suspende la ejecución de un proceso para ceder recursos a algún otro proceso.

A menos que especifiquemos lo contrario, en este trabajo hablaremos de procesos y threads de manera indistinta. Esto es porque nuestro objetivo al diseñar programas paralelos es distribuir las tareas entre varios threads, distribuidos a su vez en distintas computadoras. Para hacer esto, debemos crear procesos que contengan a esos threads en cada computadora del sistema. Sin embargo, en el sistema distribuido ideal, tendremos un thread por cada computadora; es decir, un thread por cada proceso pesado. Aunque en la mayor parte de los casos no contemos con el sistema distribuido ideal (el cual se compone de tantas computadoras como necesitemos), para efectos del diseño de programas paralelos distribuidos, podemos asumir que lo tenemos.

1.3. Virtualización

En años recientes, el concepto de virtualización ha permitido a las aplicaciones, e incluso a ambientes completos como el sistema operativo, ejecutarse de manera concurrente con otras aplicaciones, pero de forma altamente independiente de su hardware y su plataforma subyacentes, provocando un alto grado de portabilidad [23]. Tal es el caso, por ejemplo, de la Java Virtual Machine, que es capaz de ejecutar los programas escritos y compilados en otras computadoras, con posiblemente hardware muy diferente al de la computadora que los ejecuta [6].Más aún, permite aislar las fallas ocasionadas por errores o problemas de seguridad. Todas estas características hacen de la virtualización, un concepto importante para los sistemas distribuidos [23].

En general, podemos ver a los hilos y a los procesos como una manera de hacer más cosas al mismo tiempo. Nos permiten construir partes de programas que parecen ejecutarse en forma simultánea. En una computadora con un solo procesador, esta ejecución simultánea es una ilusión: dado que solamente contamos con un procesador, sólo se ejecuta una instrucción de un proceso o hilo a la vez. El rápido intercambio entre hilos y procesos crea la ilusión de paralelismo [23].

La separación entre tener un solo procesador y ser capaz de pretender que existen más unidades de procesamiento se puede extender también a otros recursos. Esto origina lo que se conoce como *Virtualización de recursos*. Esta virtualización se ha aplicado durante décadas, pero su uso se ha incrementado conforme los sistemas distribuidos se han vuelto mas populares y complejos. Esto es, debido a que en su esencia, la virtualización trata con la extensión o el reemplazo de una interfaz existente de modo que imite el comportamiento de otro sistema. Esta característica permite a los programadores desarrollar programas paralelos en una computadora con un solo procesador [23].

1.4. Interconexión de procesos

En este trabajo, la comunicación e intercambio de datos entre los procesos de ejecución paralela se lleva a cabo por medio de *Paso de Mensajes*. Esto es debido a las primitivas de comunicación que ofrece Java para comunicar computadoras en una red [7]. Además, el paso de mensaje es un mecanismo que proporciona sincronización y comunicación entre procesos y tiene la ventaja de prestarse a ser implementado tanto en sistemas distribuidos como en sistemas de memoria compartida[20].

1.4.1. Paso de Mensajes

El paso de mensajes es un mecanismo para comunicar y sincronizar procesos. La funcionalidad de este mecanismo es proporcionada por dos primitivas básicas [10, 11, 20]:

- send(destino, mensaje)
- receive(origen, mensaje)

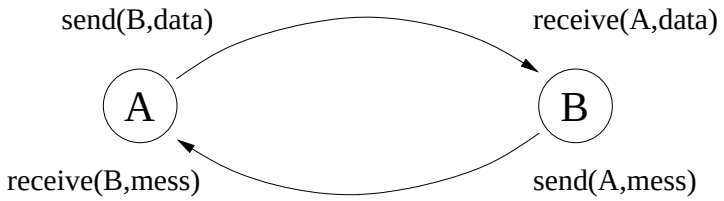

Figura 1.3: *Diagrama de paso de mensajes*

Un proceso envía información a otro proceso *destino* en forma de un *mensaje* serializado mediante la primitiva *send*. El proceso recibe información de algún proceso *origen* mediante la primitiva *receive*, y la almacena en la variable *mensaje* [20].

Al comunicar procesos mediante paso de mensajes, se requiere cierto nivel de sincronización. Un proceso no puede recibir un mensaje que no ha sido enviado por el proceso emisor. De igual forma, podría ser que el proceso emisor detuviera su ejecución hasta que el proceso receptor llamara a la primitiva *receive*[20]. En resumen, tenemos tres posibilidades:

- **El emisor y el receptor se bloquean.** Tanto el proceso emisor como el proceso receptor se bloquean hasta entablar la comunicación [20].

- **El emisor no se bloquea, pero el receptor sí.** En este caso, el emisor continúa su ejecución después de haber enviado el mensaje; el receptor, detiene su ejecución hasta que haya un mensaje por recibir [20].
- **El emisor se bloquea, pero el receptor no se bloquea.** En este caso, el proceso emisor bloquea su ejecución hasta que el receptor esté preparado para recibir el mensaje. El receptor, por su parte, verifica si hay un mensaje por leer; si dicho mensaje no existe, el receptor abandona su intento de recepción y continúa su operación [20].
- **Ni el emisor ni el receptor se bloquean.** Ninguno de los dos procesos involucrados en la comunicación detiene su ejecución. El emisor, como en el punto anterior envía el mensaje y continúa su ejecución. El receptor actúa como en el caso anterior [20].

Es necesario mencionar que los procesos involucrados en la comunicación por medio de paso de mensajes deben implementar ambas primitivas *send()* y *receive()* mediante los mismos protocolos para evitar errores de transmisión [20]. En este trabajo, utilizaremos el segundo punto como esquema para comunicar procesos (receptor bloqueante, emisor no bloqueante), pues creemos que tiene las siguientes ventajas:

- Al no bloquearse el proceso emisor, este puede continuar con su trabajo sin tener que esperar disponibilidad del receptor, lo cual creemos que disminuye el tiempo de espera para comunicar procesos.
- Al bloquearse el proceso receptor, el programador de aplicaciones paralelas puede estar seguro de que al finalizar la llamada *receive()*, el proceso emisor ha enviado el dato que se desea recibir, y este está disponible para su utilización en el proceso local. Aunque esto representa tiempo de espera por parte del proceso receptor, creemos que es más importante asegurar que los datos hayan sido recibidos antes de intentar utilizarlos.

1.5. Patrones de diseño para programación en paralelo

El procesamiento paralelo se define de la siguiente forma:

> *Procesamiento paralelo es la división de un problema, presentado como una estructura de datos y/o un conjunto de acciones, mediante múltiples componentes de procesamiento que operan simultáneamente. El resultado esperado es una ejecución más eficiente de la solución al problema* [10, 11, 12, 13].

Por su parte, dada la definición de procesamiento paralelo, un programa paralelo se define como:

> *La especificación de un conjunto de componentes de software que procesan y se comunican entre ellos de manera simultánea, para alcanzar un objetivo común* [10, 11].

Finalmente, un programa paralelo puede describirse por medio del tipo de componentes que lo conforman:

Componentes de procesamiento. Que realizan el trabajo de procesamiento y se enfocan en realizar operaciones simultáneas con los datos [10, 11].

Componentes de comunicación Que realizan el intercambio de información que los componentes de procesamiento requieren para realizar su tarea. La información intercambiada puede contener datos o llamadas a procedimientos [10, 11].

El poder del paralelismo recae en la partición de un problema en pequeños subproblemas más fáciles de resolver. Esta partición permite al programador diseñar e implementar por separado las operaciones necesarias para resolver el problema general, pero también, permite ejecutar dichas operaciones de manera simultánea [10, 13].

La especificación de los *Componentes de procesamiento* y de los *Componentes de comunicación*, define el comportamiento y eficiencia de un programa paralelo. En términos simples, los componentes de procesamiento son sub-procesos ejecutados sobre un conjunto de datos. Cada sub-proceso resuelve una pequeña parte del problema general. Los componentes de comunicación, por su parte, definen el flujo de información a través de estos sub-procesos, intercambiando datos o requiriendo operaciones entre ellos. Aunque los componentes de procesamiento se encargan de realizar las operaciones que resuelven el problema, son los componentes de comunicación (es decir el flujo de información entre sub-procesos) los que determinan la eficiencia y velocidad de ejecución de un programa paralelo.

A la organización y especificación de los componentes de comunicación de un programa paralelo se les conoce como *Patrón de Diseño del Programa Paralelo* [10, 11]. Los patrones de diseño se definen formalmente como sigue:

> *Los patrones de diseño... son descripciones de objetos comunicantes y clases que son configurados para resolver un problema general de diseño en un contexto particular* [10, 11].

Estos patrones, se clasifican de acuerdo a las siguientes características:

El tipo de paralelismo del sistema de software paralelo. Esto es, el tipo de partición del problema, que puede ser [10, 11]:

- *Paralelismo Funcional.* Que se enfoca en dividir el algoritmo principal en sub-algoritmos independientes mas pequeños
- *Paralelismo de Dominio.* Que se enfoca en dividir el conjunto de los datos procesados.
- *Paralelismo de actividad.* Que divide tanto el algoritmo como los datos.

La organización de la memoria de la plataforma de hardware paralela. Que define la accesibilidad de las direcciones de memoria a los procesos involucrados. Hay dos tipos de organización de la memoria[10, 11, 23]:

- *Memoria compartida.* En la cual todos los procesadores cuentan con el mismo espacio de direcciones de memoria, y por tanto, cualquier dirección de memoria, puede ser leída y modificada por cualquier proceso.
- *Memoria distribuida.* En la que cada procesador cuenta con una memoria local propia, y el acceso a la memoria de otros procesadores solo puede llevarse a cabo mediante operaciones de E/S.

Tipo de Sincronización. Dados dos procesos A y B que se comunican entre sí, la *Sincronización de la comunicación* determina si un proceso espera hasta que el otro esté listo para establecer la comunicación antes de continuar con su trabajo o no. La comunicación entre procesos puede ser [10, 11, 20]:

- *Síncrona.* En la cual, el proceso A que envía o recibe se bloquea hasta que el proceso B está listo para establecer la comunicación. Una vez establecida, ambos procesos, A y B, continúan su ejecución (tal es el caso, por ejemplo, de una llamada telefónica).
- *Asíncrona.* Donde ningún proceso se bloquea al esperar disponibilidad de su contraparte y ambos continúan su ejecución después de enviar o recibir el mensaje (como en los mensajes de correo electrónico).

Aunque existen diversos patrones de diseño, en este trabajo describimos únicamente dos de ellos, que ocuparemos en el capítulo 6 para realizar ejemplos de aplicación de la biblioteca J-MIPS.

1.5.1. Patrón de Filtros y Tuberías Paralelas (Parallel Pipes and Filters)

Este patrón de diseño es de tipo funcional, pues el algoritmo completo es particionado en múltiples procesos independientes (filtros) que representan los componentes de procesamiento del programa paralelo. Es ejecutado en un sistema de memoria distribuida y la comunicación entre sus componentes de procesamiento es asíncrona[10, 11, 13].

> *En el patrón Parallel Pipes and Filters, los datos pasan como un flujo de un componente (representando una estación de computo) a otro a través de un pipeline de diferentes componentes de procesamiento simultáneos. La característica principal es que los datos resultantes siguen una sola dirección a través de la estructura. La ejecución paralela completa se construye incrementalmente, cuando los datos llegan a cada estación. Los distintos componentes existen y procesan simultáneamente durante el tiempo de ejecución [10, 13].*

En la figura 1.4 se representa la estructura de un pipeline y la dirección en la que fluye la información.

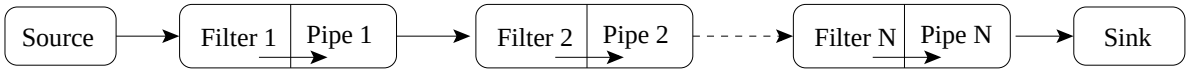

Figura 1.4: *Estructura de un pipeline*

Conceptualmente, podemos ver a este patrón de diseño como una línea de ensamblaje de automóviles. Ésta se compone de distintas estaciones de ensamblaje y una banda que circula en una sola dirección, por la cual el automóvil ensamblado recorre las estaciones. Cada estación, se dedica a realizar únicamente una tarea específica (como poner una puerta o pintar la carrocería), pero la labor de todas las estaciones resulta en un automóvil completamente ensamblado. Sin embargo, todas las estaciones están activas simultáneamente, ensamblando una parte distinta en automóviles distintos.

De igual forma, en un pipeline, los datos pasan de forma ordenada a través de todos los filtros. Como estos están activos simultáneamente, aceptan datos, realizan operaciones sobre ellos y los envían al siguiente filtro. Las tuberías sincronizan la comunicación entre los filtros[10, 13]. En la figura 1.5 se muestra la interacción entre los filtros al intercambiar datos y se puede apreciar que las operaciones se

llevan a cabo de manera simultánea entre ellos.

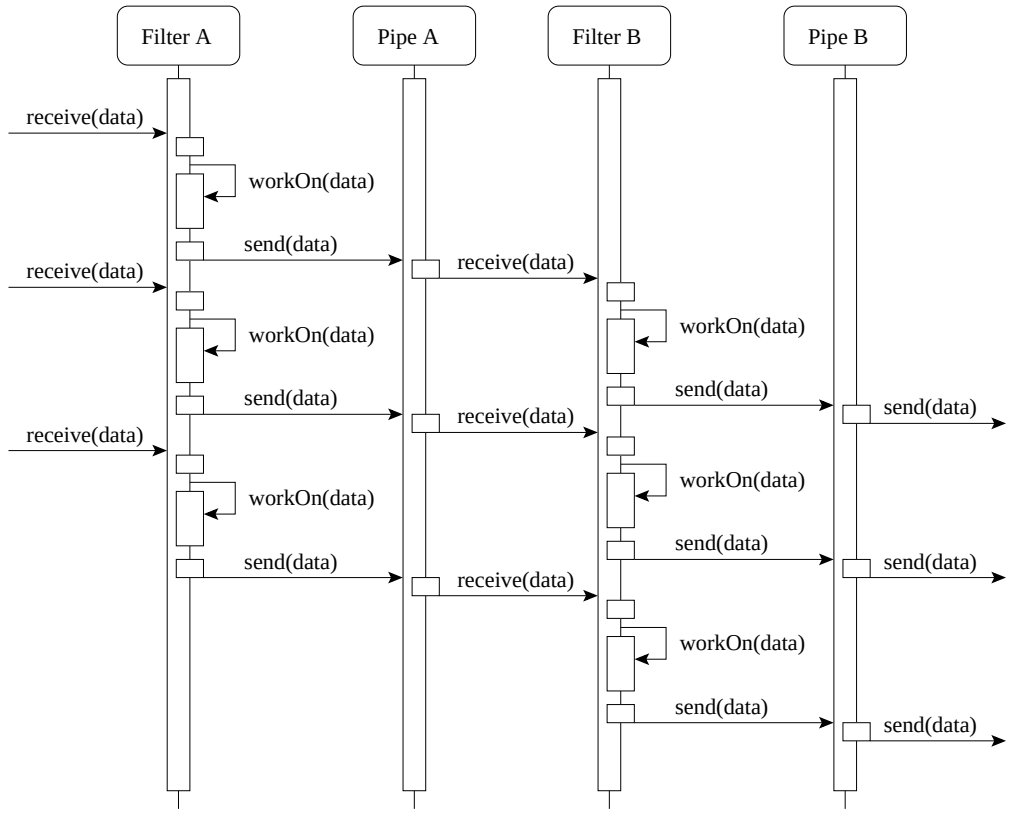

Figura 1.5: *Diagrama de secuencia del patrón Parallel Pipes and Filters*

Los componentes del patrón de Filtros y Tuberías son los siguientes:

- *Filtros (Filters)* Su tarea es recibir datos de una tubería, realizar operaciones sobre estos datos y enviar el resultado al siguiente filtro a través de la siguiente tubería[10, 13].
- *Tuberías (Pipes)* Son responsables de intercambiar datos entre filtros. En ocasiones es necesario utilizar "Buffers"[3] para evitar que los filtros detengan su ejecución esperando a que el siguiente filtro esté disponible para recibir datos. Debe considerarse el tamaño de estos buffers en términos de la cantidad de datos que deben ser "transportados" de un filtro a otro, para evitar desbordamiento del buffer o bloqueo en el flujo de información[10, 13].

[3]Un buffer es un espacio de memoria dedicado a almacenar resultados temporales requeridos por algún proceso en algún momento [20]

- *Fuente (Source)* La fuente es una estación especial del pipeline. Se encuentra antes del primer filtro y su responsabilidad es generar y transmitir el conjunto inicial de datos que serán pasados al primer filtro[10, 13].
- *Recolector (Sink)* Es la última estación del pipeline y se encarga de obtener el resultado final de todo el proceso de cómputo desde el último filtro [10, 13].

1.5.2. Patrón de Capas Paralelas (Parallel Layers)

Este patrón de diseño utiliza paralelismo de dominio, pero puede ser adaptado para realizar paralelismo funcional; sus componentes de procesamiento son llamados *"Componentes de capas"* e involucran, como antes, la ejecución de una parte del cómputo completo. Este patrón puede ejecutarse en un sistema de hardware de memoria compartida o distribuida y la comunicación entre sus componentes se lleva a cabo mediante comunicación síncrona [10, 12].

> *En este patrón arquitectónico, diferentes operaciones son realizadas por entidades conceptualmente independientes, ordenadas en forma de capas. Cada capa, como un nivel implícitamente diferente de abstracción, se compone de varios componentes que realizan la misma operación. Para comunicarse, las capas utilizan llamadas, refiriéndose entre ellas como componentes de alguna estructura compuesta [10, 12].*

El paralelismo entra cuando dos o más componentes de una capa se ejecutan simultáneamente, comúnmente realizando las mismas operaciones sobre un conjunto distinto de datos. Cada componente de una capa, puede ser creado de manera estática, con todos los componentes esperando llamadas de capas superiores, o de manera dinámica, en la cual una llamada de una capa dispara la creación de componentes en la capa inferior. La figura 1.6 muestra la organización de los componentes en el patrón de capas paralelas[10, 12].

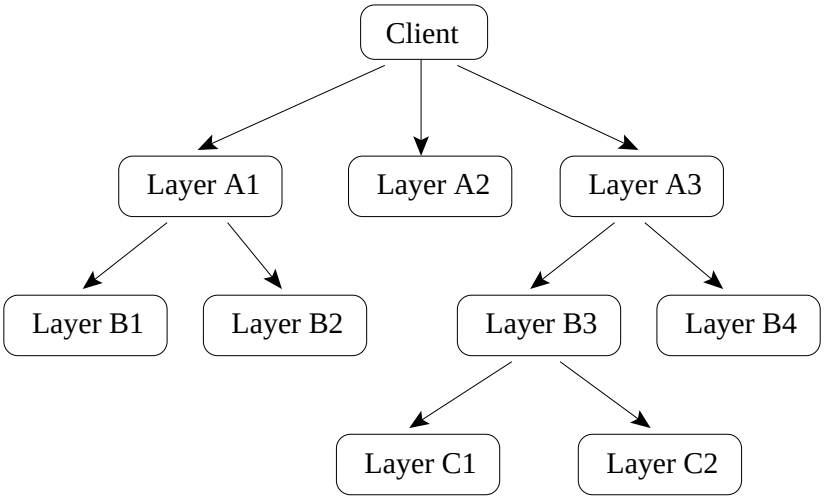

Figura 1.6: *Estructura del patrón Parallel Layers*

En este patrón, todas las capas están activas simultáneamente, reciben llamadas de capas superiores, ejecutan un conjunto de operaciones y regresan el resultado a la capa superior o envían otra llamada a algún componente de la capa inferior. Cuando un componente realiza una llamada, éste se bloquea hasta recibir un mensaje de retorno del componente que fue llamado en la capa inferior. Es importante aclarar que los componentes de una capa solo pueden tener comunicación con los componentes de la capa inferior inmediata efectuando una llamada y los de la capa superior inmediata, enviando un mensaje de retorno a la llamada recibida[10, 12]. En la figura 1.7 se muestra un ejemplo de las llamadas y valores de retorno en un cómputo con Parallel Layers. En la figura 1.8 se muestra el flujo de información y la ejecución de las llamadas entre los distintos componentes de procesamiento de la figura 1.7.

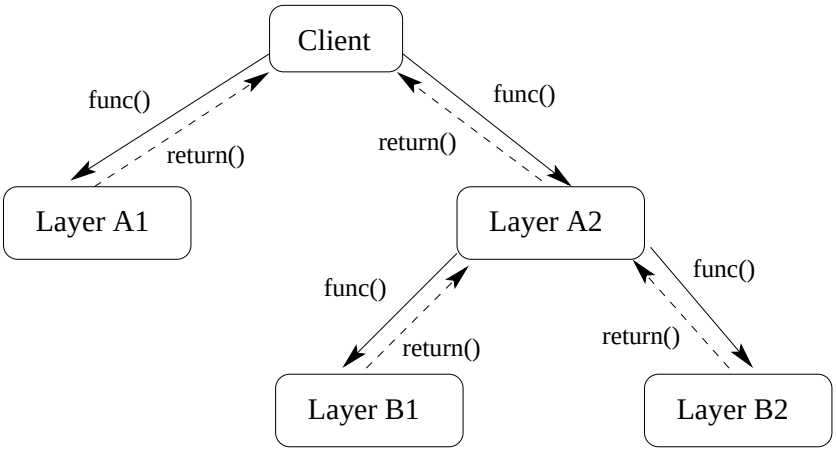

Figura 1.7: *Diagrama de un cómputo de 3 niveles con el patrón Parallel Layers. Las líneas continuas representan llamadas a funciones en la capa inferior. Las líneas punteadas representan valores de retorno a las capas superiores.*

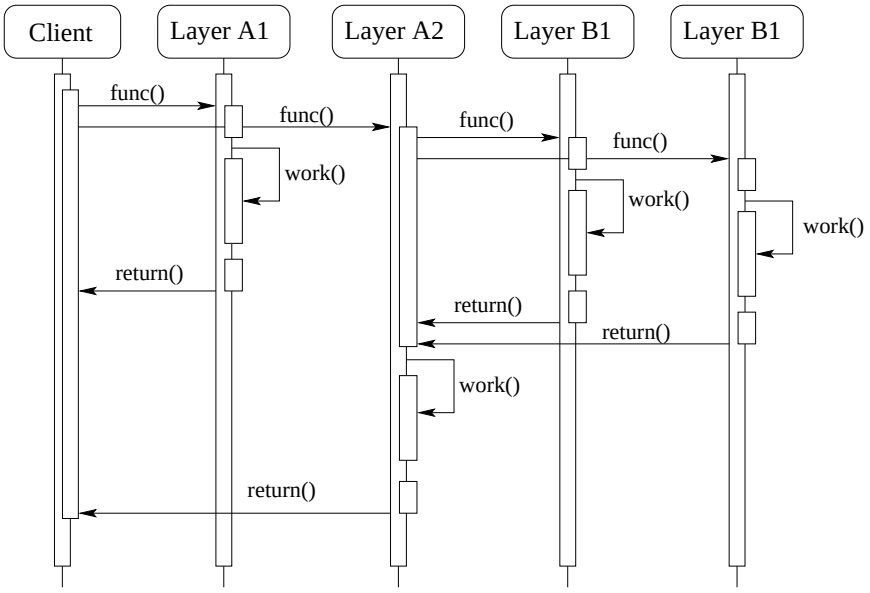

Figura 1.8: *Diagrama de secuencia del cómputo de la figura* 1.7

1.6. Mapeo de procesos

En la última parte del diseño de programas paralelos, es necesario especificar en dónde va a ser ejecutada cada tarea[3]. Es decir, cuando los procesos son ejecutados en una computadora real de memoria distribuida, éstos deben ser asignados (mapeados) a los procesadores. En la mayor parte de los casos, los procesos a ejecutar exceden en cantidad a los procesadores disponibles para ejecutarlos, de tal forma que más de un proceso será mapeado a un mismo procesador. La forma en la que esta asignación es llevada a cabo, tiene un impacto directo y pronunciado en el tiempo de ejecución del programa paralelo y de esta forma, en el rendimiento general del sistema paralelo. Al proceso de encontrar la mejor de estas asignaciones, esto es, aquélla en la que el tiempo de ejecución del sistema paralelo es mínimo, se le conoce como *El problema del mapeo* [8].

1.6.1. El problema del mapeo

Supóngase que un programa paralelo es un conjunto de n procesos comunicantes sin precedencia entre ellos. El programa será ejecutado en p procesadores. El objetivo es asignar los procesos a los procesadores de tal forma que el tiempo de ejecución del sistema paralelo completo sea mínimo. Esto es equivalente a encontrar una partición de n procesos en p subconjuntos que minimice el tiempo de ejecución, el cual está determinado por el máximo de los tiempos de ejecución de los subconjuntos de esa partición [8, 2].

La cantidad $S(n,p)$ de posibles p-particiones, cuando $p \geq 2$ y $n > p$ es [8]:

$$S(n,p) = \frac{1}{p!}\sum_{j=1}^{p}(-1)^{p-j}\binom{p}{j}j^n \tag{1.1}$$

Más formalmente, el tiempo de ejecución de un programa paralelo está dado como $T(P) = t(P_i(p))$ donde $t(P_i(p))$ es el tiempo de ejecución de la p-partición $P_i(p)$, $i = 1, ..., S(n,p)$. El objetivo es minimizar la función $T(P)$ encontrando la p-partición $P_i(p)$ apropiada [8].

El proceso de minimización de $T(P)$, sin embargo, ha probado ser un problema muy complicado por resolver. Por una parte, la cantidad $S(n,p)$ crece exponencialmente con n, por lo cual, para valores de n muy grandes, probar todas las posibles particiones no es factible [8]. Por otro lado, se ha probado que este problema es de tipo *NP-Completo* [2], por lo cual, cualquier aproximación correcta y completa que encuentre la solución a este problema no sería distinta que probar

con todas las posibles particiones [4] [8, 3].

Se han hecho muchas y variadas investigaciones alrededor del problema del mapeo general los últimos 20 años, pero solamente para casos muy específicos se ha encontrado una solución polinomial. Sin embargo, existen estrategias y algoritmos que permiten balancear la carga de trabajo de cada procesador de tal forma que el tiempo de ejecución del sistema paralelo sea eficiente [8, 3].

Mencionamos el problema del mapeo por ser un problema presente en todas las aplicaciones de cómputo paralelo en memoria distribuida. Sin embargo, el objetivo de la biblioteca J-MIPS es dar al usuario la libertad de mapear los procesos en las computadoras del clúster. Esto es, el usuario tiene la responsabilidad de especificar dicho mapeo y la biblioteca J-MIPS tiene la responsabilidad de hacerlo realidad.

1.7. Resumen del capítulo

En este capítulo se ven las bases y conceptos que permiten entender al lector el diseño de la biblioteca J-MIPS, así como el diseño de los ejemplos de aplicación proporcionados en el capítulo 6.

En la primera parte del capítulo se revisan y especifican los conceptos Red de computadoras, Clúster de computadoras, LAN, Ethernet, Sistema de cómputo distribuido y cuello de botella desde un enfoque técnico para ayudar a entender al lector la plataforma y entorno de trabajo para los cuales fue diseñada la biblioteca J-MIPS. También se aclara la diferencia entre Red de computadoras y Sistema de cómputo distribuido que suele presentarse en la literatura.

En la segunda parte de este capítulo se abordan los aspectos teóricos que permiten el diseño e implementación de programas paralelos. Se abordan los conceptos Proceso, Thread y Virtualización y se provee al lector con las herramientas que permiten comunicar y organizar los componentes de un programa paralelo: Paso de mensajes y Patrones de software.

Finalmente, se introduce al lector en el Problema del mapeo que surge al combinar la estructura teórica del programa paralelo con la estructura física del clúster de computadoras en el cual es ejecutado.

[4] A menos que sea demostrado que $P = NP$

Capítulo 2

Trabajo relacionado

Muchas aplicaciones han sido desarrolladas desde diferentes enfoques para tratar el problema que intentamos resolver en esta tesis. La compañia Apple Inc. por ejemplo, ha desarrollado una herramienta llamada *Xgrid*, que toma una red no dedicada de computadoras (como internet) y aprovecha los recursos de los nodos inactivos o con poca carga de trabajo para realizar cálculos [14]. Los principales componentes de Xgrid son el *cliente*, el *controlador* y uno o más *agentes*. El *cliente* manda un trabajo para realizar al *controlador*, quien divide el trabajo en tareas más pequeñas y las asigna a los *agentes*. Estos realizan las tareas asignadas y regresan datos al *controlador*. El *controlador* supervisa a los *agentes*, recolecta los datos y notifica al *cliente* cuando el trabajo termina [24].

En este capítulo se revisan las características dos herramientas muy populares para la programación paralela: Parallel Virtual Machine (PVM) y Message Passing Interface (MPI). Estas dos herramientas intentan resolver el mismo problema que se aborda en esta tesis (interconexión y mapeo de procesos) desde perspectivas diferentes.

No profundizamos en este trabajo en los aspectos técnicos de estas herramientas, pero mencionamos sus características más importantes respecto a:

- Portabilidad
- Comunicación entre procesos
- Control de los procesos
- Control de los recursos
- Topología
- Tolerancia a fallas

2.1. Parallel Virtual Machine (PVM)

PVM inició en el verano de 1989 en el Oak Ridge National Laboratory a cargo de Vaidy Sunderam y Al Geist debido a la necesidad de explorar el cómputo distribuido heterogéneo. En este ámbito, surgió la idea de una Maquina Virtual Paralela. Años después, en marzo de 1991, PVM fue reescrito a la versión PVM 2 en la University of Tenessee. Después de diversas modificaciones, la versión 3 fue programada y completada en febrero de 1993 [4, 5].

El diseño central de PVM gira alrededor de la noción de Máquina Virtual, la cual es un conjunto dinámico de computadoras no necesariamente homogéneas conectadas en una red que a nivel lógico, se presentan a sí mismas como una sola computadora paralela. Este concepto es fundamental para la perspectiva de PVM, pues provee las bases para la heterogeneidad, portabilidad y encapsulamiento que constituyen al software PVM [4, 5].

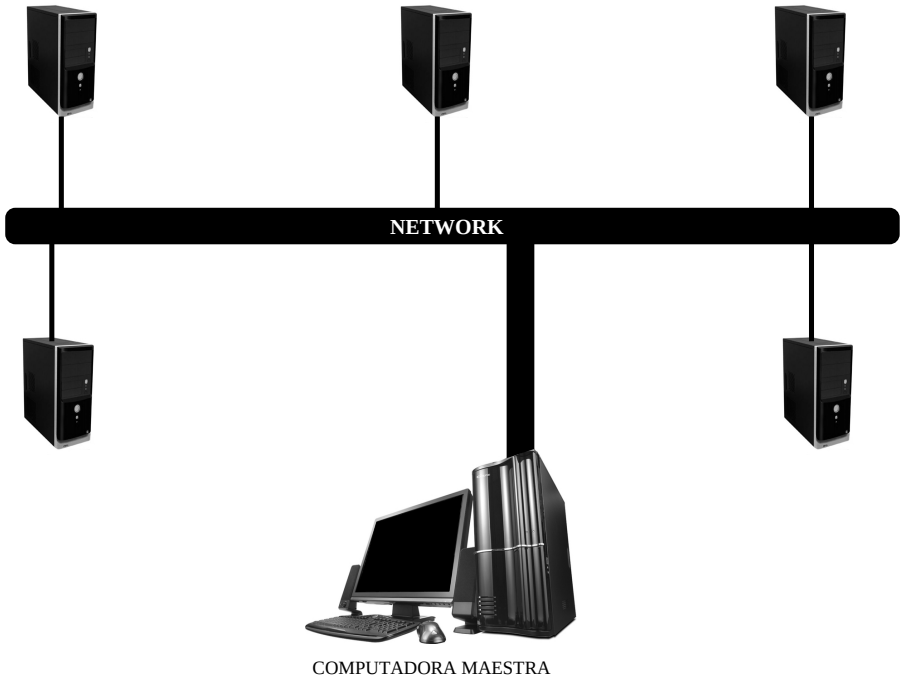

Figura 2.1: *Esquema del concepto de PVM. La computadora maestra es la parte del sistema que interactúa con el usuario*

PVM provee un marco de trabajo unificado dentro del cual los programas paralelos pueden ser desarrollados de una forma eficiente e

> *intuitiva utilizando el hardware existente. PVM permite ver a una colección de computadoras heterogéneas como una sola máquina virtual paralela. PVM administra transparentemente las rutas de los mensajes, conversión de datos y la agenda de procesos a través de una red de arquitecturas de cómputo incompatibles* [5].

Desde la perspectiva del usuario, éste escribe una aplicación paralela como una colección de procesos comunicantes. Los procesos acceden a los recursos de PVM mediante una biblioteca de funciones. Estas permiten el inicio y término de los procesos a través de la red, así como la comunicación y sincronización entre ellos. En cualquier punto de la ejecución, el usuario o la misma aplicación puede modificar dinámicamente los procesos existentes o la cantidad de computadoras en el sistema. Es decir, un usuario o proceso puede iniciar o detener otros procesos, así como agregar o eliminar computadoras de la maquina virtual [5].

En las secciones siguientes, describimos las características principales del funcionamiento de PVM.

2.1.1. Portabilidad

Los programas en PVM son portables en el sentido de que pueden ser copiados, compilados y ejecutados en arquitecturas diferentes sin ser modificados. Además, diferentes aplicaciones en PVM pueden comunicarse entre sí, permitiendo la comunicación y cooperación entre más de un sistema distribuido. A esta característica se le llama *Interoperabilidad* de los programas [4].
Los programadores de PVM dieron mucha más importancia a la portabilidad de los programas que al rendimiento del sistema, pues el diseño de PVM se enfoca en la ejecución de programas donde la máquina virtual es heterogénea, escalable y con tolerancia a fallos [4].

2.1.2. Comunicación entre procesos

La comunicación entre procesos en PVM se lleva a cabo por medio de paso de mensajes (descrito en la sección 2.4.1) [4, 5].

Enviar un mensaje requiere tres pasos en PVM: 1) Un buffer de salida debe ser inicializado; 2) el mensaje debe ser "empacado" en ese buffer; 3) El mensaje empacado se envía a otro proceso [5].

Para recibir un mensaje, PVM ofrece la posibilidad de llamar a la rutina *receive* de manera bloqueante o no bloqueante. El mensaje recibido debe ser "desempacado" y leído desde un buffer de entrada [5].

La función *receive* puede ser configurada para recibir mensajes solo de un determinado grupo de usuarios o procesos [5].

Para comunicarse con otras computadoras de la máquina virtual, PVM provee un sistema general de servicio de nombres. Cada proceso puede construir un mensaje y ponerlo en el servidor de nombres con una "llave" asociada. Esta llave es una cadena de texto definida por el usuario. Cuando un proceso busca un nombre, se le envía el mensaje almacenado. Este mensaje puede contener diferentes tipos de datos, por ejemplo, los identificadores de un conjunto de procesos que deben ser ejecutados, la dirección de un servidor o datos numéricos que requieren ser procesados [4].

2.1.3. Control de procesos

PVM es capaz de iniciar y detener procesos, así como de obtener información sobre cuáles procesos están siendo ejecutados y en dónde están siendo ejecutados. Para realizar estas tareas, PVM es capaz de averiguar el estado del sistema para determinar cuántos procesos pueden ser ejecutados en los posiblemente dinámicos recursos de cómputo [4].

2.1.4. Control de recursos

En términos del manejo de recursos, PVM es dinámico. El sistema administra una lista de computadoras disponibles (host pool). Esta lista puede ser modificada en cualquier momento agregando o eliminando computadoras, ya sea desde una consola del sistema o por medio de la misma aplicación paralela en ejecución. Al permitir que las aplicaciones interactúen y modifiquen su entorno de cómputo, se facilita el balanceo de la carga de trabajo, la migración de tareas y la tolerancia a fallas, pues si un procesador deja de funcionar, PVM es capaz de distribuir su trabajo al resto de los procesadores del sistema y de modificar la lista de computadoras disponibles para evitar que nuevas tareas sean asignadas al procesador defectuoso [4].

En PVM, el acceso al hardware es transparente. En vez de dejar que el programador seleccione manualmente dónde va a ser ejecutado cada proceso, la máquina virtual provee una abstracción simple para mapear los procesos. El usuario puede crear una colección arbitraria de máquinas y tratarlas como elementos de procesamiento uniformes sin atributos especiales, o puede elegir explotar las capacidades de computadoras físicas específicas, registradas en el host pool, mapeando ciertos procesos a las computadoras más adecuadas para ejecutarlos [4, 5].

2.1.5. Topología

En PVM, el usuario crea manualmente grupos de procesos que deben ser ejecutados y especifica la organización de las comunicaciones entre ellos; sin embargo, PVM no soporta la especificación de una topología virtual de interconexión de procesos[5].

2.1.6. Tolerancia a fallas

Generalmente, en los sistemas paralelos cuando un proceso falla, todo el sistema falla sin ninguna clase de aviso y el programador puede pasar horas o días tratando de encontrar el error que hizo fallar al sistema [5].

PVM soporta un esquema básico de notificación de fallas. Bajo el control del usuario, los procesos pueden registrarse en el sistema para ser "notificados" cuando el estatus de la máquina virtual cambia o cuando los procesos fallan. Esta notificación se manifiesta en mensajes de eventos especiales que contienen información sobre el evento que se generó debido a la falla. Un proceso puede notificar un error a cualquier otro proceso que espere recibir un mensaje [5].

Supongamos por ejemplo, que dos procesos registrados A y B tienen comunicación entre ellos. B espera un mensaje de A, pero este último muere súbitamente debido a un error. Entonces B recibe un mensaje de notificación en vez del mensaje que esperaba de A. El mensaje de notificación permite a B tomar una acción en respuesta a la falla, en vez de colgarse o fallar junto con A [5].

Este tipo de notificaciones en la máquina virtual son útiles al controlar los recursos de cómputo. Cuando una computadora sale de la máquina virtual, los procesos pueden utilizar mensajes de notificación para reconfigurar su distribución entre las computadoras restantes. Cuando una nueva computadora entra en la máquina virtual, los procesos pueden notificarse de igual forma y redistribuir su carga de trabajo [5]. Las notificaciones en PVM pueden ser vistas por los programadores orientados a objetos como el sistema de manejo de excepciones try-catch [6].

2.2. Message Passing Interface (MPI)

La API MPI-1 fue diseñada por un comité de cerca de 60 expertos en la investigación e industria del cómputo de alto rendimiento entre 1993 y 1994. MPI surgió debido a que cada proveedor de arquitecturas distribuidas diseñaba su propia API de paso de mensajes, de tal forma que era imposible hacer aplicaciones portables. La intención de MPI es especificar un estándar de paso de mensajes

para el cómputo paralelo distribuido que los proveedores implementen en sus sistemas, permitiendo así el diseño e implementación de aplicaciones portables para sistemas distribuidos. Además, MPI fue pensado y diseñado para el cómputo de alto rendimiento [4, 18].

El estándar MPI-1 tiene entre algunas de sus características principales [4]:

- *Un conjunto extenso de funciones para comunicación punto a punto.*
- *Un conjunto de rutinas para comunicación colectiva que permiten comunicar grupos de procesadores.*
- *Un contexto de comunicación que provee soporte para el diseño de bibliotecas de software paralelo.*
- *La habilidad de especificar topologías de comunicación.*

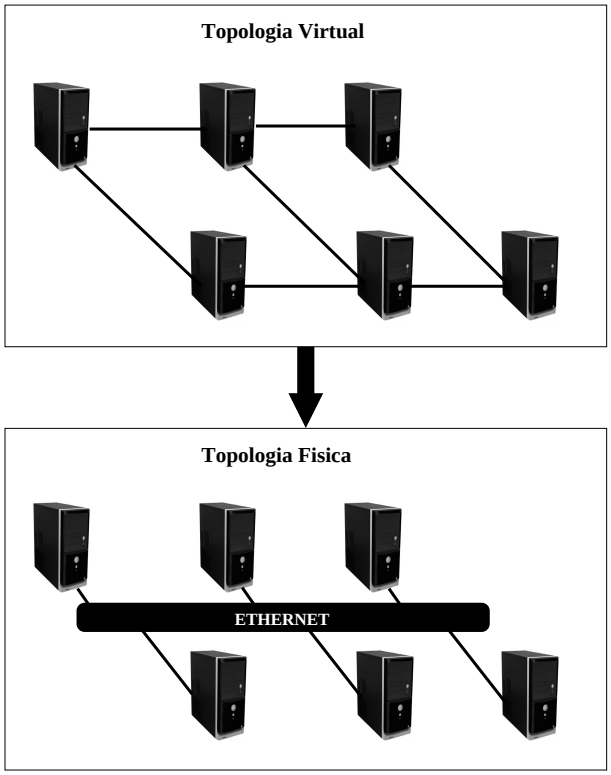

Figura 2.2: *Esquema del concepto de MPI. El diagrama muestra la capacidad de MPI de manejar topologías virtuales.*

Los usuarios de MPI-1 se dieron cuenta de que las aplicaciones no eran del todo portables, pues no había un método estándar para iniciar las tareas de MPI en computadoras diferentes, ya que diferentes implementaciones de MPI utilizaban métodos diferentes. En 1995 se inició el diseño de MPI-2 para corregir este problema y añadir características especiales de comunicación, entre las cuales están [4]:

- Funciones para iniciar procesos MPI y procesos No-MPI.
- Funciones de comunicación unidireccionales.
- Funciones de comunicación colectiva no-bloqueantes.
- Compatibilidad con C++

MPI-2 se terminó en enero de 1997 y agregó 120 funciones a las 128 existentes en MPI-1 [4].

2.2.1. Portabilidad

MPI fue diseñada con la intención de aprovechar todas las ventajas y características del paso de mensajes y clústers de computadoras de tal forma que las aplicaciones construidas bajo MPI pudieran ser ejecutadas en varios sistemas. MPI es portable en un sentido similar al de PVM, los programas escritos para cierta arquitectura pueden ser copiados, compilados y ejecutados en otras arquitecturas sin modificaciones. Sin embargo, los programas en MPI no son interoperables. Un programa puede ser ejecutado en diversas arquitecturas y es portable en ese sentido, pero diferentes aplicaciones en MPI no pueden interactuar entre ellas. [4, 18].

2.2.2. Comunicación entre procesos

La comunicación en MPI se lleva a cabo por medio de paso de mensajes. En este ámbito, el concepto más importante introducido por MPI es el *comunicador*. Este último puede ser pensado como un enlace entre un contexto de comunicación y un grupo de procesadores.

La figura 2.3 muestra un problema típico de comunicación entre dos contextos. Supóngase que existen dos procesos comunicantes A y B, además de cierta biblioteca con una subrutina llamada *lib()*. Esta subrutina realiza de manera transparente al usuario una llamada a la función *send(B,data)*. Su contraparte en la comunicación, realiza una llamada a la función *receive(A,data)*. En este contexto, cada

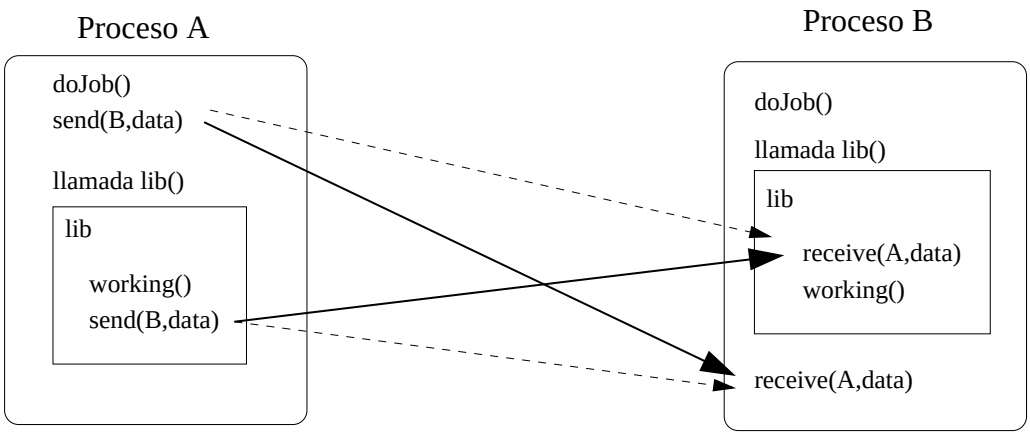

Figura 2.3: *Problema de comunicación en contextos distintos. Las flechas continuas indican el orden correcto en el que deberian ser recibidos los mensajes. Las flechas punteadas indican el orden incorrecto en el que en realidad son recibidos.*

vez que se llama a las funciones *send* y *receive* se desea establecer comunicación entre elementos de la función *lib()* de la biblioteca utilizada [4].

Supóngase ahora que en el proceso A, el usuario llama a la función *send(B,data)* justo antes de llamar a la función *lib()* (figura 2.3). Por su parte, en el proceso B, el usuario invoca la función *lib()* justo antes de llamar a *receive(A,data)*. El primer *send(B,data)* del proceso A es recibido por la función *receive(A,data)* de la función *lib()*. Análogamente, *send(B,data)* de la biblioteca *lib()* en el proceso A, es recibida por *receive(A,data)* en el código del usuario del proceso B. Es decir, los mensajes se reciben en desorden porque las llamadas a *send* y *receive* se salen del contexto de comunicación [4].

Para resolver este problema, se asigna una etiqueta a cada contexto de comunicación y se envía junto con el mensaje. De esta forma, el receptor es capaz de distinguir los mensajes enviados a su propio contexto de los mensajes enviados a otros contextos de comunicación [4].

MPI agrupa los conceptos de *contexto* y *grupo de procesadores* en un **comunicador**. Cuando un programa inicia su ejecución, todos los procesos reciben un comunicador global y una lista estática de todos los procesos que iniciaron junto con él. Cuando es requerido, el programa crea un nuevo contexto a partir de uno ya existente. La creación de un nuevo contexto se convierte en una operación síncrona entre los procesos que forman el nuevo comunicador. Sin embargo, es posible que dos grupos independientes de procesos compartan la misma etiqueta de

contexto. Esto quiere decir que es inseguro que los grupos se envíen mensajes entre sí. MPI resuelve este problema introduciendo el concepto de *inter-comunicador* que permite a dos grupos de procesos ponerse de acuerdo en un contexto de comunicación seguro [4].

A pesar de los intentos del comité diseñador de MPI, la comunicación segura entre contextos sigue generando dificultades. Por ejemplo, dada la naturaleza de los comunicadores, se hace engorrosa la creación de nuevos procesos que deban comunicarse con los ya existentes. Además, cuando un proceso falla, el contexto del comunicador se hace inválido. Dado que todas las intra-comunicaciones se crearon a partir este comunicador inválido, el comportamiento del programa se hace impredecible [4].

2.2.3. Control de procesos

MPI-1 no contiene métodos para iniciar, detener o supervisar procesos o aplicaciones. MPI-2, en contraste, contiene funciones para iniciar un grupo de procesos y para enviar señales a tales procesos (por ejemplo la señal de terminación "kill"), lo cual permite cierto control sobre procesos y aplicaciones de manera remota dentro del clúster [18].

2.2.4. Control de recursos

MPI fue diseñado para ser de naturaleza estática, con el objetivo de mejorar el rendimiento y la velocidad de los programas y a diferencia de PVM, no encierra el concepto de virtualización o máquina virtual. Debido a ello, MPI carece de funciones para supervisar la ejecución de los programas en las computadoras del clúster, para conocer en todo momento los recursos disponibles del sistema, para agregar o eliminar computadoras de manera dinámica y para reasignar procesos a diferentes procesadores en la forma en la que PVM lo hace. Estas tareas de manejo de recursos se dejan al usuario[4].

2.2.5. Topología

MPI provee un alto nivel de abstracción por encima del cálculo de recursos en términos de la topología de paso de mensajes. En MPI, los procesos pueden ser acomodados para interactuar dentro de una topología especificada a nivel lógico. La comunicación entre procesos, se da entonces dentro de la topología especificada sin importar si ésta se corresponde con la topología física de la red [18].

2.2.6. Tolerancia a fallas

Aunque MPI-1 no incluía originalmente ningún mecanismo para el control de fallos, el estándar MPI-2 incluye un sistema de notificaciones similar al de PVM [1] [4].

En MPI-1, el problema respecto a la tolerancia a fallas es que los procesos y los procesadores de la red son considerados estáticos. Una aplicación MPI-1 debe ser iniciada como un solo grupo de procesos de ejecución simultánea. Si un procesador o proceso falla, el sistema completo falla sin posibilidad de recuperación. Esto es importante para evitar procesos bloqueados. Sin embargo, no hay una manera de responder al error [18].

MPI-2 incluye especificaciones para iniciar nuevos procesos y para que los procesos sean notificados cuando ocurre un error. Esto expande las capacidades originales de MPI. Sin embargo, MPI-2 aún no tiene un mecanismo para recuperarse automáticamente de ciertos errores, como por ejemplo, la muerte súbita de un proceso. Una de las razones fundamentales de que MPI no sea tolerante a fallas es que la comunicación que se lleva a cabo entre procesos es síncrona, y por tanto, si se lleva a cabo comunicación entre dos procesos y uno falla, seguramente su contraparte en la comunicación fallará también [4].

2.3. Resumen del capítulo

En este capítulo se revisan las dos herramientas más populares para hacer cómputo distribuido en un cluster: PVM y MPI. La revisión se lleva a cabo de manera superficial y gira en torno a algunas características importantes de los sistemas distribuidos: Portabilidad, comunicación entre procesos, control de procesos, control de recursos, topología y tolerancia a fallas.

Tanto MPI como PVM son capaces de implementar y ejecutar programas paralelos distribuidos. Sin embargo, cada uno lo hace desde una perspectiva diferente. PVM aborda los problemas desde el concepto de Máquina Virtual, lo cual hace de esta API una herramienta dinámica, flexible y cómoda sacrificando el rendimiento general del sistema, mientras que MPI está diseñada para ser estática y menos flexible que PVM, pero con el objetivo de ser una API rápida y eficiente. En diversas ocasiones, éste es el objetivo buscado por los desarrolladores de aplicaciones paralelas distribuidas.

[1] Ver sección 3.1.6

Capítulo 3

Diseño de la biblioteca J-MIPS

En este capítulo se introduce al lector al diseño de la biblioteca J-MIPS. Antes de exponer el diseño en sí de los componentes de la biblioteca, se inicia describiendo las condiciones que fueron tomadas en cuenta para la implementación y planeación de los elementos que intervienen en la interconexión y mapeo de procesos de software.

Se continúa el capítulo describiendo el concepto y la idea general de la biblioteca; esto es, la perspectiva desde la cual se resuelve el problema planteado en esta tesis y el funcionamiento general de los componentes de la biblioteca en conjunto.

3.1. Consideraciones para el diseño

La biblioteca que diseñamos e implementamos en este trabajo está orientada a los programadores de sistemas paralelos distribuidos. Tomando en cuenta esto, consideramos necesarios los siguientes puntos para el desarrollo de J-MIPS:

- **Sobre el hardware.** Veremos en secciones posteriores que la flexibilidad de J-MIPS permite su ejecución en redes de computadoras de muy diversos tipos. Sin embargo, suponemos que el entorno de hardware en el cual se utiliza J-MIPS es un clúster de computadoras como el especificado en la sección 1.1.1, es decir, una red LAN de computadoras conectadas a través de un bus de datos, controladas por un único nodo maestro (figura 3.1). Esta suposición, más que un requisito de ejecución, puede ser vista como una recomendación para quien pretenda usar la biblioteca J-MIPS.
- **Sobre el software.** Asumimos que cada nodo de la red donde será ejecutada nuestra biblioteca, así como el nodo maestro (debe haber uno) poseen su propia Máquina Virtual de Java, versión 1.5 o superior, para poder ejecutar los programas paralelos realizados con ayuda de J-MIPS. Asimismo,

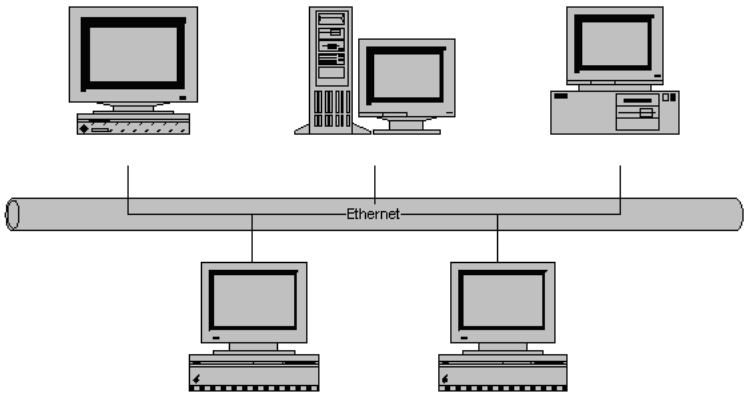

Figura 3.1: *Red Ethernet*

suponemos que el programador cuenta con un entorno de programación Java. Asumimos también que cada nodo del cluster, incluido el nodo maestro, cuenta con el protocolo de conexión TCP/IP, pues es éste el que Java utiliza para comunicar máquinas virtuales remotas.

Las consideraciones anteriores nos permiten iniciar el desarrollo de J-MIPS.

3.2. Concepto e idea general

Como especificamos en la introducción de esta tesis, nuestro objetivo es resolver un problema importante que surge en el procesamiento paralelo distribuido:

> Dado un conjunto de procesos que residen en una sola computadora de un clúster y un mapeo de los procesos hacia otras computadoras del mismo clúster, distribuir los procesos a las computadoras de acuerdo al mapeo y permitir la transferencia de información entre ellos para poder ser ejecutados paralelamente en el clúster.

Como hemos visto en el capítulo 2, este problema no es nuevo y existen actualmente varias aproximaciones en distintos lenguajes de programación, incluido Java. Sin embargo, al momento del diseño de nuestro trabajo, incluimos aspectos que no solamente resuelven el problema planteado, sino además facilitan al programador el diseño e implementación de aplicaciones distribuidas.

Nuestra idea es implementar una red virtual de computadoras dirigidas por una única computadora virtual maestra. Esta red ejecuta los programas paralelos en

forma de múltiples procesos divididos entre las computadoras que la conforman y es mapeada a la red física disponible. Este mapeo se realiza en dos niveles: las computadoras virtuales son mapeadas a las computadoras físicas y luego los procesos son mapeados a las computadoras virtuales (Figura 3.2) . Para lograr esto, dividimos la red virtual en tres tipos de componentes: La computadora virtual *Master* , las computadoras virtuales esclavas y los procesos *RemoteJob* que se ejecutan en la red. **Por el resto de esta tesis, nos referimos a la computadora virtual maestra como *objeto Master* o simplemente *Master*. De igual forma, hacemos referencia a las computadoras virtuales esclavas como *objetos Manager* o solo *Managers*[1].**

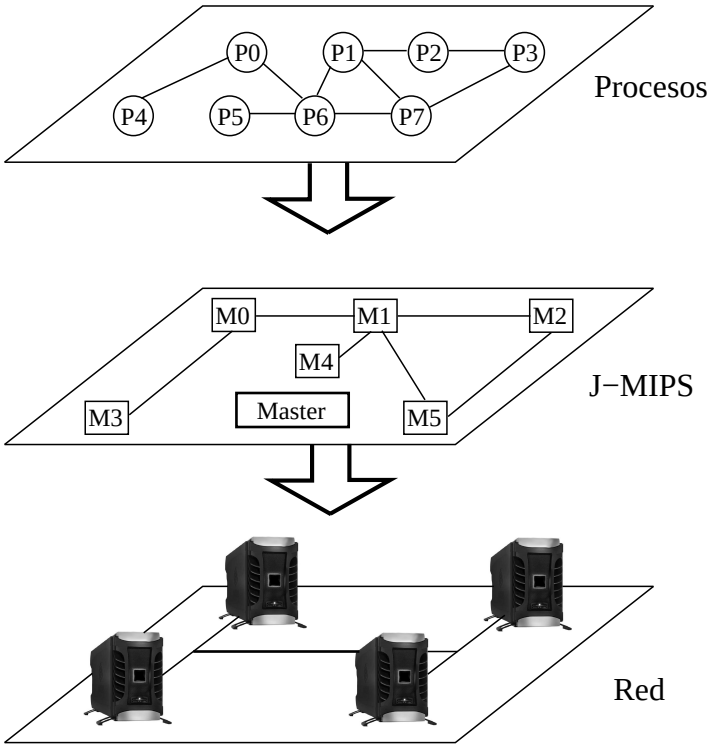

Figura 3.2: *Mapeo de dos niveles. Los procesos P_i son mapeados a las computadoras virtuales M_k, que a su vez, son mapeadas a la red física.*

La computadora virtual maestra sirve para comunicar al sistema con el usuario y realiza tareas administrativas y de inicio del sistema. Las tareas más importantes

[1]Escogimos el nombre *Manager* porque como vemos más adelante, son administradores de mensajes y de ejecución de procesos.

que desempeña son:

- Servir como interfaz de comunicación entre el sistema completo y el usuario.
- Realizar el mapeo de procesos especificado por el usuario en la red virtual.
- Realizar las conexiones virtuales entre computadoras virtuales.
- Supervisar la ejecución del programa paralelo e informar al usuario.

Las computadoras virtuales esclavas son el resto de los nodos que conforman la red virtual. Estas ejecutan los procesos asignados a ellas y se conectan entre sí para formar una *Topología Virtual* independiente de la topología física. Cada nodo de la red virtual tiene las siguientes funciones:

- Ejecutar los procesos que el objeto *Master* le asigne.
- Enviar los mensajes designados por algún proceso al proceso de destino.
- Comunicar al objeto *Master* el estado de la ejecución de los procesos locales.
- Entregar al objeto *Master* los mensajes de los procesos dirigidos al usuario.

Finalmente, los procesos *RemoteJob* representan el trabajo que el programador desea que se realice. Estos procesos cuentan con primitivas de comunicación similares a las del patrón de *paso de mensaje* (sección 1.4.1). Son especificados y programados por el usuario y se entregan al objeto *Master*; el cual a su vez, los entrega a las computadoras virtuales esclavas quienes los ejecutan.

El sistema completo funciona de la siguiente forma:

1. El programador inicia el proceso principal de las computadoras virtuales esclavas, que esperan instrucciones del objeto *Master*. Este proceso simboliza el mapeo de las computadoras virtuales en las computadoras físicas.
2. El programador especifica al objeto *Master* la lista de computadoras virtuales disponibles y su ubicación física en el cluster, así como el mapeo de los procesos a los objetos *Manager*.
3. El programador proporciona al objeto *Master* los procesos que desea que se ejecuten en la red virtual.
4. El programador indica al objeto *Master* que inicie la ejecución del sistema.

5. El objeto *Master* establece comunicación con las computadoras virtuales esclavas y les proporciona los procesos que deben ejecutar y la información necesaria para establecer la topología virtual de la red.

6. El objeto *Master* sincroniza las conexiones entre los objetos *Manager*.

7. Cada *Manager* ejecuta los procesos asignados a ella administrando los mensajes enviados entre procesos.

8. Cuando todos sus procesos han sido finalizados, cada objeto *Manager* avisa al objeto *Master* que su trabajo ha finalizado.

9. Cuando todos los objetos *Manager* han terminado, el *Master* termina la ejecución del sistema.

El lector puede notar que el programador de aplicaciones paralelas solo está involucrado en los primeros 4 puntos de la lista anterior. Esto significa que la labor del programador se reduce a tres tareas: Especificar los procesos, el mapeo de las computadoras virtuales en las físicas y el mapeo de los procesos a las computadoras virtuales. A primera vista, éstos parecen ser los pasos necesarios para cualquier sistema que permita ejecutar procesos en múltiples computadoras; sin embargo, en este trabajo intentamos facilitar estas tareas con el objetivo de que el programador se enfoque más en los procesos que deben ejecutarse y menos en las tareas de mapeo y comunicación de dichos procesos.

La figura 3.3 esquematiza el funcionamiento de J-MIPS. Primero el mapeo y los procesos son especificados en el objeto *Master* mientras los objetos *Manager* permanecen en espera. Después el *Master* establece la conexión con los *Managers* y les envía el mapeo de conexiones virtuales y los procesos que cada una debe ejecutar. En el tercer paso, las computadoras virtuales esclavas establecen la topología virtual de la red y ejecutan los procesos que les fueron asignados. Por último, los *Managers* indican al *Master* que han terminado su trabajo y el sistema finaliza.

3.3. Diseño de la biblioteca J-MIPS

3.3.1. Buffers y comunicación asíncrona

En las siguientes secciones hacemos uso recurrente de Buffers para amortiguar el tiempo de espera producido en las comunicaciones entre procesos. Por el resto de esta tesis, consideramos un *Buffer* como una cola de prioridades que respeta el esquema FIFO[2] de las estructuras de datos y que contiene operaciones *read()* y

[2]First In First Out

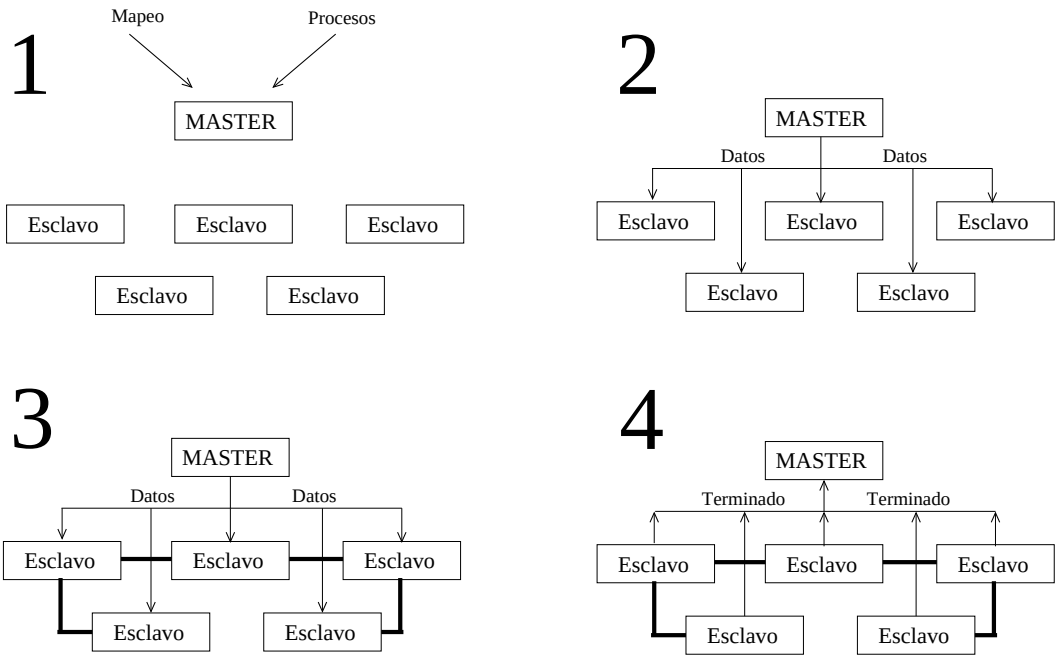

Figura 3.3: *Esquema de la biblioteca J-MIPS.*

write() que leen y escriben elementos en dicha cola.

La construcción de un Buffer nos remite al problema del *productor-consumidor* que no tratamos en este trabajo, pero lo explicamos con el siguiente ejemplo: supongamos que existe un proceso llamado PA, quien en algún momento de su ejecución desea enviar un dato a otro proceso llamado PB, mediante la llamada *send(PB,Data)* como en la sección 1.4.1. A su vez, el proceso PB recibe en algún momento los datos enviados por PA mediante la llamada *receive(PA,Data)*. Ambos procesos deben establecer la comunicación al mismo tiempo mediante estas llamadas para intercambiar datos. Sin embargo, si uno de ellos, por ejemplo PA llega a la llamada *send(PB,Data)* antes de que PB llegue a la llamada *receive(A,Data)*, PA debe bloquear su ejecución hasta que PB ejecute la llamada correspondiente que le permita recibir datos. De igual manera, si PB llega primero a la llamada *receive(A,Data)*, este debe esperar hasta que PA envíe los datos para poder continuar su ejecución (figura 3.4).

Para disminuir estos tiempos de espera, incluimos un Buffer entre PA y PB que llamaremos *SyncBuffer* y que sirve como un deposito donde PA puede almacenar datos destinados a PB sin esperar hasta que este último esté dispuesto a recibir-

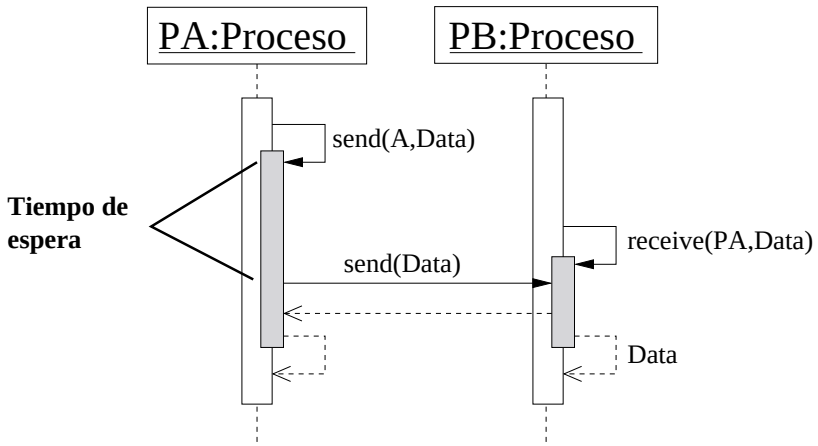

Figura 3.4: *Diagrama UML de secuencia de la comunicación entre dos procesos por paso de mensajes de* ***forma síncrona****. El proceso PA debe detener su ejecución hasta que PB este dispuesto a recibir los datos.*

los. De igual forma, el proceso PB puede leer datos del buffer en vez de pedírselos directamente a PA, con lo cual PB no detiene su ejecución hasta que PA envíe los datos (figura 3.5).

Así, la llamada *send(PB,Data)* ejecuta el método *buffer.write(Data)* del buffer de datos. Análogamente, la llamada *receive(PA,Data)* lee datos del buffer mediante una operación parecida a *Data = buffer.read()*.

El método *write(Data)* del buffer agrega el dato *Data* al final de la cola de prioridades. El método *read()* regresa la cabeza de dicha cola. Ambos métodos de acceso al buffer son atómicos para evitar corrupción en los datos.

Debemos considerar que un buffer no puede crecer indefinidamente, por lo cual debe existir una variable *maxSize* que controle el tamaño máximo del buffer. El acceso al tamaño del buffer en todo momento se da por medio del método *size()* que devuelve el número de elementos en el buffer al momento de la llamada al método; esta operación también es atómica.

Esta estructura lleva a dos casos a los que debemos atender:

1. **El proceso PA envía información al buffer, pero éste ha alcanzado su tamaño máximo y no se pueden agregar más elementos.** En este caso, el buffer bloquea la ejecución del proceso PA hasta que el proceso PB

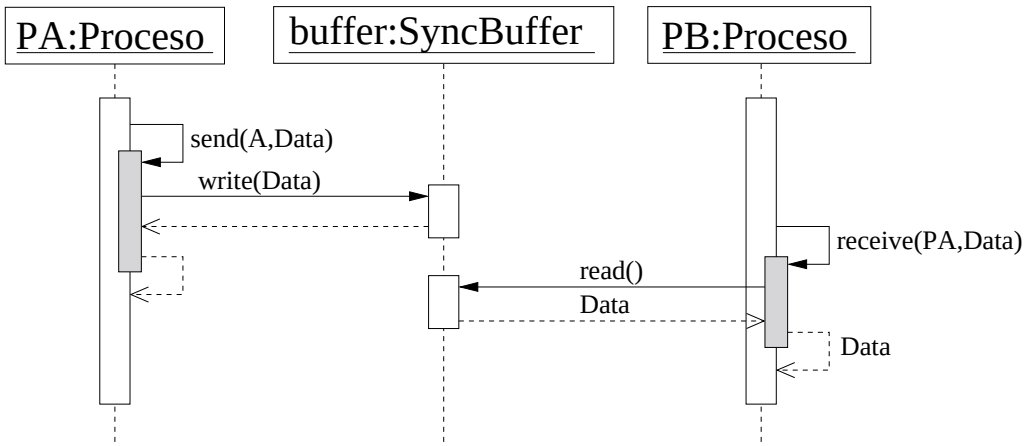

Figura 3.5: *Diagrama UML de secuencia de la comunicación* ***asíncrona*** *entre dos procesos a través de un buffer. Se ha eliminado el tiempo de espera.*

saque un dato de la cola.

2. **El proceso PB ejecuta la llamada *receive(Data)*, pero el buffer no contiene ningún elemento**. En este caso, el buffer debe bloquear la ejecución del proceso PB hasta que el proceso PA escriba un dato en la cola.

La figura 3.6 muestra el diagrama de clases del buffer que proponemos.

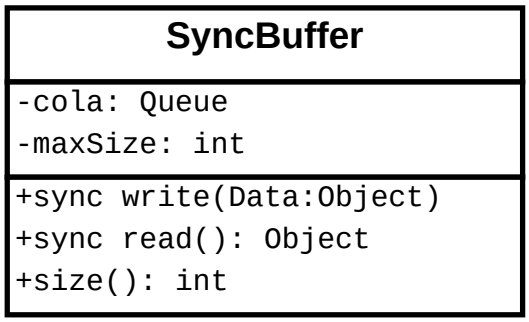

Figura 3.6: *Diagrama UML de clase para SyncBuffer.*

3.3.2. Mensajes

Los datos en J-MIPS fluyen entre los componentes del sistema a través de *mensajes*. Estos mensajes los dividimos en dos tipos:

- Mensajes Maestros.
- Mensajes regulares.

Los *Mensajes Maestros* son transmitidos entre el objeto *Master* y los *Managers* y van acompañados de un identificador que señala el remitente del mensaje. Un *Mensaje Maestro* puede ser una instrucción, un mensaje de error, un mensaje para el usuario o un simple dato; debido a ello, al mensaje también se adjunta un número entero que representa el *tipo de mensaje*. Un mensaje maestro entonces, es una estructura que se compone de los siguientes elementos:

- Dato que debe ser enviado o recibido.
- Identificador del emisor del mensaje.
- Tipo de mensaje.

El "Tipo de mensaje" encapsula una instrucción que el destinatario puede interpretar. Esto es, cuando se recibe un mensaje *MasterMessage*, quien lo recibe obtiene la siguiente información:

- El emisor del mensaje.
- Los datos que contiene.
- Qué debe hacer con esos datos.

El diagrama de clases 3.7 muestra la estructura de los mensajes maestros de J-MIPS. Además de los datos que ya explicamos, la clase *MasterMessage* cuenta con un constructor que acepta los datos, el remitente y el tipo de mensaje como parámetros, así como con métodos de acceso a estos parámetros. Una vez que un objeto MasterMessage ha sido creado, sus atributos no pueden ser modificados.
Aunque aún no hemos dicho nada sobre la estructura de los procesos que el usuario quiere ejecutar, podemos adelantar que todos tienen un identificador único en el sistema y es de tipo String[3].

Los *Mensajes Regulares*, que de ahora en adelante llamaremos simplemente *Mensajes* son estructuras parecidas a los *Mensajes Maestros*. Estos mensajes fluyen exclusivamente entre las computadoras virtuales esclavas y contienen la información enviada entre los procesos que se ejecutan en ellos. Dicho de otra forma, los *Mensajes Regulares* son los datos enviados y recibidos por los procesos especificados por el usuario. A estos datos se les añaden dos indicadores más: Quién envía

[3]String es el tipo de dato que utiliza Java para representar a las cadenas de caracteres

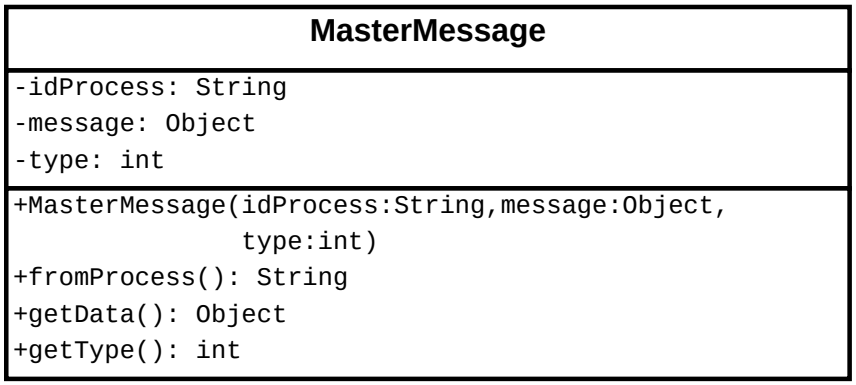

Figura 3.7: *Diagrama UML de clase para el Mensaje Maestro.*

el dato y a quién se lo envía. Esta información es necesaria para que los nodos virtuales esclavos puedan entregar correctamente el mensaje a su destino. Una estructura de *Mensaje* como ésta puede ser vista como una carta dentro de un sobre que debe entregarse por correo: En el sobre debe aparecer la información del remitente y el destinatario, y debe contener la carta que debe ser entregada.

Podemos entonces interconectar procesos mediante paso de mensajes si ponemos los siguientes datos en la misma estructura:

- El dato que debe ser enviado o recibido.
- El identificador del proceso que envía el mensaje.
- El identificador del proceso que recibe el mensaje.

En J-MIPS, integramos estos elementos en la clase Message, representada por el diagrama de clases 3.8.

Al igual que en la clase *MasterMessage*, la clase *Message* contiene un constructor y métodos de acceso a los atributos que mencionamos. Asimismo, los objetos *Message* no pueden ser modificados después de su creación.

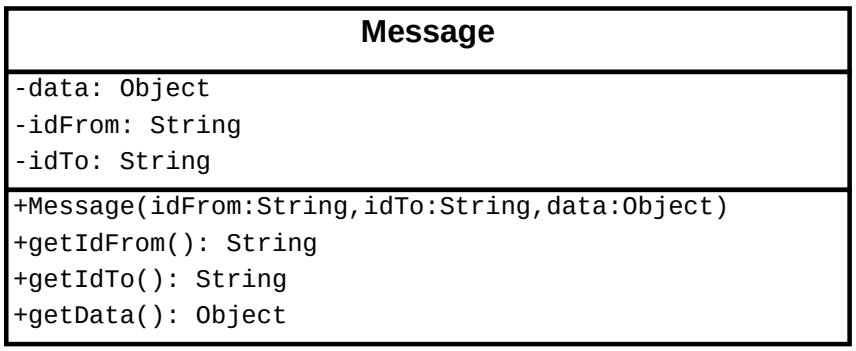

Figura 3.8: *Diagrama UML de clase para los mensajes regulares.*

3.3.3. Mapeo de procesos

Como mencionamos antes, el mapeo de los procesos se realiza en dos niveles:

1. Las computadoras virtuales son mapeadas a las computadoras físicas.
2. Los procesos son mapeados a las computadoras virtuales.

El usuario debe especificar los dos niveles de mapeo en un archivo de texto y después pasarlo al objeto *Master*. Llamamos a este archivo *Archivo de configuración*, en el cual, cada computadora virtual debe especificarse mediante una línea como la siguiente:

$$\text{Manager} = (\text{IP} \ , \ \text{PUERTO} \ , \ \text{ID})$$

Donde IP es la dirección IP (o el nombre asociado a esta) de la computadora física en la que reside la computadora virtual, PUERTO es el puerto que la computadora virtual utiliza para establecer las comunicaciones e ID es el identificador de la computadora virtual.

Por ejemplo, considérese la línea:

$$\text{Manager} = (192.168.1.100 \ , \ 3360 \ , \ \text{M1})$$

Al agregar esta línea al archivo de configuración, se pretende decir al objeto *Master* *"Existe una computadora esclava virtual llamada M1 en la dirección 192.168.1.100, la cual habla y escucha a través del puerto 3306"*. Como mencionamos antes, al momento de pasar el archivo de configuración al objeto *Master*, debe existir en ejecución un objeto *Manager* en la computadora 192.168.1.100 escuchando por el puerto 3306 y llamado M1, esperando que el objeto *Master* se conecte con él.

Con este esquema, es posible tener más de un *Manager* en la misma computadora física; sin embargo, cada computadora virtual dentro de una computadora física debe escuchar por un puerto diferente y debe tener diferente ID. Por ejemplo, son válidas las siguientes especificaciones en el mismo archivo de configuración:

Manager = (192.168.1.100 , 3360 , M1)
Manager = (192.168.1.100 , 3361 , M2)

Vemos que ambos *Managers* (Computadoras virtuales) se encuentran en la misma computadora física con dirección 192.168.1.100; sin embargo, el host M1 escucha a través del puerto 3360, mientras que el host M2 lo hace a través del puerto 3361.

Las siguientes limitaciones deben ser consideradas:

- No es válido que dos *Managers* tengan el mismo identificador. De existir este error, el objeto *Master* lanza una excepción al intentar procesar el archivo.
- No es válido que dos *Managers* que comparten la misma dirección IP compartan también el mismo puerto. Si este error existe, puede haber conflictos de comunicación al intentar ejecutar el sistema. Si se desea ejecutar más de un *Manager* con la misma dirección IP, estos deben ser ejecutados en Threads diferentes y deben utilizar diferente puerto.
- Para cada línea de *Manager* en el archivo de configuración, debe existir un correspondiente proceso esclavo que cumple las especificaciones de dicha línea. Si existen líneas de *Manager* para las cuales no existe un nodo esperando conexión, el objeto *Master* lanza una excepción al intentar procesar el archivo.

Una vez mapeadas las computadoras virtuales, cada proceso debe ser mapeado a éstas agregando al archivo de configuración una línea como la siguiente:

Process = (ID , MANAGERID , [CONNECTIONS])

Donde ID es el identificador del proceso, MANAGERID es la computadora virtual en la cual está alojado dicho proceso y [CONNECTIONS] es la lista de los procesos conectados al proceso ID. Considérese la siguiente línea en el archivo de configuración:

Process = (P2 , M1 , [P1,P4,P7])

La línea anterior significa *"Existe un proceso llamado P2, que debe ser ejecutado en la computadora virtual M1 y que está conectado a los procesos P1,P4 y P7"*.

Al especificar las conexiones de cada proceso, se define la topología de los procesos en ejecución. De esta información y del mapeo de los procesos a las computadoras virtuales, el objeto *Master* deduce la topología de la red virtual y ordena a los nodos esclavos (*Managers*) llevarla a cabo. Abordamos ese tema más adelante en este capítulo.

Es posible agregar comentarios al archivo de configuración agregando el símbolo # al principio de la línea que se desea comentar.

Finalmente, los espacios entre palabras y las líneas en blanco en el archivo de configuración no son tomados en cuenta por el objeto *Master*. En ese sentido, las líneas:

```
Process = (P2 , M1 , [P1,P4,P7] ) y
Process=(P2,M1,[P1,P4,P7])
```

significan exactamente lo mismo.

Los siguientes puntos deben ser considerados al especificar el mapeo de los procesos:

- No puede haber más de una línea de especificación de procesos con el mismo identificador. Esto quiere decir que el identificador de cada proceso debe ser único. Esta característica es para evitar conflictos al decidir a qué proceso debe entregarse un mensaje. Si en el archivo de configuración existiera más de un proceso con el mismo identificador, el objeto *Master* lanza una excepción al procesar el archivo y detiene su ejecución informando al usuario.
- Un proceso no puede estar conectado consigo mismo, pues no deseamos que un proceso sea capaz de enviarse mensajes a sḿismo, por lo cual un identificador de proceso no puede aparecer en la lista de procesos a los que él mismo está conectado. Como en el caso anterior, el objeto *Master* es capaz de detectar cuando en una linea *Process* del archivo de configuración, aparece en la lista de conexiones el identificador del proceso que se está especificando en la misma linea. Este error provoca un alto total en la operación del objeto *Master*, quien termina lanzando una excepción e informando al usuario.
- La información de los procesos especificados en el archivo de configuración debe coincidir exactamente con la lista de procesos dados al objeto *Master*.

Más adelante en este capítulo vemos con detalle que el objeto *Master* obtiene diversa información a partir del archivo de configuración. Más específicamente de las lineas de especificación de procesos. Parte de esta información es utilizada también tanto por los *Managers* como por los procesos mismos para la correcta ejecución del sistema. Esto implica que la información contenida en las lineas de especificación de procesos debe ser exacta, lo cual quiere decir lo siguiente:

- Por cada identificador de proceso en el archivo de configuración, debe existir un proceso en el objeto *Master* con el mismo identificador. El objeto *Master* es capaz de identificar un error en esta restricción, deteniendo su ejecución y avisando al usuario mediante una excepción.
- Si para un proceso dado A, este envía un mensaje a otro proceso B, entonces B debe aparecer en la lista de conexiones del proceso A en el archivo de configuración y A debe aparecer en la lista de conexiones del proceso B en dicho archivo. El objeto *Master* no es capaz de detectar un error en esta restricción, pues para poder hacerlo, debería revisar el código de las llamadas *send* y *receive* en cada proceso. Sin embargo, un error de esta naturaleza es detectado en tiempo de ejecución provocando que el sistema se detenga por completo.

El código 3.1 es un ejemplo de un archivo de configuración utilizado para construir el mapeo representado por la figura 3.9. En esta figura, se representa únicamente la red de procesos y la red de computadoras esclavas virtuales que los ejecutan mapeadas en las computadoras físicas. El objeto *Master* no está incluido en el diagrama pues es él quien construye éstas dos redes. Los números en círculos representan procesos; las líneas que unen a los procesos son las conexiones entre ellos. Los polígonos punteados rodeando a los procesos representan a las computadoras virtuales que los ejecutan (*Managers*). Finalmente, los rectángulos continuos representan a las computadoras físicas, a las cuales son mapeadas las computadoras virtuales.

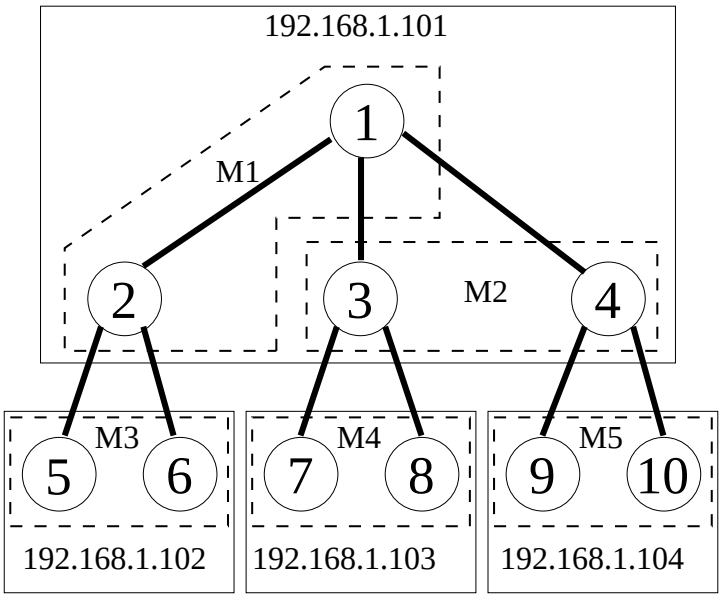

Figura 3.9: *Red construida mediante el archivo de configuración del código 3.1.*

Código 3.1: Ejemplo de archivo de configuración

```
1  #Las computadoras virtuales
2  Manager = (192.168.1.101,3300, m1)
3  Manager = (192.168.1.101,3301, m2)
4  Manager = (192.168.1.102,3300, m3)
5  Manager = (192.168.1.103,3300, m4)
6  Manager = (192.168.1.104,3300, m5)
7
8  #Los procesos
9  Process = (p1, m1, [p2,p3,p4])
10 Process = (p2, m1, [p1,p5,p6])
11 Process = (p3,m2,[p1,p7,p8])
12 Process = (p4,m2,[p1,p9,p10])
13 Process = (p5,m3, [p2])
14 Process = (p6,m3, [p2])
15 Process = (p7,m4, [p3])
16 Process = (p8, m4, [p3])
17 Process = (p9, m5, [p4])
18 Process = (p10,m5, [p4])
```

3.3.4. Procesos de ejecución remota

Los procesos que se ejecutan en la red de computadoras virtuales o (*Managers*) son vistos por J-MIPS como procesos ligeros o Threads. Asumimos que debe haber

intercambio de información entre ellos. Esta información debe poderse convertir en un flujo finito de bits; es decir, los datos transmitidos entre procesos deben ser serializables. El programador debe especificar el comportamiento de estos procesos para que realicen el trabajo que desea y después proveer al objeto *Master* con estos procesos para que este último los asigne a cada *Manager*. Sin embargo, estos procesos deben cumplir los siguientes requerimientos:

- Cada proceso debe tener un identificador en forma de cadena de caracteres.
- Un proceso debe poder enviar un dato serializable[4] Ã§ a otro proceso, especificando únicamente el identificador del proceso de destino y el dato que se desea transmitir.
- Un proceso debe poder recibir un dato serializable desde otro proceso especificando únicamente el identificador del proceso emisor del mensaje. Además, esta operación debe poder realizarse cuando el proceso receptor lo desee; por supuesto, si no ha llegado dicho mensaje, el proceso debe bloquearse hasta que el emisor lo envíe.
- Todos los procesos deben ser capaces de enviar mensajes al *Master* para dar información al usuario.

Los primeros tres puntos de las especificaciones anteriores cumplen con el patrón de paso de mensajes entre procesos de la sección 1.4.1. El último punto permite a los procesos comunicar su estado de ejecución al usuario.

Para cumplir con estas características, utilizamos objetos que llamamos *RemoteJob*. Cada proceso debe ser una instancia de la clase que posee el mismo nombre. Sin embargo, hacemos uso de la herencia de la programación orientada a objetos para que el usuario pueda definir el comportamiento de los procesos y que éstos cumplan con las características especificadas anteriormente. Esto es, la clase *RemoteJob* representa a un proceso que cumple con el comportamiento básico que señalamos, pero no tiene un *trabajo de procesamiento asociado*. El programador debe construir sus procesos creando clases que "hereden" de *RemoteJob* y donde se especifique el trabajo que dichos procesos realizan[5].

Para enviar un mensaje desde un *RemoteJob* a otro, el emisor debe especificar el dato que debe ser enviado y el receptor del dato mediante el método *sendMessage(String destino, Serializable dato)*. Este método simplemente encapsula los

[4]En Java, solo los datos serializables pueden ser transmitidos a través de la red.

[5]Como trabajo, nos referimos a las instrucciones que el procesador debe ejecutar por cada proceso.

argumentos en un mensaje de tipo *Message* como el de la sección 3.3.2 y deja la labor de encontrar al proceso destino y la transmisión del mensaje al *Manager* donde reside el proceso emisor.

Para recibir un mensaje, el proceso es un poco más complicado. El proceso receptor contiene una tabla Hash llamada *incMessages*, cuyos códigos son los identificadores de los procesos conectados al receptor y cuyos valores son buffers *SyncBuffer* que almacenan los mensajes recibidos. De esta forma, cuando un mensaje llega al *Manager* que contiene al receptor, el*Manager* entrega el mensaje al proceso, el cual identifica al emisor y almacena el mensaje en el buffer asociado a dicho emisor mediante un método llamado *adminMessage(Serializable dato, String emisor)*. Cuando el proceso receptor requiere un dato del emisor de dicho mensaje, hace una llamada al método *receiveMessage(String emisor)*. Este método busca el mensaje en el buffer de mensajes de llegada asociado al proceso emisor, al cual accede por medio de la tabla Hash *incMessages*.

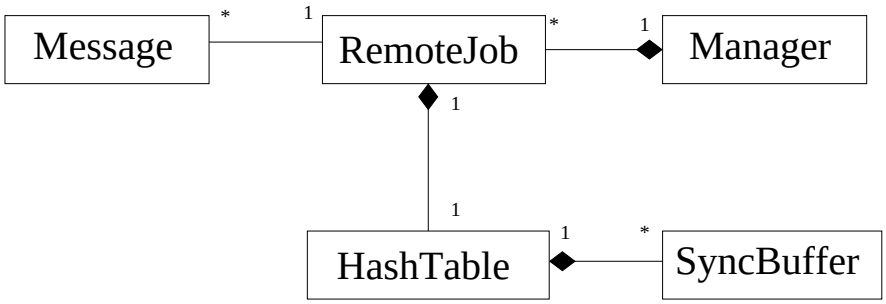

Figura 3.10: *Diagrama UML de clases de los elementos involucrados en la comunicación entre dos procesos RemoteJob.*

En la figura 3.10 esquematizamos en un diagrama de clases los componentes que utiliza un proceso *RemoteJob* y que intervienen en la comunicación.

Podemos ilustrar estos conceptos con un ejemplo que ilustramos en la figura 3.11 por medio de un diagrama de secuencia: Supóngase que se tienen dos procesos *RemoteJob pa* y *pb*, *pa* se ejecuta en el *Manager ManA* y *pb* se ejecuta en el *Manager ManB*. En algún momento de la ejecución del RemoteJob *pa*, éste envía un número entero *int* al *RemoteJob pb* mediante la llamada al método *sendMessage(pb,int)*. El proceso *pa* encapsula el dato en un *Message men*, donde también guarda el identificador de destino *pb*, así como el suyo propio y delega la tarea de enviar el mensaje al *Manager ManA* quien envía el mensaje y continua con su

trabajo. Por su parte, el *Manager ManB* recibe el mensaje y "observa" que es un mensaje para *pb*. Después, *ManB* hace una llamada a *pb.adminMessage(int,pa)*, quien obtiene de la tabla Hash *incMessages* de mensajes de llegada, al buffer de tipo *SyncBuffer* asociado a los mensajes que lleguen desde *pa*, llamado *aBuffer* y agrega a *int* en dicho buffer. En algún momento de la ejecución de *pb*, este *RemoteJob* requiere al entero *int* de *pa* mediante la llamada *receiveMessage(pa)*. En vez de comunicarse con *pa* directamente, *pb* busca el mensaje en el buffer de mensajes de llegada asociado a *pa*, el cual se encuentra en la tabla Hash *incMessages*. A partir de este punto, pueden ocurrir dos casos:

- El mensaje se encuentra en el buffer esperando ser utilizado. En cuyo caso, la llamada *receiveMessage(pa)* regresa el entero *int* enviado desde *pa*.
- El buffer de llegada está vacío. Eso significa que *pa* aún no ha enviado al entero *int*. En cuyo caso, *receiveMessage(pa)* se bloquea hasta que haya un elemento en el buffer.

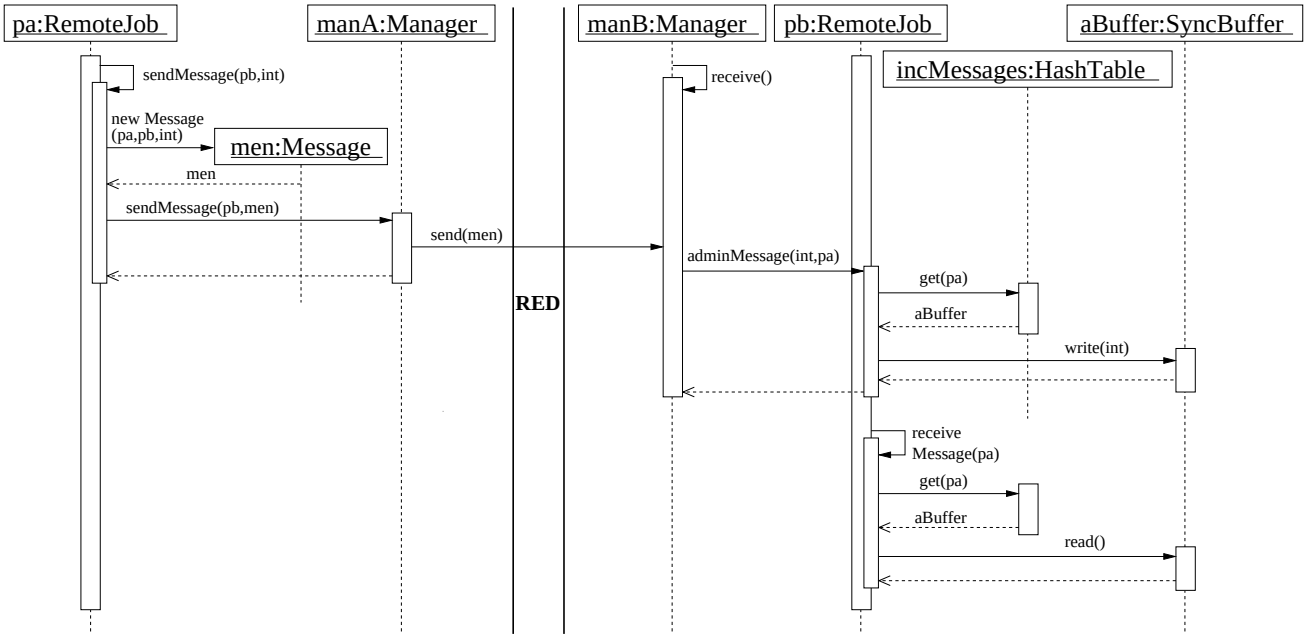

Figura 3.11: *Diagrama UML de secuencia de la comunicación entre dos procesos RemoteJob.*

Una vez que el *RemoteJob* ha terminado su ejecución, el usuario debe notificar este hecho por medio de un método llamado *finished()*. En la siguiente sección veremos la importancia de este método.

Para que la tabla Hash de buffers de llegada pueda ser poblada, el *RemoteJob* debe conocer el identificador de todos los procesos a los cuales está conectado. Para ello, cada *RemoteJob* cuenta con una lista ligada de cadenas de caracteres llamada *connectedProcesses* que los almacena y un método llamado *setConnectedProcesses* para modificar dicha lista.

Además de estos componentes principales, cada *RemoteJob* cuenta con un método *sayToMaster(Serializable data)* que envía mensajes serializables al objeto *Master*.

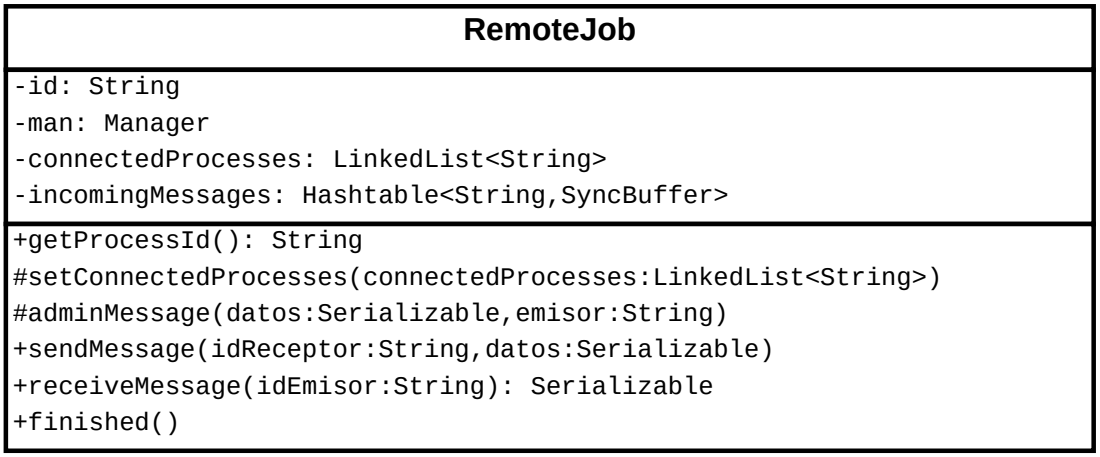

Figura 3.12: *Diagrama de la clase RemoteJob.*

La figura 3.12 muestra el diagrama de clases de RemoteJob. Los métodos *setConnectedProcesses* y *adminMessage* solo deben ser utilizados por otros componentes de la biblioteca J-MIPS, por lo cual su acceso se restringe a protegido.

3.3.5. La conexión entre computadoras reales

Para que dos procesos que se encuentran en distintas computadoras físicas puedan intercambiar información, es necesario crear un enlace lógico entre estas computadoras. En J-MIPS se utilizan dos enlaces de este tipo: Las conexiones entre el objeto *Master* y los *Managers* y las conexiones entre las computadoras esclavas virtuales.

En J-MIPS existe una clase llamada *Connection* que representa la conexión física entre dos computadoras y contiene información sobre la ubicación de éstas en la red. Es decir, se almacenan los identificadores, direcciones IP y puertos de las computadoras involucradas en la conexión. Dentro de todos los pasos que se

requieren para comunicar dos procesos *RemoteJob*, es un objeto *Connection* el que se encarga de comunicarle a la Maquina Virtual de Java su deseo de enviar o recibir datos, así como de establecer comunicación con otro objeto *Connection* que se encuentre en otra computadora (o en la misma por medio de un puerto diferente).

La clase *Connection* realiza las siguientes funciones:

- Establece la comunicación con otro objeto *Connection*.
- Envía un objeto de tipo *Serialiable* a otro objeto *Connection*.
- Recibe objetos de tipo *Serializable* desde otro objeto *Connection*.
- Cierra la conexión establecida.

Para explicar la forma en la que dos objetos *Connection* establecen la conexión entre ellos, podemos usar una analogía de la vida cotidiana. Supongamos que dos personas desean hablar por teléfono, esto no puede ocurrir si las dos esperan la llamada o si las dos la hacen al mismo tiempo. Una debe esperar la llamada y la otra debe hacerla. Una vez que la que espera la llamada levanta la bocina al sonar el tono, las dos pueden intercambiar información. Los objetos *Connection* funcionan bajo el mismo principio. Uno de ellos debe anunciar al otro que quiere establecer conexión con él, mientras que el segundo debe esperar a que el primero haga contacto. Este es el modelo de comunicación Cliente-Servidor. En J-MIPS, cada objeto *Connection* es cliente y servidor de su contraparte en la comunicación y el objeto *Master* es quien determina en todo momento quién debe realizar la petición de conexión y quien debe esperarla. Abordamos este tema con detalle en la sección 3.3.7. Debemos señalar que el objeto *Connection* que espera la petición de conexión, debe saber exactamente de quien viene. En nuestra analogía, esto quiere decir que incluso antes de que suene el teléfono, quien levanta el auricular sabe exactamente quién lo va a llamar.

Una vez que la comunicación entre los objetos *Connection* ha sido establecida, puede darse el intercambio de datos entre ellos por medio de métodos *sendMessage(data)* y *receiveMessage()*. Esta comunicación es asíncrona por parte del emisor y síncrona por parte del receptor.

Finalmente, un objeto *Connection* es capaz de cerrar la conexión con su contraparte. En nuestra analogía, esto equivale al hecho de que cualquiera de los dos participantes en la llamada puede colgar el teléfono en el momento que quiera. El que queda del otro lado de la línea, no tiene más remedio que colgar el

teléfono. De igual forma, los objetos *Connection* pueden cerrar la conexión en cualquier momento [6]. Sin embargo, quien la cierra debe avisar a su contraparte en la comunicación que haga lo mismo. De esta forma, ambos participantes consideran cerrada la conexión si uno de ellos decide terminar el proceso de comunicación.

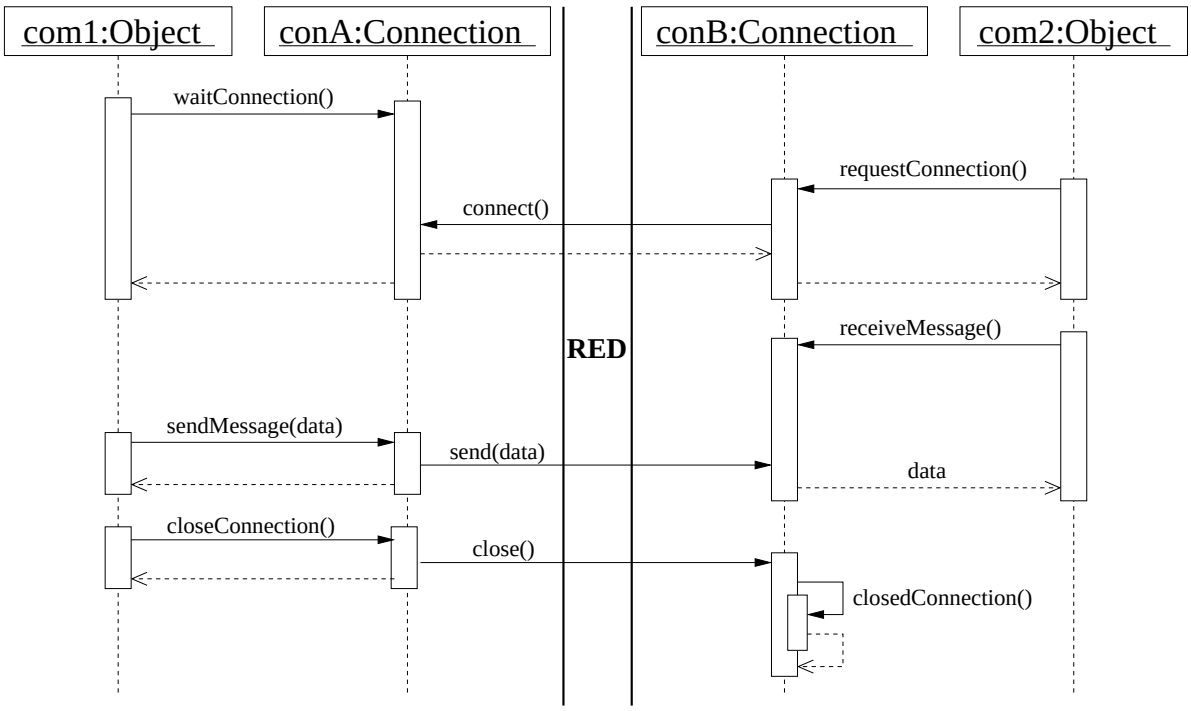

Figura 3.13: *Diagrama UML de secuencia de la interacción entre dos objetos Connection.*

La figura 3.13 muestra la interacción entre dos objetos de tipo *Connection* llamados *conA* y *conB*. Dado que los objetos *Connection* no se autocontrolan, asumimos que existen objetos externos que les dan instrucciones; *com1* en el caso de *conA* y *com2* en el caso de *conB*. Primero, *com1* le da instrucciones a *conA* de esperar una petición de conexión por parte de *conB*. En algún momento, *com2* instruye a *conB* para realizar una petición de conexión a *conA*. Cuando *conA* espera una petición de conexión y *conB* la realiza, la conexión entre ambos objetos *Connection* queda establecida y lista para intercambiar información.

[6]Cerrar una conexión quiere decir que se suspende la transmisión de datos, no es posible enviar mas datos a través de ese objeto *Connection* y el canal de comunicación (puerto) queda libre para que alguien más lo use.

Después, *com2* requiere un dato que proviene de *com1*. Para obtenerlo, llama al método *receiveMessage()* del objeto *Connection conB*, el cual se bloquea hasta recibir un dato desde *conA*. El objeto *com1* envía en algún momento un dato a *com2* mediante la llamada *sendMessage(data)* del objeto *conA*. Este método envía el dato sin bloquear su ejecución y devuelve el control al objeto *com1* que lo llamó. Cuando el método bloqueado *receiveMessage()* del objeto *conB* recibe los datos, continúa su ejecución regresando dichos datos al objeto *com2*.

Finalmente, *com1* ejecuta la llamada *closeConnection()* de *conA*, quien cierra la conexión con *conB*, no sin antes notificarle este hecho. *conB* recibe la notificación y considera a la conexión como cerrada.

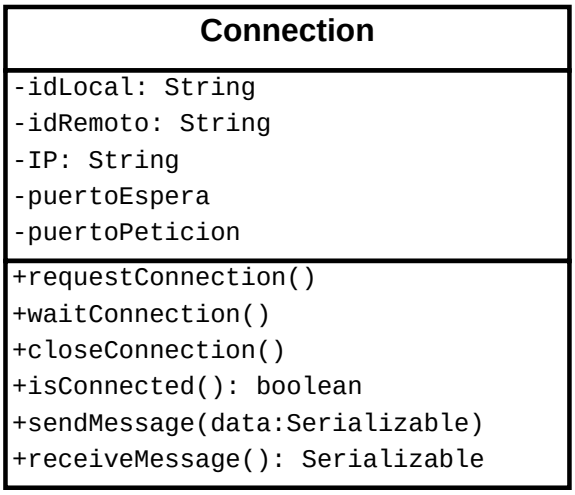

Figura 3.14: *Diagrama de la clase Connection.*

La figura 3.14 es el diagrama de la clase *Connection*. Este diagrama contiene solamente los elementos que mencionamos en esta sección y que interesan al diseño de J-MIPS. La clase real contiene otros atributos de más bajo nivel para realizar las conexiones por medio de sockets en Java. Sin embargo, el tratamiento de los aspectos técnicos se deja para el capítulo 5.

3.3.6. Computadoras esclavas virtuales

Los *Managers* son las computadoras *esclavas virtuales* que ejecutan los procesos *RemoteJob* que el *Master* les asigna. Las llamamos *Managers* porque son quienes administran el flujo de mensajes entre procesos, así como la ejecución de éstos. Las conexiones entre los *Managers* definen una topología

virtual independiente de la topología física de la red. Sin embargo, su principal función es administrar el tráfico de los mensajes que son transmitidos entre los procesos. Cuando un proceso envía un mensaje a otro, lo hace especificando el proceso de destino y el mensaje que debe enviar. El ***Manager*** donde el emisor reside debe saber si el proceso de destino es parte de los procesos que ejecuta o reside en otro *Manager*. De igual forma, todos los mensajes que llegan a los procesos de una computadora virtual esclava lo hacen a través de esta última. El *Manager* debe identificar hacia qué proceso va dirigido cada mensaje entrante y entregárselo indicándole quién fue el emisor.

Los *Managers* también mantienen comunicación con el objeto *Master* para indicarle el estado de su ejecución, así como para recibir instrucciones. Cuando son iniciados, los *Managers* no conocen los procesos que deben ejecutar ni están conectados a ninguna otra computadora virtual (incluyendo al *Master*). De hecho, ni siquiera conocen su identificador. Se inician como computadoras virtuales vacías que esperan instrucciones. El objeto *Master* se conecta con ellas para asignarles un nombre (identificador), la configuración de las conexiones que deben realizar con otros *Managers*, les asigna los procesos y les instruye ejecutarlos. Una vez que todos los procesos de un *Manager* han terminado, este último avisa ese hecho al *Master* y espera instrucciones.

La computadora virtual *Master* y la configuración

Cuando un *Manager* es iniciado, no sabe cuál es el trabajo que debe realizar. Esto significa lo siguiente:

- No tiene un nombre único que lo identifique de otros *Managers*.
- No tiene ningún proceso asignado para ejecución.
- No está conectado a ninguna otra computadora ni tiene información sobre las conexiones que debe realizar.
- No tiene información sobre la ubicación de los procesos remotos a los cuales se conectaran sus propios procesos.

Lo único que sabe es que debe esperar a que el *Master* haga conexión con ella. Elegimos este modelo para evitar que el objeto *Master* se sature de peticiones de conexión por parte de los *Managers*.

Una vez que el objeto *Master* ha establecido la conexión con el *Manager*, éste espera a que la primera le asigne un identificador y las conexiones que debe realizar con otros *Managers*. Esto quiere decir que el *Master* envía los identificadores,

direcciones IP y puertos de los *Managers* a los cuales el receptor del mensaje debe conectarse. Las conexiones de un *Manager* con los otros están representadas por una tabla Hash que almacena objetos de tipo *Connection* y que llamamos *physicalCons*, con entradas de tipo (Identificador,Connection), donde el código Hash Identificador es el identificador del *Manager* conectado y el valor *Connection* es la conexión con dicho *Manager*. El objeto *Master* envía esta tabla Hash. La figura 3.15 representa el hecho de que el vínculo entre los *Managers* se da a través de objetos *Connection*.

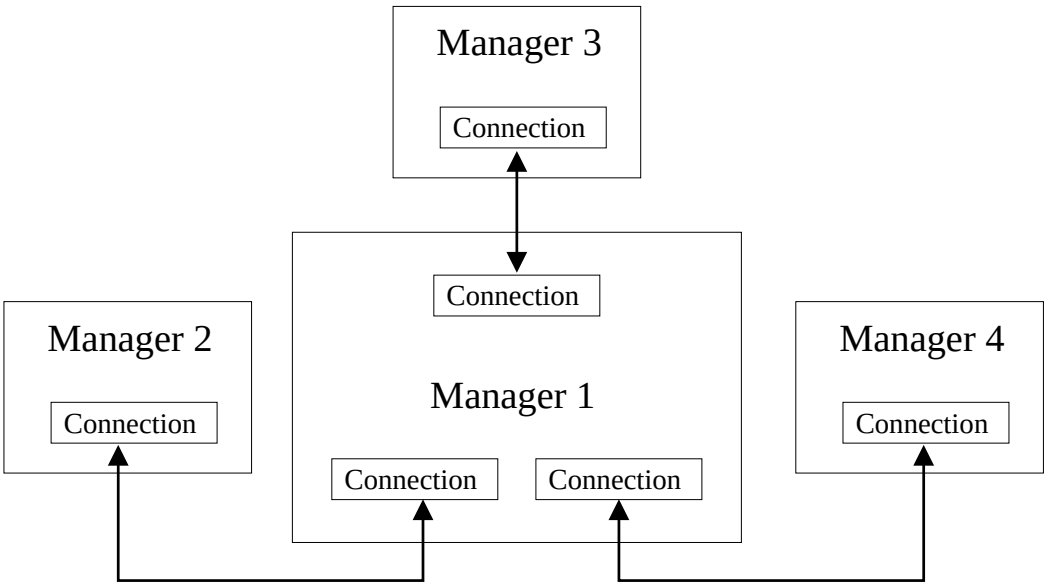

Figura 3.15: *Esquema de los Managers y las conexiones entre ellos.*

Supongamos que un proceso que reside en un *Manager* envía un mensaje a otro proceso que se encuentra en otro *Manager*. Para que el *Manager* del emisor pueda saber donde reside el proceso receptor, los *Managers* tienen una tabla Hash que llamaremos *pMapping*, donde cada entrada es de la forma (Proceso,Manager), el código Hash de la entrada es el identificador del proceso y el valor de dicho código es el identificador del *Manager* en el cual reside. Esta tabla Hash es enviada por el objeto *Master* después de enviar la información de las conexiones. La utilización de una tabla hash para manejar esta información permite que el tiempo de búsqueda sea constante.

El objeto *Master* envía por último los procesos que el *Manager* debe ejecutar en forma de una tabla Hash de valores (Identificador,RemoteJob), que llamamos

procs, donde el código Hash es el identificador del proceso y el valor asociado es un objeto RemoteJob.

Finalmente, una vez que el *Manager* ha sido alimentado con los elementos anteriores, queda a la espera de instrucciones por parte del objeto *Master*.

Hasta este punto, el *Manager* que inició siendo una computadora virtual sin ninguna información sobre su ejecución, ha sido alimentada con los elementos necesarios para iniciar su trabajo. Sin embargo, sigue siendo una computadora virtual desconectada de otros *Managers* y que no se encuentra ejecutando los procesos asignados. El siguiente paso es que el *Manager* se conecte con los otros *Managers* para formar la red virtual.

Conexiones iniciales

Para poderse comunicar con otras computadoras esclavas virtuales, los *Managers* utilizan dos métodos: *waitConnnectionFrom(Id)* y *connectToAll()*. El primero espera a que el *Manager* remoto con identificador Id establezca conexión con el *Manager* local. El segundo establece la conexión con todos los *Managers* remotos a los cuales el local está conectado.

Después de que el objeto *Master* envía la información especificada en la subsección anterior (3.3.6), los *Managers* esperan una instrucción del *Master* que les indica si deben esperar una conexión o establecerla con todas sus compañeras. Es decir, si la instrucción es "establecer conexión", el *Manager* ejecuta el método *connectToAll()*. Este método ejecuta uno por uno, el método *requestConnection()* de cada objeto *Connection* en la tabla Hash que simboliza la conexión con otros *Managers*. Una vez que una conexión ha sido establecida exitosamente, el *Manager* hace lo mismo con la siguiente conexión. Si la instrucción que llega es "esperar conexión de Id", el *Manager* ejecuta el método *waitConnectionFrom(Id)*. Este método es simplemente hacer que la conexión asociada al *Manager* Id ejecute el método *waitConnection()*.

Estos métodos son sensibles al estado de la conexiónes. El método *connectToAll()* establece conexión solo con aquellos *Managers* con los cuales no existe una conexión previa. De igual forma, el método *waitConnectionFrom(Id)* solo se ejecuta si no existe una conexión previa con el *Manager* de identificador Id.

La figura 3.16 muestra el procedimiento de conexión desde un *Manager* a sus tres compañeros. El objetivo es conectar al *Manager m1* con los *Managers m2,*

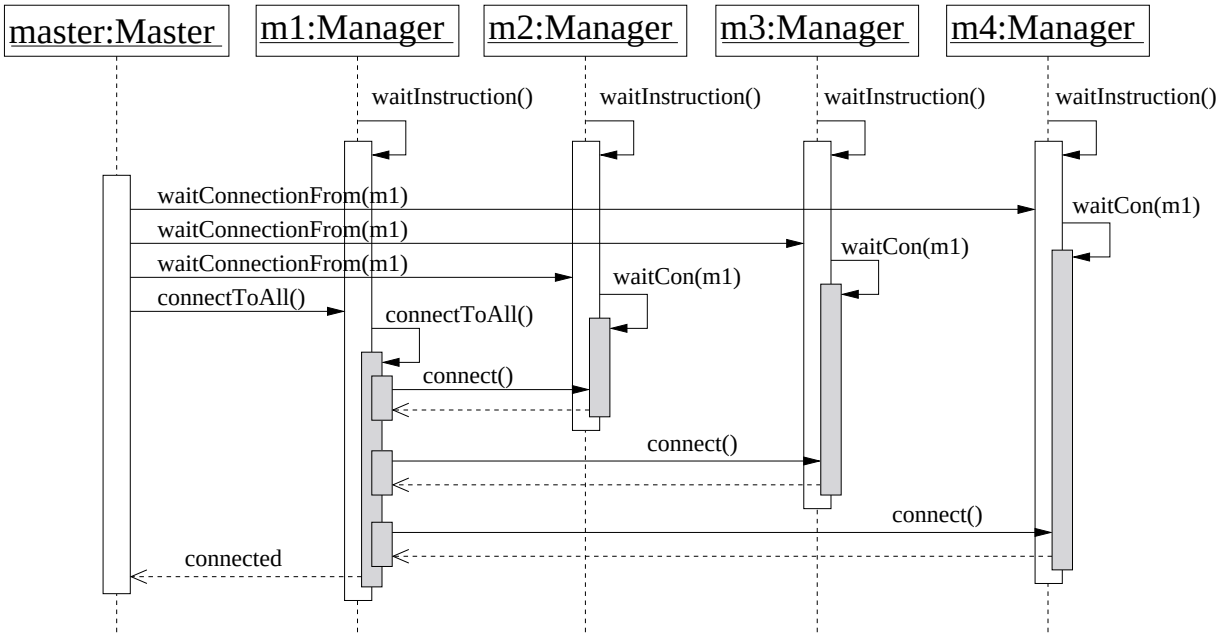

Figura 3.16: *Diagrama UML de secuencia de la conexión entre Managers organizados por el objeto Master.*

m3 y *m4*. Los *Managers* inician esperando una instrucción del objeto *Master*. Esta instruye a los *Managers m2, m3* y *m4* a que esperen a que el *Manager m1* se conecte a ellos. Después, el *Master* manda la instrucción al *Manager m1* para conectarse con todas las demás. Este último realiza las conexiones con los *Managers m2, m3* y *m4* uno por uno y termina avisando al objeto *Master* que ha finalizado.

Ejecución de los procesos

Si revisamos el escenario que tenemos hasta ahora, los *Manager* han sido provistos de procesos para ejecutar, una tabla de mapeo de procesos que indica hacia dónde deben ser enviados los mensajes y están completamente conectados con los *Managers* necesarios para enviar dichos mensajes. En este punto, los *Managers* esperan hasta recibir una instrucción del *Master* que indique que se debe iniciar la ejecución de los procesos. Cuando esta instrucción es recibida, sin embargo, la tarea de ejecutar procesos se delega al sistema operativo local.

Comunicación de dos niveles

Cuando un mensaje es enviado desde un proceso A a un proceso B, el *Manager* donde se ejecuta A debe decidir si el mensaje debe ser enviado a otro *Manager* (y a cuál) o solo entregado a algún proceso local (y a quién). Para esto, el *Manager* del emisor busca en su conjunto de procesos si hay alguno con el identificador del receptor del mensaje. Si ambos, emisor y receptor se encuentran en el mismo *Manager*, basta con que éste llame al método *adminMessage(mensaje,emisor)* del proceso de destino. Si el *Manager* no encuentra al receptor en su conjunto de procesos, eso quiere decir que el receptor se encuentra en otra computadora virtual esclava. Para encontrar en cuál, el *Manager* del emisor busca en la tabla Hash *pMapping* el identificador del proceso receptor y obtiene el identificador del *Manager* donde se ejecuta. Así, el *Manager* del emisor puede enviar el mensaje al *Manager* del receptor utilizando la conexión establecida entre ellos (figura 3.17).

En la recepción de datos, por cada conexión de tipo *Connection* establecida con otros *Managers*, el local pone en ejecución un sub-proceso que llamamos *Receiver* que se encarga de recibir mensajes desde esa conexión. Esto es para evitar que el bloqueo en el método *receiveMessage()* de dicha conexión bloquee también la ejecución normal del *Manager*. Cuando un mensaje es recibido, éste llega en forma de un objeto de tipo *Message*. El *Receiver* recupera los datos encapsulados (emisor, receptor y mensaje) y entrega los datos al proceso de destino mediante una llamada al método *adminMessage(datos,emisor)* (3.18).

La figura 3.19 muestra las relaciones involucradas entre los componentes que intervienen en la comunicación entre dos procesos *RemoteJob*, y por tanto, entre los *Managers* que los contienen.

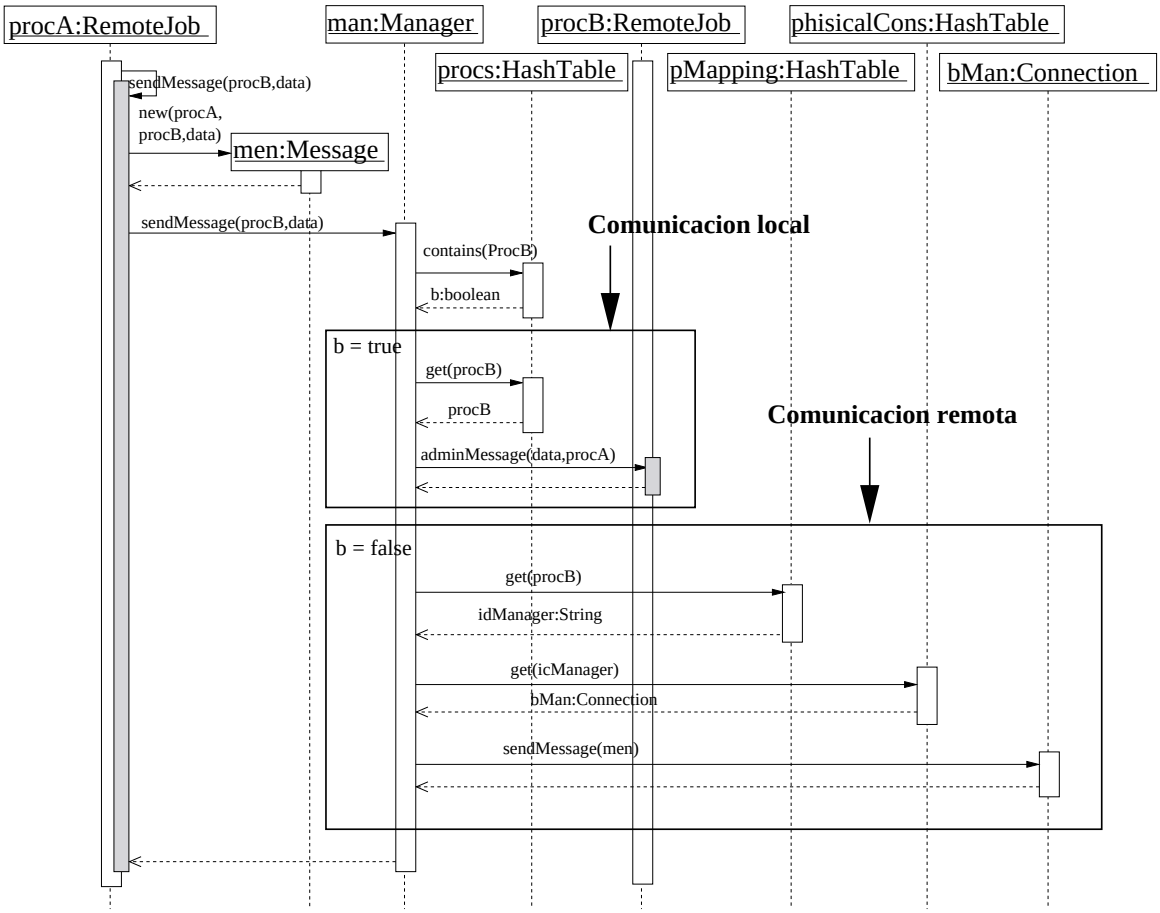

Figura 3.17: *Diagrama UML de secuencia de las instrucciones que se ejecutan en un Manager (man) cuando uno de sus procesos (procA) envía un dato a otro proceso (procB).*

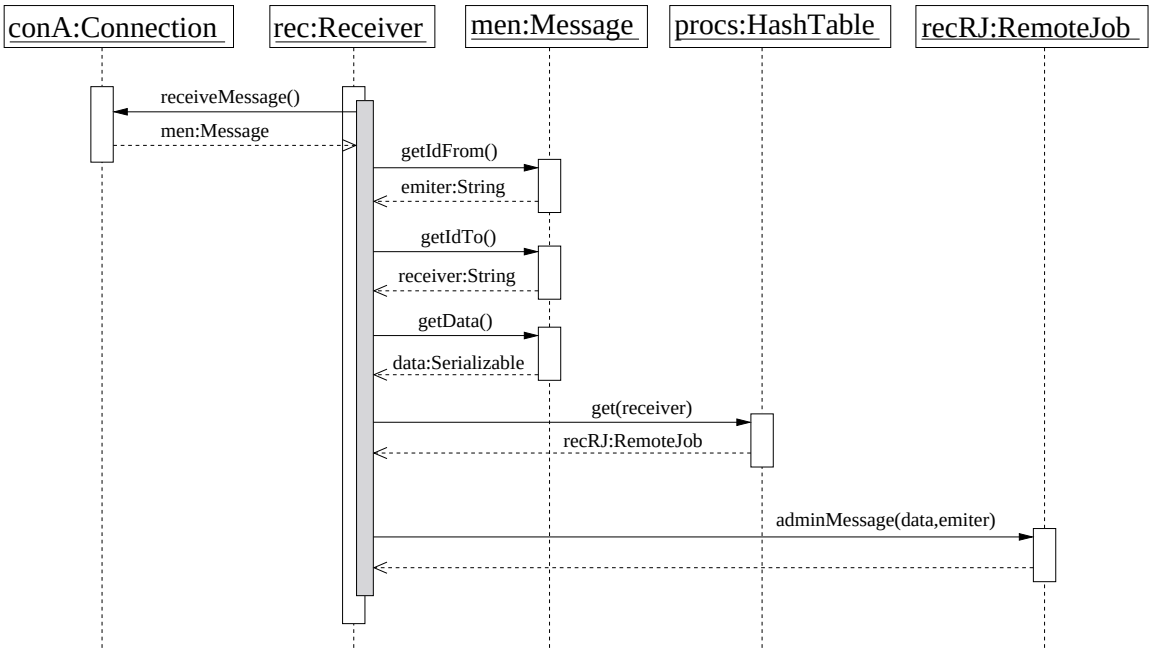

Figura 3.18: *Diagrama UML de secuencia de la recepción de mensajes en J-MIPS.*

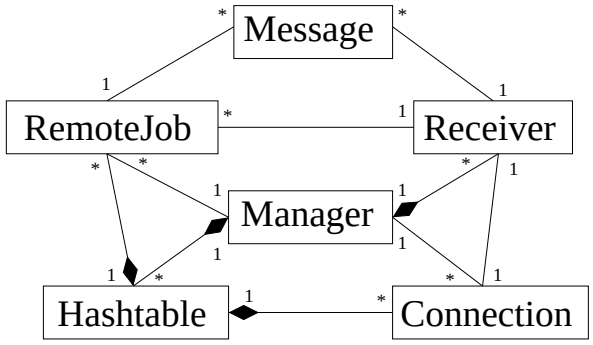

Figura 3.19: *Componentes involucrados en la comunicación entre procesos RemoteJob.*

Comunicación con la computadora virtual *Master*

La comunicación con el objeto *Master* se da a través de dos sub-procesos; uno para enviar mensajes y el otro para recibirlos.

Dado que varios procesos *RemoteJob* podrían querer enviar mensajes al objeto *Master*, primero deben escribirse en un buffer de tipo *SyncBuffer* de mensajes de salida de tipo *MasterMessage*. Un sub-procreso llamado *MasterSender* lee estos mensajes del buffer y los envía al objeto *Master* a través de la conexión *cMas* que es de tipo *Connection* (figura 3.20).

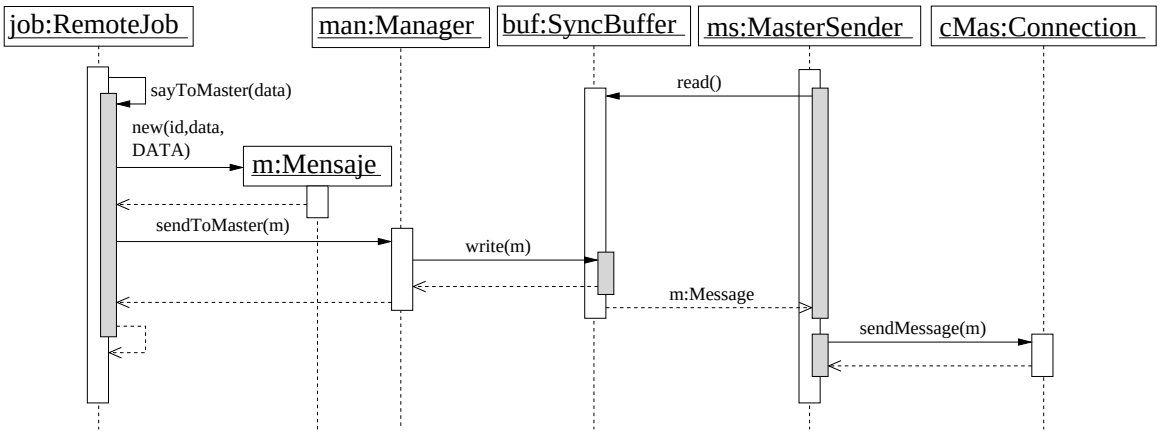

Figura 3.20: *Diagrama UML de secuencia cuando un proceso RemoteJob envía un dato al objeto Master.*

Para recibir mensajes, y dado que el método *receiveMessage()* del objeto *Connection* es síncrono, utilizamos un sub-proceso llamado *MasterReceiver*. Este sub-proceso utiliza la conexión establecida con el *Master* para recibir mensajes de tipo *MasterMessage*; una vez recibido el mensaje, lo pasa al decodificador de instrucciones del *Manager*, quien además, realiza la acción especificada en el mensaje (figura 3.21) mediante el método *processInstruction(MasterMessage m)*. Debemos recordar que la clase *MasterMessage* contiene el identificador del proceso que lo envía, los datos enviados y un indicador del tipo de mensaje o instrucción que contiene. Dependiendo del tipo del mensaje recibido, el método *processInstruction(MasterMessage m)* puede:

- Esperar conexiones de otros *Managers*.
- Realizar la conexión con otros *Managers*.

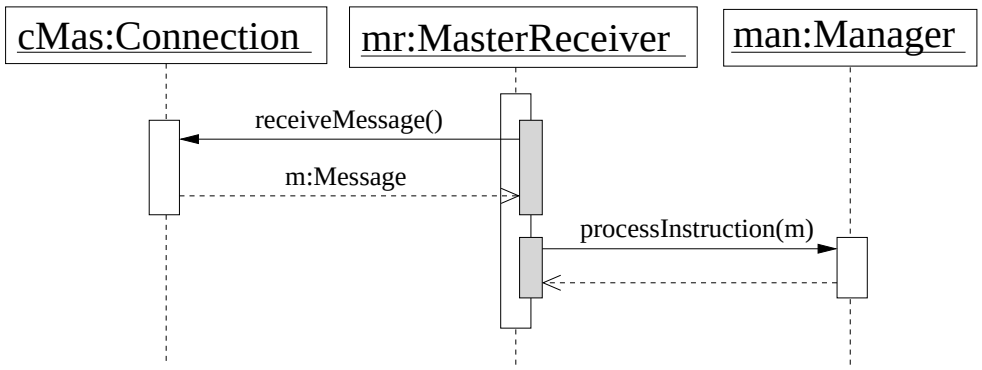

Figura 3.21: *Diagrama UML de secuencia cuando un Manager recibe una instrucción del objeto Master.*

- Iniciar la ejecución de los procesos *RemoteJob*.
- Detener la ejecución del *Manager*.

La figura 3.22 presenta el diagrama UML de las clases involucradas en la comunicación entre el objeto *Master* y los *Manager*.

Terminación

Una vez que los procesos *RemoteJob* han terminado su ejecución, el programador de aplicaciones paralelas debe avisar este hecho al *Manager* mediante la llamada *finished()* de dicho proceso. El *Manager* lleva la cuenta de cuántos procesos han terminado su ejecución. Cuando el número de procesos que han invocado al método *finished()* es igual al número de procesos que han sido ejecutados, el *Manager* manda un mensaje al objeto *Master* avisando que todos los procesos han terminado. El *Manager* queda a la espera de instrucciones del objeto *Master*.

Finalmente, la figura 3.23 presenta el diagrama de clase completo para los *Managers*.

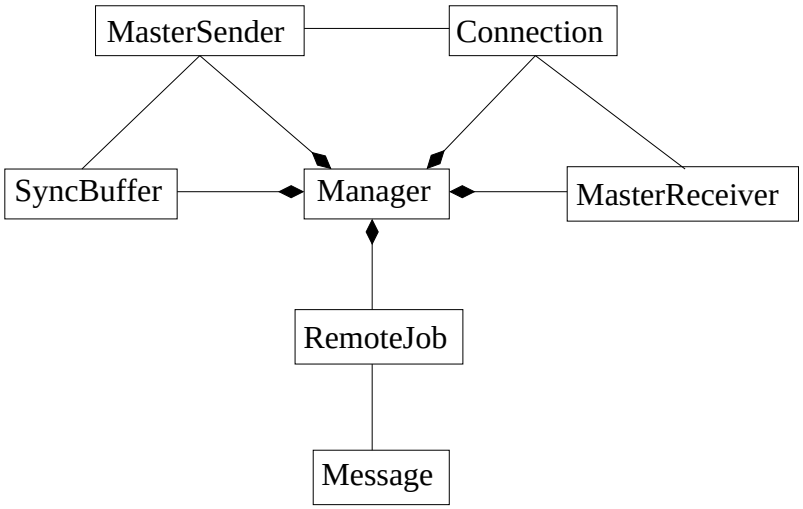

Figura 3.22: *Diagrama UML de clases de los componentes involucrados en el intercambio de información entre el objeto Master.* y los *Managers*

Manager

```
-localId: String
-masterPort: int
-masterCon: Connection
-phisicalCons: Hashtable<String,Connection>
-pMapping: Hashtable<String,String>
-procs: Hashtable<String,RemoteJob>
-buf: SyncBuffer<MasterMessage>
-mSender: MasterSender
-mReceiver: MasterReceiver
-nReceiver: NormalReceiver
```

```
+Manager(localId:String,masterPort:int)
+getId(): String
+connectToMaster()
-initConnections(cons:Hashtable<String,ConnectionData>)
-processInstruction(m:MasterMessage)
-startProcesses()
-startReceivers()
#sendMessage(id:String,m:Message)
-connectToAll()
-waitConsFor(id:String)
#jobFinished()
```

Figura 3.23: *Diagrama UML de la clase Manager.*

3.3.7. El objeto *Master*

Como mencionamos anteriormente, el objeto *Master* se encarga de administrar a los *Managers* y a los procesos que deben ejecutar. Este nodo es el único que tiene interacción directa con el usuario, pues:

- El usuario proporciona los procesos que debe ejecutar todo el sistema al objeto *Master*.
- Si hay un error en algún *Manager*, éste avisa al objeto *Master*, quien avisa al usuario.
- Si un proceso desea mandar un dato al usuario, primero debe mandarlo al objeto *Master*, quien lo comunica al usuario.

Puede pensarse en el objeto *Master* como en un pequeño sistema operativo, al cual se le provee con un conjunto de procesadores (Managers) y con una lista de procesos que debe ejecutar. Sin embargo, a diferencia de un sistema operativo convencional, el usuario puede definir qué procesadores ejecutan cuáles procesos y definir un conjunto de procesadores virtuales, entre otras cosas.

Representamos al objeto *Master* con una clase llamada *Master*. Esta clase contiene un método principal llamado *startSystem()*, el cual como su nombre indica, inicia la ejecución del sistema distribuido completo. La operación principal de este método es, a grandes rasgos, de la siguiente forma:

1. Obtiene la organización del sistema a partir del archivo de configuración. Esto quiere decir que el objeto *Master* obtiene del archivo:
    - Los *Managers* que tiene a su disposición.
    - El mapeo de los procesos a los *Managers*.
    - El mapeo de los *Managers* a las computadoras físicas.
    - La topología de la red virtual.
2. Establece comunicación con los *Managers*.
3. Envía a cada *Manager* la siguiente información:
    - El identificador del *Manager*.
    - El conjunto de procesos que debe ejecutar.
    - Las conexiones virtuales que debe establecer con otros *Managers*.

- La tabla Hash *pMapping* que contiene la ubicación de otros procesos.

4. Indica a cada *Manager* que inicie la ejecución de su conjunto de procesos.

Además, el método *startSystem()* pone en ejecución un Thread encargado de las siguientes tareas:

1. Supervisar cualquier mensaje que llegue desde un *Manager* y tomar una acción respecto a dicho mensaje [7].

2. Cuando todos los *Managers* han terminado su trabajo, indicar a cada uno que termine su ejecución [8].

3. Enviar al usuario los mensajes provenientes de los *Managers*, si es que éstos existen y si el usuario los pidió.

4. Terminar la ejecución del objeto *Master* y del sistema completo.

En las siguientes secciones profundizamos sobre cada uno de los pasos anteriores.

Calculando la red

Como especificamos en la sección pasada, la computadora virtual maestra debe saber exactamente cómo están organizados los procesos dentro de los *Managers*, así como las conexiones que deben establecerse entre estos últimos. Toda esta información es obtenida a partir del archivo de configuración y almacenada en el objeto *Master* en objetos temporales. Existen dos clases para almacenar estos datos temporalmente: *ManagerData* y *ProcData*.

ManagerData es la clase encargada de almacenar la información de cada *Manager* temporalmente en el objeto *Master*. La información almacenada sirve a esta última para establecer la comunicación con los *Managers*. Además, también es almacenada información útil a las computadoras virtuales esclavas para conectarse con otros *Managers* y para organizar la comunicación de los procesos.

Para cada *Manager* real, el objeto *ManagerData* contiene el identificador, dirección IP y puerto de enlace de dicho *Manager*. Además, debe almacenarse la información de las conexiones establecidas entre el *Manager* en cuestión y los otros *Managers*. Esto se lleva a cabo mediante la construcción de otra clase que

[7] Dependiendo del tipo de mensaje recibido, el objeto *Master* puede terminar la ejecución del sistema, pasar el mensaje al usuario o no hacer nada.

[8] Esto es, cada *Manager* termina su ejecución como proceso en la computadora física

llamamos *ConnectionData*, la cual, para cierta computadora conectada al *Manager*, contiene únicamente el identificador, dirección IP y puerto de enlace de dicha computadora. Así, *ManagerData* contiene una lista de objetos de tipo *ConnectionData* para simbolizar las conexiones entre el *Manager* que representa y los otros *Managers*.

El objeto *ManagerData* contiene también la tabla Hash *pMapping* donde se especifica la ubicación de los procesos que se ejecutan en otros *Managers*.

Finalmente, el objeto *ManagerData* contiene una lista con los identificadores de los procesos que deben ser ejecutados por el *Manager* que representa.

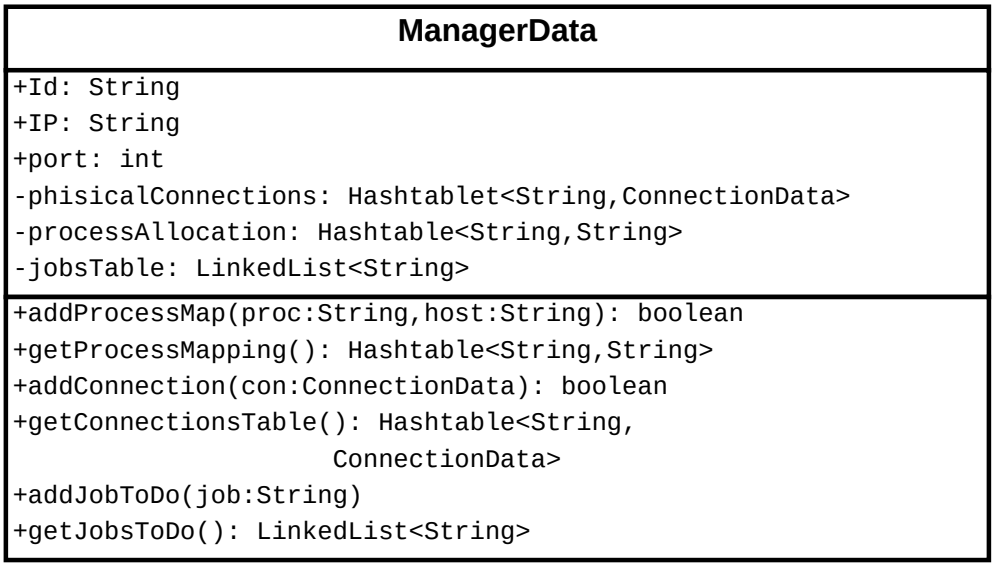

Figura 3.24: *Diagrama UML de la clase ManagerData.*

La figura 3.24 representa el diagrama de la clase *ManagerData*. Además de identificador, puerto y dirección IP, La clase *ManagerData* contiene las tablas Hash *physicalConnections* y *processAllocation* que representan las co-nexiones con otros *Managers* y el alojamiento de los procesos remotos receptores de mensajes respectivamente (la tabla *pMapping*). Elegimos utilizar tablas Hash para mantener el tiempo de respuesta más o menos constante al buscar la conexión de un *Manager* o dónde se encuentra un proceso. Además, el diagrama muestra el atributo *jobsTable* que representa la lista de procesos que debe ejecutar el *Manager*.

Las operaciones de la clase *ManagerData* agregan elementos a las colecciones mencionadas. Por ejemplo, el método *addProcessMap(String proc, String host)* agrega la pareja (proc,host) a la tabla Hash *processAllocation* para especificar que el proceso con identificador *proc* se encuentra ubicado en el *Manager* con identificador *host*. Las otras operaciones tienen funciones si-milares.

La clase *ProcData* contiene tres atributos: El identificador del proceso al que representa, el *Manager* en el cual reside (host) y una lista con los procesos con los cuales intercambia información. Esta es exactamente la misma información que la que se encuentra en las líneas *Process* del archivo de configuración.

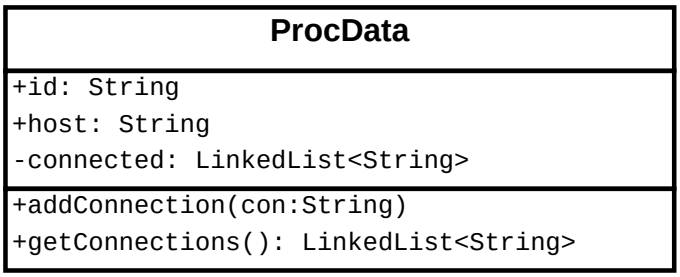

Figura 3.25: *Diagrama de la clase ProcData.*

En adelante, utilizamos la palabra *Conexión* en dos diferentes sentidos: Cuando hablamos de conexión entre dos computadoras virtuales, hablamos de un objeto *Connection* o del enlace que existe entre dichas computadoras. Cuando hablamos de conexión entre dos procesos, queremos decir que estos procesos intercambian información.

La figura 3.25 es el diagrama de la clase *ProcData*. La lista ligada *connected* simboliza las conexiones con otros procesos. El método *addConnection(String con)* agrega un proceso a la lista *connected*; el método *getConnections()* regresa la lista de todos los procesos conectados al proceso *id*.

En resumen, las clases *ManagerData* y *ProcData* contienen toda la información que el *Master* necesita para comunicarse con los *Managers* y darles instrucciones sobre la topología virtual de la red y sobre los procesos que deben ejecutar.

La clase *Master* contiene como atributos una tabla Hash de objetos de tipo *ManagerData* llamada *managerDataTable*, cuyas entradas son de la forma (Id,ManData), donde Id es el identificador del *Manager* que el valor *ManData* representa. Además, la clase *Master* también contiene una tabla Hash de objetos *ProcData* llamada *procDataTable*, con entradas de tipo (Id,PrData), donde Id es el identificador del

proceso representado por el valor PrData. Estas tablas deben ser pobladas al analizar el archivo de configuración con la información correcta sobre los *Managers* y los procesos que se ejecutan en ellos.

El análisis del archivo se realiza mediante un método de la clase *Master* llamado *processConfFile()*. Dado el carácter técnico del análisis del archivo, dejamos este tema para el capítulo 5.

Al terminar el análisis del archivo de configuración, el objeto *Master* tiene una idea clara de cómo debe funcionar el sistema. Esto quiere decir lo siguiente:

- El objeto *Master* sabe cuántos *Managers* tiene a su disposición y conoce la información necesaria para conectarse con ellos.
- Por cada *Manager*, el objeto *Master* tiene los siguientes datos:
  - Cuáles son los procesos que debe ejecutar el *Manager*.
  - En qué computadora física se aloja el *Manager*.
  - Con cuales otros *Managers* está conectado el *Manager* en cuestión.
  - La tabla *pMapping* del *Manager*.

  En otras palabras, el *Master* realiza un esquema local de la conformación de todo el sistema mediante el análisis del archivo de configuración. Este esquema permite al objeto *Master* poner en funcionamiento a los *Managers*.

Conexión con los *Managers*

Una vez analizado el archivo de configuración y pobladas las tablas *managerDataTable* y *procDataTable* con los datos correctos sobre la configuración del sistema, el siguiente paso del *Master* es conectarse a los *Managers*, que para ese momento, deben ya de esperar una petición de conexión por parte de ella.

Para conectarse con cada *Manager*, el objeto *Master* utiliza una clase que llamamos *ManagerConnection*. Esta clase básicamente encapsula un objeto de tipo *Connection* y tiene además un Thread llamado *Receiver* que se encarga de supervisar los mensajes que lleguen desde el *Manager* hasta el *Master*. Cuando un mensaje llega, el Thread *Receiver* se encarga de almacenarlo en único buffer de mensajes de llegada que el objeto *Master* supervisa. Tratamos este tema más adelante en esta misma sub-sección. Sin embargo, el Thread *Receiver* es importante para evitar bloqueos en la ejecución del objeto *Master* al intentar recibir mensajes

desde las conexiones con los *Managers*[9].

La clase *ManagerConnection* tiene entonces los siguientes atributos:

- Un identificador Id del *Manager* con el cual el *Master* está conectado.
- Una conexión *Connection* con el *Manager*.
- Un buffer *SyncBuffer* de mensajes de llegada desde los *Managers*.
- Un Thread *Receiver* para recibir los mensajes de llegada desde la conexión *Connection*.

Los métodos de la clase *ManagerConnection* se muestran en la figura 3.26 y refinan la operación del objeto *con* de la clase *ManagerConnection*. Por ejemplo, la clase *Connection* contiene un método llamado *sendMessage(Serializable data)* que acepta como argumento un objeto serializable. La clase *ManagerConnection* contiene al método *sendMessage(MasterMessage m)* que acepta únicamente mensajes de tipo *MasterMessage*. De igual forma, el método *establishConnection()* de la clase *ManagerConnection* verifica primero que no exista una conexión previa con dicho *Manager* antes de realizar la petición de conexión. Esto no lo hace el método del mismo nombre en la clase *Connection*.

Además, la clase *ManagerConnection* contiene un método llamado *startMasterReceiver()* que inicia la ejecución del Thread *Receiver*. La figura 3.26 representa el diagrama de la clase *ManagerConnection*.

El objeto *Master* contiene también como atributo: una tabla Hash llamada *managerTable*, donde almacena las conexiones establecidas con los *Managers*. Esta tabla tiene entradas de la forma (String,ManagerConnection), donde el código Hash es el identificador del *Manager* y el valor *ManagerConnection* es la conexión establecida con el mismo.

Así, para conectarse con los *Managers*, el objeto *Master* toma un objeto *ManagerData* de la tabla *managerDataTable* donde se encuentra la información de un *Manager*, con esta información crea un objeto *ManagerConnection* que representa la conexión lógica con el *Manager*, establece la conexión con este último invocando al método *establishConnection()* del *ManagerData* construido y por

[9]Recordemos que el método *receiveMessage()* bloquea la ejecución de quien lo haya invocado hasta que haya un mensaje que recibir.

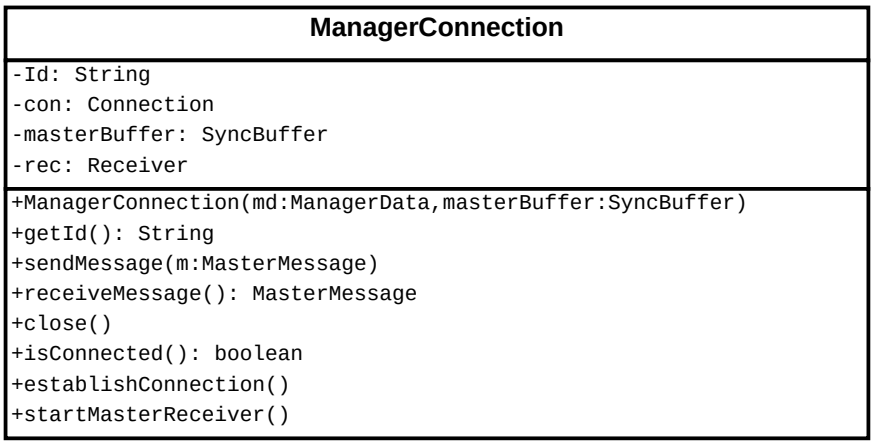

Figura 3.26: *Diagrama UML de la clase ManagerConnection.*

último almacena la conexión en la tabla *managerTable* para uso posterior. Después, el *Master* repite este procedimiento con el resto de objetos *ManagerData* de la tabla *managerDataTable*.

Asignando trabajo

En este punto, el objeto *Master* ha establecido conexión con todos los *Managers* quienes esperan más información sobre su operación. El objeto *Master* envía los siguientes datos a cada *Manager*:

- **El identificador del *Manager* en forma de una cadena de caracteres**.
- **Las conexiones que deben ser establecidas con otros *Managers*.** Esta información está contenida en los objetos *ManagerData* de la tabla Hash *ManagerDataTable*
- **La tabla *pMapping* del *Manager*.** Donde se encuentra la loca-lización de los procesos remotos al *Manager*.
- **El conjunto de procesos *RemoteJob* que el *Manager* debe ejecutar.** Este conjunto[10] se obtiene a partir de un método de la clase *Master* llamado *calcJobsFor(ManagerData md)* que utiliza la lista de identificadores de procesos *jobsTable* de la clase *ManagerData* y la tabla de procesos general *procsTable* de la clase *Master* y devuelve una tabla Hash de los procesos

[10]En realidad es una tabla Hash de procesos *RemoteJob*.

RemoteJob asignados al *Manager*. La figura 3.27 muestra el diagrama de secuencia del método *calcJobsFor(ManagerData md)*.

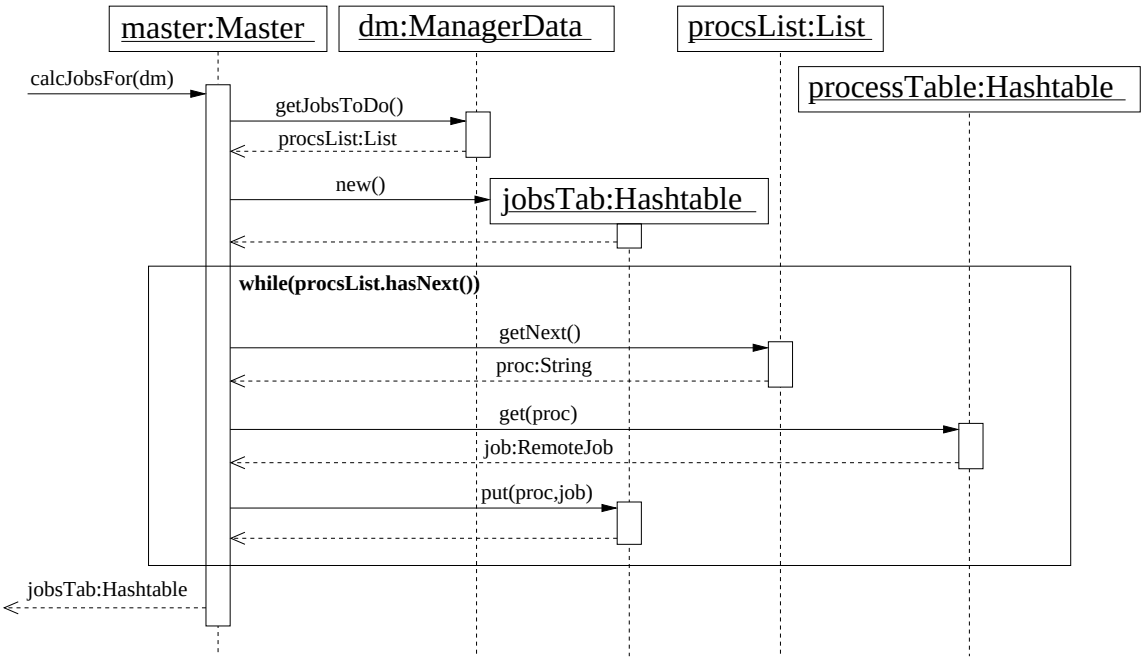

Figura 3.27: *Diagrama UML de secuencia del método calcJobsFor(ManagerData md), para obtener una tabla Hash con los procesos RemoteJob de cada Manager.*

Al terminar este procedimiento con todos los *Managers*, éstos quedan totalmente provistos con las herramientas necesarias para conectarse con otros *Managers* e iniciar la ejecución de sus procesos. Debemos hacer notar que no se envían a los *Managers* procesos en ejecución, sino las especificaciones de estos en forma de objetos de tipo *RemoteJob*.
La figura 3.28 muestra el diagrama UML de secuencia del método *StartSystem()* hasta donde lo hemos descrito, para un solo *Manager*.

Conexiones entre *Managers* e inicio de procesos

El siguiente paso es hacer que los *Managers* se conecten entre ellos para formar la topología virtual de la red. El objeto *Master* hace uso de la información almacenada en los *ManagerData*. Recordemos que cada uno de éstos, contiene una tabla Hash de objetos *ConnectionData*, los cuales representan las conexiones con otros *Managers*. Es decir, cada *ManagerData* representa a un *Manager* y contiene las conexiones que este debe establecer con otros *Managers*.

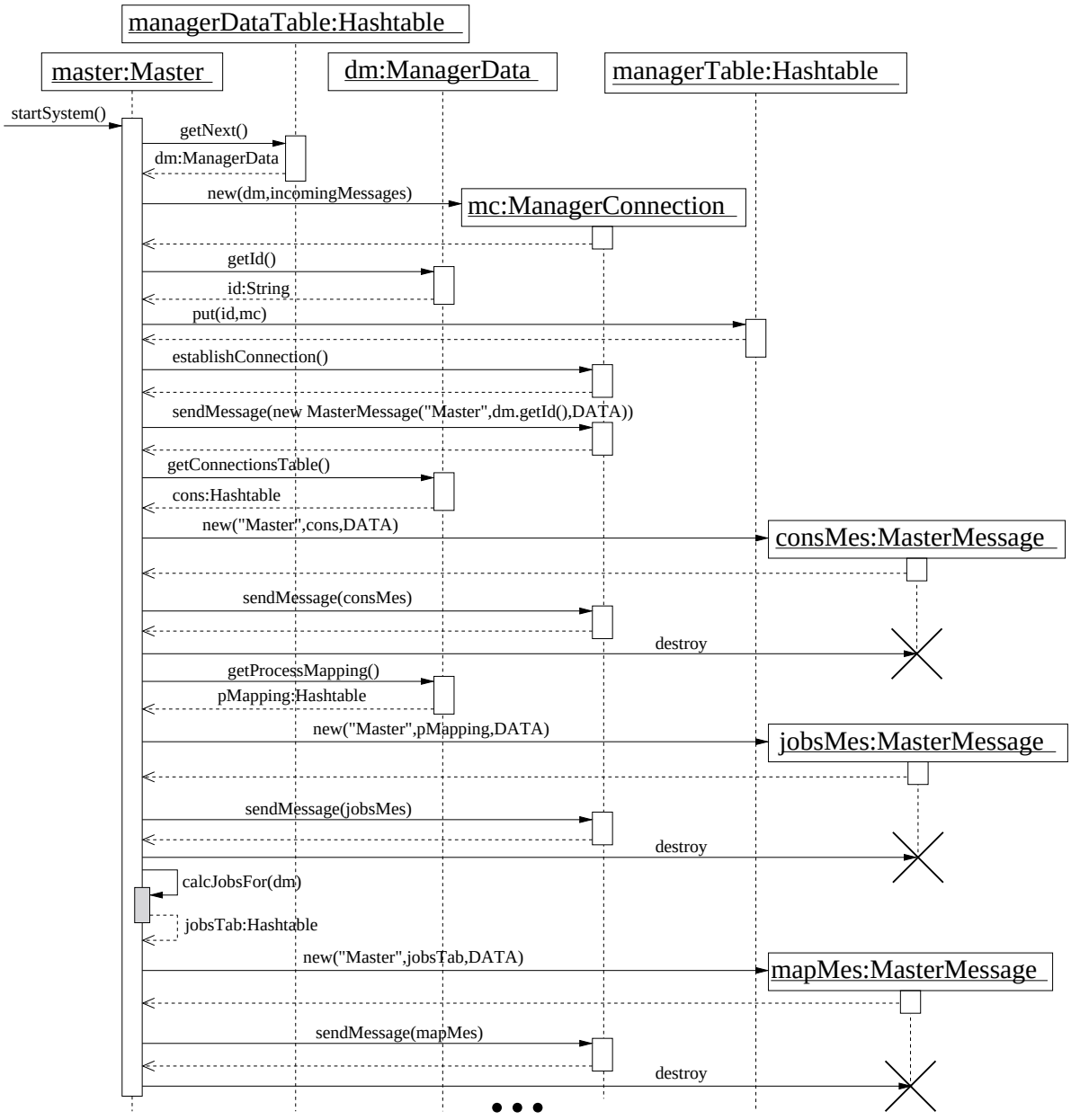

Figura 3.28: *Diagrama UML de secuencia de la primera parte del método startSystem(), para un solo Manager.*

Así, el objeto *Master* toma un objeto (*ManagerData*), que representa digamos al *Manager man1*, y mediante su conjunto de conexiones verifica a cuales otros *Managers* debe estar conectado. A estos últimos, el objeto *Master* envía un mensaje indicándoles que deben esperar una petición de conexión de *man1*. Finalmente, un mensaje es enviado al *Manager man1* indicándole que realice la petición de conexión a los otros *Managers*. Cuando *man1* termina de conectarse con todos los *Managers* especificados en su lista de conexiones, podemos asegurar que todos los procesos que se ejecuten en *man1*, podrán establecer comunicación con todos los otros procesos especificados en el archivo de configuración.

El procedimiento anterior es repetido para cada representante *ManagerData* de la tabla *ManagerDataTable* salvo el último, en un método llamado *communicate-Managers()*, construyendo así, la topología virtual entre computadoras virtuales. La figura 3.29 muestra el diagrama UML de secuencia del método *communicate-Managers()*.

Podemos aclarar lo que hemos mencionado con un ejemplo. Supongamos que existen tres *Managers*, m1, m2 y m3 conectados completamente entre si. Es decir, la red virtual es como en la figura 3.30. Después de analizar el archivo de configuración, el objeto *Master* almacena tres objetos *ManagerData*, que llamaremos md1, md2 y md3. Cada uno de estos, contiene una tabla con los identificadores de los otros dos *Managers* a los cuales está conectado.

El objeto *Master* centra su atención en md1. Sabemos que este es el representante del *Manager* m1 y especifica que este último está conectado a los *Managers* m2 y m3. El objeto *Master* envía un mensaje a m2 y m3 para avisarles que deben esperar a que m1 intente establecer conexión con ellos. Después, el *Master* envía un mensaje a m1 indicándole que realice peticiones de conexión con los *Managers* especificados en su lista de conexiones (m1 sabe que debe conectarse con m2 y m3, pues el objeto *Master* le dio esta información en pasos anteriores). Cuando m1 termina de establecer conexión con sus compañeros, envía un mensaje al objeto *Master* indicándole que ha terminado. El objeto *Master* repite el procedimiento con m2 y lo omite con m3, pues por la construcción del algoritmo de conexión entre *Managers*, el último de estos siempre queda completamente conectado con todos los *Managers* de su lista.

Una vez construida la red de computadoras virtuales, el objeto *Master* envía un mensaje a cada una de estas indicándoles que inicien la ejecución de sus procesos

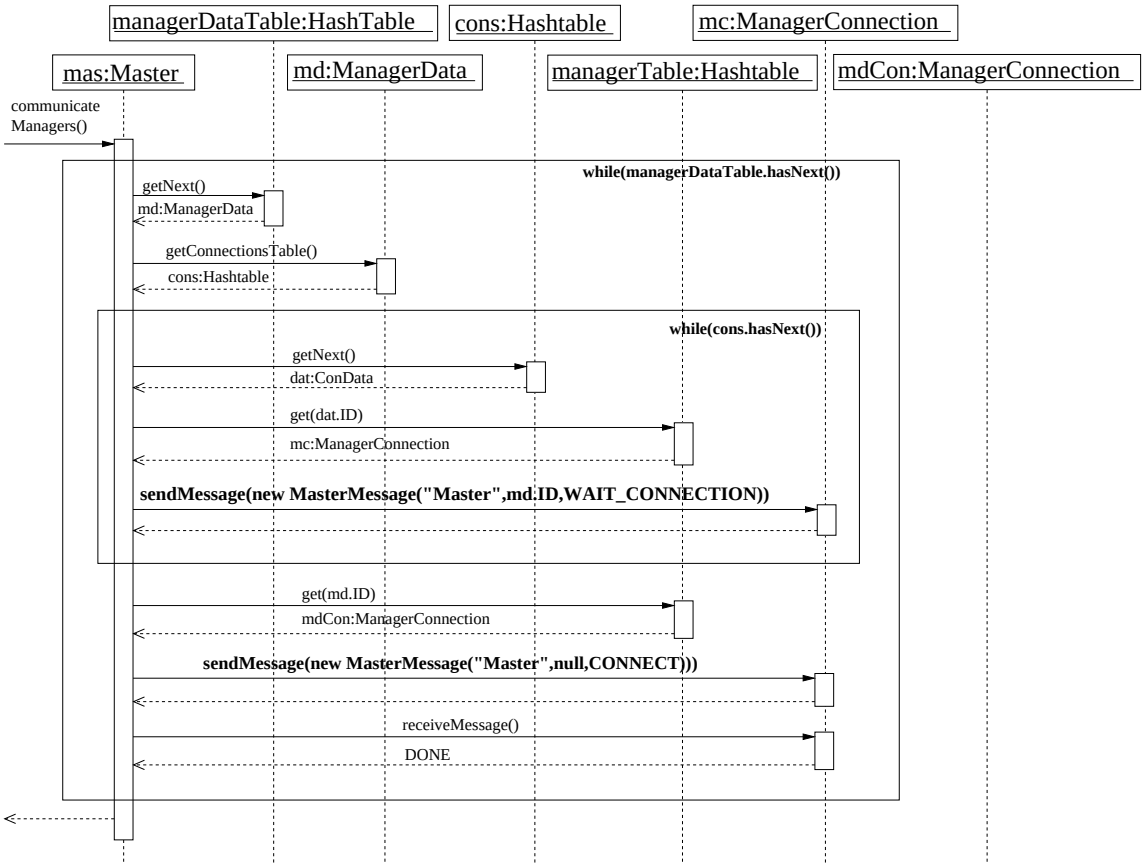

Figura 3.29: *Diagrama UML de secuencia del método communicateManagers() para establecer las conexiones entre computadoras virtuales.*

RemoteJob asignados.

Finalmente, son iniciados los Threads *Receiver* de las conexiones *ManagerConnection* para recibir mensajes desde los *Managers*.

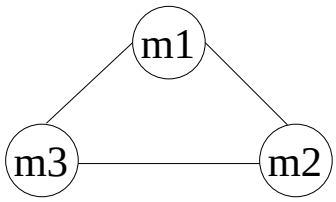

Figura 3.30: *Diagrama de la red de tres Managers m1, m2, m3 completamente conectada.*

Mientras los procesos trabajan

Una vez que los procesos han sido iniciados en cada *Manager*, el trabajo del objeto *Master* consiste en supervisar las conexiones con éstos en busca de mensajes que puedan ser generados por algún proceso *RemoteJob* o por los mismos *Managers*. Estos mensajes son almacenados en un buffer de mensajes en el objeto *Master* llamado *incomingMessages*. *Master* ejecuta un Thread llamado *MessageProcesor* encargado de leer mensajes del buffer y procesarlos según la naturaleza del mensaje. Para esto, el *Master* cuenta con un método llamado *processMessage()* que obtiene de cada mensaje el identificador del proceso emisor, el tipo del mensaje y otros posibles datos, con los cuales este método decide la acción que debe ser tomada respecto a ese mensaje. Esta acción puede ser una de las siguientes:

- Actualizar el estado de la ejecución del sistema. Esto es ¿Cuántos *Managers* han terminado la ejecución de sus procesos? En consecuencia, si todos han terminado, el sistema termina su ejecución.
- Pasar los datos al usuario, si esa era la intención del mensaje.
- Informar al sistema de un error provocado por alguno de los procesos o *Managers* y reaccionar ante él.

Podemos analizar el flujo de un mensaje emitido por algún *Manager*:

1. El *Manager* envía al *Master* un objeto *MasterMessage*
2. En el objeto *Master*, el Thread *Receiver* del objeto *ManagerConnection* correspondiente al *Manager* recibe el mensaje y lo almacena en el buffer *incomingMessages*.
3. El Thread *MessageProcesor* lee el mensaje del buffer e invoca al método *processMessage()* con el mensaje recibido

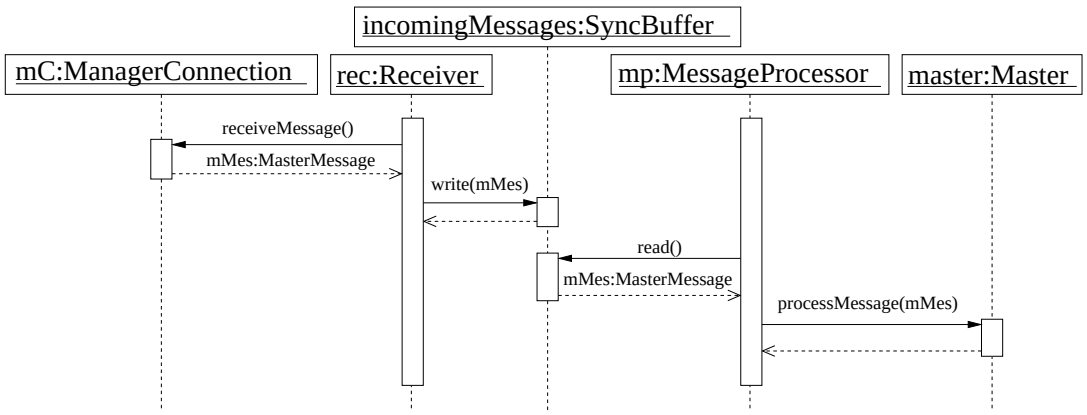

Figura 3.31: *Diagrama UML de secuencia del flujo de un mensaje desde la conexión con el Manager hasta su procesamiento en el objeto Master.*

4. El método *processMessage()* obtiene del mensaje el identificador del emisor, el tipo del mensaje y los posibles datos que se hayan enviado y decide qué debe hacer sobre ese mensaje.

La figura 3.31 muestra el diagrama UML de secuencia de las operaciones necesarias para procesar un mensaje *MasterMessage* que llega desde un *Manager*.

Interacción con el usuario

Como hemos mencionado, la interacción entre el sistema y el usuario (es decir el programador) se da a través del objeto *Master*. El programador puede comunicarse con ésta en los siguientes puntos:

- Antes de iniciar la ejecución del sistema para proveer al objeto *Master* con el archivo de configuración y con los procesos que deben ser ejecutados.
- Después de que todos los *Managers* han terminado su ejecución, para obtener mensajes que hayan llegado desde los procesos remotos.
- Para terminar la ejecución de todo el sistema.

Ya hemos hablado sobre el primero de los puntos anteriores. Acerca del segundo punto, el *Master* contiene una tabla Hash de buffers donde almacena los mensajes que lleguen desde los *Managers* y que vayan dirigidos al usuario. En esta tabla, existe un buffer de mensajes por cada proceso *RemoteJob* que se haya ejecutado.

Podemos recordar que cuando un mensaje llega al objeto *Master*, lo hace por

medio de un objeto *MasterMessage*. Este mensaje contiene el identificador del proceso que lo envía, el tipo del mensaje y un objeto *Data* donde se pueden enviar más datos. Si un *MasterMessage* llega al objeto *Master* y su tipo de mensaje indica que lo que se envían son datos para el usuario, el método procesador de mensajes *processMessage()* almacena el objeto *Data* en el buffer de mensajes del proceso que señala el identificador del mensaje.

Cuando todos los *Managers* terminen la ejecución de sus procesos, los buffers de mensajes para el usuario contendrán todos los datos que los procesos hayan querido enviar al usuario. En este momento, el programador puede obtener los datos que le hayan sido enviados mediante un método de la clase *Master* llamado *messageFrom(String id)*, donde como argumento se pasa el identificador del proceso del cual se quieren recuperar datos[11]. Además, la clase *Master* contiene un método llamado *isMessageFrom(String id)* que devuelve *true* si el buffer del identificador *id* tiene por lo menos un elemento.

Los métodos *isMessageFrom()* y *messageFrom()* permiten al usuario recuperar todos los datos que hayan enviado los procesos *RemoteJob* de forma ordenada.

Finalmente, una vez que los *Managers* han terminado la ejecución de sus procesos, quedan en estado de espera de instrucciones del objeto *Master*. Este, a su vez, sigue activo aun cuando los *Managers* han terminado y el usuario ha revisado los buffers de mensajes de los procesos. Para detener todo el sistema y salir de los múltiples programas que se ejecutan en las diversas computadoras *físicas*, el programador debe dar una instrucción al objeto *Master* por medio de un método llamado *shutdown()*. Este método, propaga un mensaje a través de todos los *Managers* indicándoles que terminen su ejecución como programas y después termina la ejecución del objeto *Master* y de todos los sub-procesos involucrados en el sistema.

Aspectos finales sobre el objeto *Master*

A pesar de todo el trabajo que realiza el objeto *Master*, tanto para organizar a los *Managers* y asignarles trabajo, como para supervisar su ejecución (pues existe un Thread por cada *Manager* para recibir mensajes desde éstos y un buffer de mensajes por cada proceso), los procesos *RemoteJob* se ejecutan con independencia total de estas tareas. El objeto *Master* se convierte en un mero observador cuando los procesos se ejecutan en los *Managers* y solo reacciona a los mensajes enviados por ellos. Al no intervenir el objeto *Master* en la ejecución del programa paralelo,

[11] En este sentido, el método *messageFrom()* no devuelve objetos de tipo *Message* o *MasterMessage*, sino los datos *Serializable* almacenados en éstos.

creemos que este esquema favorece la ejecución del resto del sistema distribuido. Los procesos pueden ejecutarse sin interrupciones o notificaciones del objeto *Master*, concentrándose plenamente en la labor especificada por el programador.

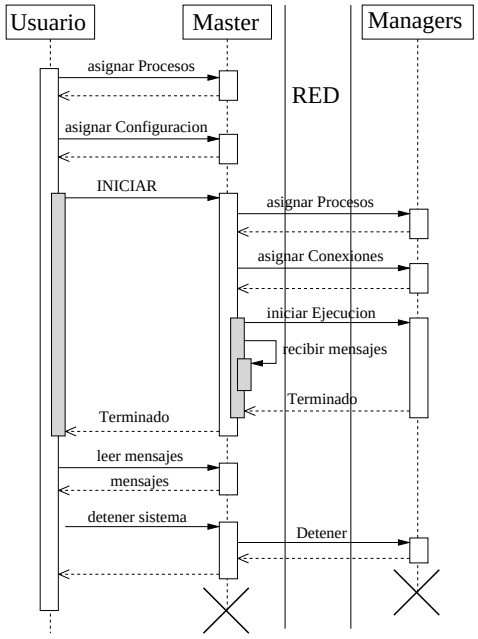

Figura 3.32: *Resumen de la operación del objeto Master.*

La figura 3.32 muestra un diagrama parecido a los diagramas de secuencia UML que muestra las tareas principales del objeto *Master* a lo largo de la ejecución del sistema paralelo. Puede verse que a pesar de mantener Threads internos para recibir y manejar mensajes, el *Master* no interrumpe la ejecución de los procesos en ningún momento.

La figura 3.33 muestra el diagrama UML del objeto *Master*. Esta clase en realidad posee más métodos que los que se muestran en el diagrama, pero su utilidad es realizar sub-tareas que realizan los métodos mostrados. Debido a ello, decidimos no incluirlos en el diagrama, en el que se muestran los métodos principales de la clase y de los cuales hemos hablado.

La figura 3.34 muestra el diagrama UML de clases de los componentes de la biblioteca que interactúan en la ejecución del objeto *Master*. Esto es, los componentes involucrados en el método *startSystem()*.

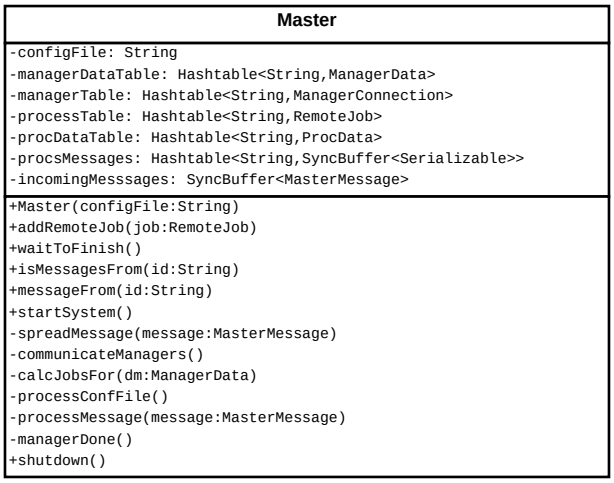

Figura 3.33: *Diagrama UML de la clase Master.*

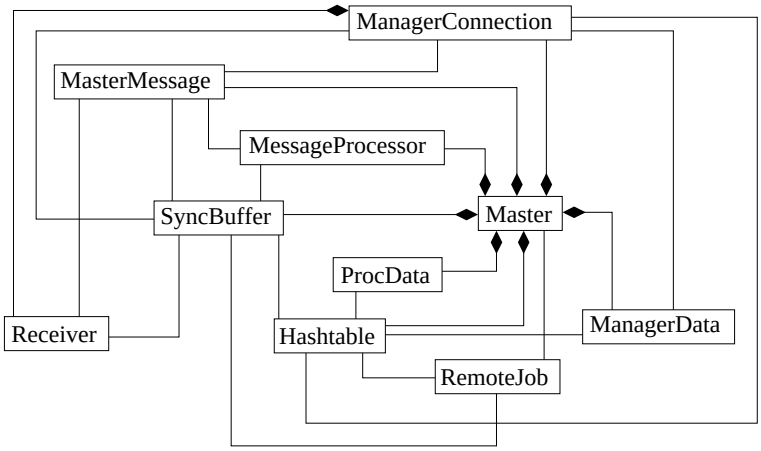

Figura 3.34: *Diagrama UML de clases de los componentes que interactúan en la operación del objeto Master.*

3.3.8. ¿Y la topología?

Al principio de este capítulo, hablamos sobre tres tipos de topología involucrados en la ejecución de aplicaciones paralelas por medio de J-MIPS:

- La topología de red de los procesos.
- La topología de la red de computadoras virtuales esclavas.
- La topología de la red física de computadoras.

La topología de procesos se mapea a la red de computadoras virtuales, que a su vez, es mapeada a la red física como en la figura 3.35.

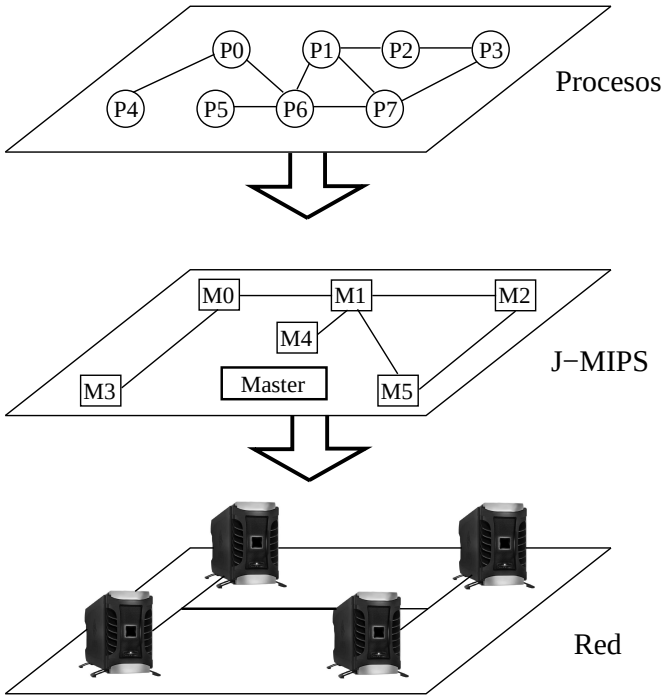

Figura 3.35: *Tres niveles de topología en J-MIPS. Los procesos P_i son mapeados a los Managers M_k, que a su vez son mapeados a la red física.*

El usuario debe especificar la topología de los procesos, el mapeo de estos en los *Managers* y el mapeo de los *Managers* en las computadoras físicas mediante el archivo de configuración. Al poder decidir donde se ejecutan los procesos, se obtienen diversas ventajas:

- Se pueden tomar en cuenta las limitaciones de la topología física de la red, pues aunque la forma más común de red de computadoras para el cómputo paralelo en memoria distribuida es Ethernet, existen otras topologías de red, donde es importante tomar en cuenta el flujo de la información entre computadoras; por ejemplo, en las topologías de anillo o estrella. Así, el usuario puede tomar en cuenta el flujo de información entre sus procesos, para disminuir la saturación en las comunicaciones entre computadoras.

- De manera similar, es posible tomar en cuenta la arquitectura de las computadoras de la red, pues esta no siempre es homogénea. En una red pueden existir computadoras con mejores recursos de hardware que otras. De igual forma, en una misma aplicación paralela, pueden existir procesos que requieran una mayor capacidad de hardware que otros. Así, tener control sobre el mapeo de los procesos permite al programador ajustar estos a las computadoras mejor equipadas para ejecutarlos.

- Cuando en la aplicación paralela los procesos tienen una fuerte dependencia entre ellos (como en un pipe-line), el mapeo de los procesos puede impactar en el tiempo de ejecución de la aplicación paralela. El usuario puede entonces definir un mapeo que facilite la comunicación entre sus procesos.

En J-MIPS, se considera que la topología de los procesos, los *Managers* y la red física es estática, lo cual quiere decir que no cambia durante la ejecución de las aplicaciones paralelas. Sin embargo, por cada ejecución, el usuario puede definir la topología de los procesos y de los *Managers*. Consideremos la siguiente línea de definición de procesos en un archivo de configuración hipotético:

$$\text{Process} = (\text{P2 , M1 , [P1,P4,P7] })$$

Esta linea define la siguiente información:

- El programador especifica que el proceso *P2* intercambia información con los procesos *P1,P4* y *P7*. Al hacer esto, el usuario establece todas las conexiones entre *P2* y el resto de los procesos (consideramos que si el identificador de un proceso no aparece en la lista de conexiones, la conexión con ese proceso no existe). Al especificar líneas de definición similares para cada proceso en el archivo de configuración, el usuario define exactamente la topología de la red de procesos.

- Además, el usuario mapea (asigna) el proceso *P2* al *Manager M1*. Es decir, el mapeo de los procesos en los *Managers* se da en cada linea *Process* cuando se designa en cuál *Manager* debe ejecutarse el proceso que se desea definir.

- Supóngase, que el proceso *P4* se define en el archivo de configuración mediante la siguiente linea:

  Process = (P4, M2, [P2])

  Es decir, el proceso *P4* se ejecuta en el *Manager M2* y está conectado únicamente con el proceso *P2*. Estas dos lineas *Process* (la de *P2* y la de *P4*) definen de forma indirecta una conexión entre los *Managers M1* y *M2*, pues para que *P2* y *P4* puedan intercambiar información, los *Managers* donde se ejecutan deben estar conectados. Así, mediante las mismas lineas *Process*, el usuario define también de forma indirecta la topología de los *Managers*.

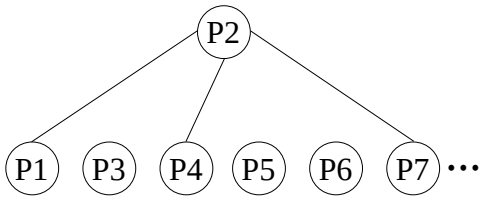

Figura 3.36: *El proceso P2 está conectado con los procesos P1,P4 y P7.*

La topología completa de los *Managers* es deducida por el objeto *Master* a partir del archivo de configuración utilizando el siguiente principio de conexión:

> Sean p y q procesos de tipo *RemoteJob*; y sean M y N *Managers* tales que p se ejecuta en M y q en N. Si p y q intercambian información por medio de paso de mensajes, entonces **existe una conexión virtual entre *M* y *N*, la tupla *(q,N)* aparece en la tabla *pMapping* de *M* y la tupla *(p,M)* aparece en la tabla *pMapping* de *N*.**

Es decir, si dos procesos se ejecutan en *Managers* distintos e intercambian mensajes, entonces los *Managers* deben estar conectados. Además, cada *Manager* debe saber en donde se ejecuta el proceso remoto con el cual el proceso local mantiene comunicación (la tabla *pMapping*).

Si se evalúan todas las líneas *Process* del archivo de configuración, analizando cada identificador de proceso, el *Manager* donde se ejecuta y su lista de conexiones, siguiendo el principio de conexión mencionado, entonces se puede calcular la topología de las computadoras virtuales esclavas.

Por supuesto, si cambian las asignaciones (mapeos) de los procesos a los *Managers*, lo más probable es que también cambie la topología de las computadoras virtuales esclavas, independientemente de la topología de los procesos.

3.3.9. Excepciones

Desafortunadamente, muchos errores pueden ocurrir durante la ejecución de un programa paralelo con J-MIPS, los cuales pueden alterar seriamente el comportamiento de todo el sistema. Por ejemplo, el programador puede especificar un archivo de configuración inexistente. A estas situaciones excepcionales las llamamos precisamente excepciones. Para manejar estos errores, nos guiamos por los siguientes principios:

- Siempre que ocurra un error, J-MIPS debe informar al usuario cual fue el error y en dónde se generó.
- Siempre que ocurra un error, J-MIPS debe detener la ejecución de todo el sistema

Con todo el sistema nos referimos al objeto *Master* y todos los *Managers*.

En J-MIPS consideramos tres tipos de errores que son tratados de diferentes maneras:

- **Excepciones del *Master*.** Estas excepciones alteran únicamente el funcionamiento del objeto *Master* y ocurren antes de que este establezca comunicación con los *Managers*. Por ejemplo, cuando el programador proporciona un nombre incorrecto para el archivo de configuración, o cuando existen errores en éste. Al detectar este tipo de excepciones, el objeto *Master* simplemente detiene su ejecución e informa al usuario mediante la consola de comandos sobre el error ocurrido. Los *Managers* quedan activos y a la espera; exactamente en el mismo estado en el que se encontraban antes de que ocurriera el error.

  En esta categoría, manejamos tres excepciones:

  - *ConFileException.* Que es lanzada cuando existe un error en el archivo de configuración.
  - *DuplicatedElementException.* Que se lanza cuando el programador intenta agregar en el objeto *Master* un proceso cuyo identificador ya existe.

- *IDNotFoundException.* Que se lanza cuando el programador intenta obtener un mensaje desde un proceso que no se encuentra en la tabla de procesos.

- **Excepciones en la comunicación.** Este tipo de errores pueden darse al establecer comunicación entre dos computadoras virtuales, o al enviar o recibir mensajes. En general, estos errores son bastante catastróficos, pues provocan una falla en todo el sistema.

  La excepción *ConnectionException* es utilizada para todos los errores de este tipo.

- **Excepciones generadas por los procesos *RemoteJob*.** Es decir, errores generados por el código escrito por el programador y que no tienen que ver con J-MIPS. Por ejemplo, una división entre cero o un error de conversión de tipos. El manejo de este tipo de excepciones se delega al programador, el cual debe asegurarse de que los errores que se produzcan en los procesos siempre sean controlados.

  No manejamos ninguna excepción para este tipo de error. Esta tarea es delegada al programador. Si existe una excepción y esta no es atrapada por el programador, es posible que el proceso nunca anuncie al *Manager* en el que reside que su trabajo ha terminado, por lo cual el *Manager* se quedará esperando este aviso indefinidamente.

Respecto a las **Excepciones en la comunicación**, éstas pueden darse en dos situaciones:

- En la comunicación entre *Managers*.
- En la comunicación entre los *Managers* y el *Master*.

Cuando ocurre un error en la comunicación entre *Managers*, éstos deben comunicarse con el objeto *Master* informándole el error y esperar instrucciones por parte de ésta. El objeto *Master* recibe el error e informa a TODOS los *Managers* que deben detener su ejecución, informa al usuario de qué tipo es el error y en donde se generó y termina su ejecución. La intervención del objeto *Master* es necesaria para que todos los *Managers* sepan que hubo un error en algún punto de la red y detengan su ejecución.

Cuando existe un error en la comunicación entre algún *Manager* (o todos) y el *Master*, ésta debe informar a todos los *Managers* que pueda que se detengan,

informar el error al usuario y después detenerse ella misma. Los *Managers* que detecten errores en la comunicación con el objeto *Master* deben simplemente terminar su ejecución.

Los errores pueden ser enviados a través de mensajes *MasterMessage*. El objeto *Master* puede entonces obtener el error y el identificador de la computadora virtual en donde se generó e informar al usuario.

Este esquema permite que todas las computadoras virtuales, incluido el *Master*, se detengan automáticamente cuando se encuentra un error en las comunicaciones.

A primera vista, el detener todo el sistema frente a una excepción de este tipo parece ser una medida muy drástica; sin embargo, si la red física es estable, este tipo de excepciones rara vez ocurrirán. Además, al no corregir estas excepciones con tiempos de espera, reconexiones o retransmisiones, se evitan demoras que de otra forma retrasarían la ejecución de la aplicación paralela y serian transparentes al usuario.

3.4. Resumen del capítulo

En este capítulo se revisa a fondo el diseño de la biblioteca J-MIPS. Se inicia introduciendo al lector en los aspectos que son considerados para diseñar la biblioteca.También se proporciona una idea general de su funcionamiento y los componentes que la integran para poder cumplir nuestro objetivo: mapear e interconectar procesos de software. Estos componentes son los procesos *RemoteJob*, el objeto *Master* y las computadoras virtuales *Manager* que ejecutan los procesos. Estos tres componentes permiten al programador implementar aplicaciones de ejecución paralela y distribuida en un cluster de computadoras, configurando la ejecución por medio de un archivo de texto.

Se continua el capítulo detallando el diseño de los componentes *RemoteJob*, *Manager* y *Master* así como las interacciones y el flujo de datos entre ellos.

Se sigue el capítulo tratando los tres niveles de topología existentes en la ejecución de un programa paralelo (topología de procesos, de *Managers* y de computadoras físicas), así como los grados de dependencia entre ellos.

Finalmente, se trata el manejo de las posibles excepciones que pueden surgir al ejecutar un programa de ejecución paralela mediante J-MIPS.

Capítulo 4

Biblioteca J-MIPS. Implementación en Java

En este capítulo revisamos la implementación de la biblioteca J-MIPS en el lenguaje de programación Java. Utilizamos para ello, la versión 1.5 de java. Debemos recordar al lector que hay ciertas características de la implementación que impiden su ejecución en versiones anteriores a Java 1.5.

Debido a la extensión de la biblioteca (alrededor de 2000 líneas de instrucciones), no incluimos todo el código fuente, sino solo los bloques mas importantes para esclarecer el funcionamiento de J-MIPS. El lector puede encontrar el código completo en el disco anexo a esta Tesis.

Iniciamos directamente describiendo el código de los componentes que consideramos más importantes en la biblioteca que desarrollamos. Estos componentes son: *SyncBuffer*, *Message*, *MasterMessage*, *RemoteJob*, *Connection*, *Manager* y *Master*. El estudio de la implementación de estos componentes incluye a otros de menor relevancia pero necesarios para el correcto funcionamiento e interacción entre los componentes principales. Todos los componentes desarrollados en este capítulo están integrados en un solo paquete que llamamos *jmips*.

El capítulo continua describiendo las instrucciones que el programador de aplicaciones paralelas debe usar para poner en marcha un programa usando J-MIPS, mediante un ejemplo sencillo.

Finalmente, se abordan ciertos aspectos que el programador debe tener en cuenta al programar aplicaciones paralelas, para obtener el máximo provecho de J-MIPS. Esto quiere decir disminuir el tiempo de ejecución y evitar errores en tiempo de ejecución.

4.1. *SyncBuffer*: el buffer sincronizado

En la sección 3.3.1 hablamos de un componente muy importante para la comunicación entre procesos; el buffer. Para representar un buffer, utilizamos la clase *SyncBuffer*. Esta clase es genérica para que se pueda definir el tipo de datos que el buffer debe almacenar. Por ahora, consideramos que cualquier objeto puede ser almacenado en un buffer.

Para poder representar a la cola de prioridades de los datos que el buffer almacena, utilizamos una lista ligada genérica; una *LinkedList*<*T*>. Cuando el constructor de la clase *SyncBuffer* es invocado, éste debe inicializar la lista ligada y fijar el tamaño máximo del buffer.

Así, la definición de la clase *Syncbuffer* y su constructor quedan como en el código 4.1.

Código 4.1: Definición de la clase *SyncBuffer* con su constructor

```
1  public class SyncBuffer<T> {
2
3      private LinkedList<T> buffer;
4      private int maxSize;
5
6      public SyncBuffer(int size){
7          this.buffer = new LinkedList<T>();
8          this.maxSize = size;
9      }
10     ...
11 }
```

En la sección 3.3.1 mencionamos la existencia de tres métodos para este buffer: *read, write* y *size*. Mencionamos que estos tres métodos deben ser atómicos para evitar corrupción de datos cuando existe una condición de competencia por el uso del buffer. En Java, podemos simular este comportamiento por medio de la palabra reservada *synchronized*.

Mencionamos también el comportamiento del buffer cuando está lleno o vacío:

- Cuando el buffer está lleno, los procesos que quieran escribir datos en el buffer deben escribir a que otro proceso lea un dato del buffer.
- Análogamente, cuando el buffer está vacío, los procesos que quieran leer datos del buffer deben esperar a que otro proceso escriba un dato en el buffer.

Este comportamiento puede alcanzarse mediante el uso de los métodos *wait()* y *notifyall()* de la superclase *Object*. Los método *write()* y *read()* quedan como en el código 4.2

Código 4.2: Métodos *write()* y *read()* de la clase SyncBuffer

```
1  public synchronized void write(T data){
2      while(buffer.size() >= maxSize){
3          try{
4              wait();
5          }
6          catch(InterruptedException e){
7              e.printStackTrace();
8              return;
9          }
10     }
11     buffer.add(incoming);
12     notifyAll();
13 }
14
15 public synchronized T read(){
16     while(buffer.size() == 0){
17         try{
18             wait();
19         }
20         catch(InterruptedException e){
21             e.printStackTrace();
22             return null;
23         }
24     }
25     T res = buffer.remove();
26     notifyAll();
27     return res;
28 }
```

El acceso a los métodos "synchronized" de una clase está restringido a un solo proceso (Thread). Cuando un Thread invoca un método "synchronized" de una clase y un segundo proceso desea invocar algún método "synchronized" de la misma clase, el segundo debe esperar hasta que el primero termine. De esta forma, haciendo "synchronized" los métodos *read(), write()* y *size()*, aseguramos que no hay corrupción de datos por interrupciones a la ejecución de estos métodos.

Al ser la clase *SyncBuffer* genérica de tipo T, el método *write()* acepta como argumento un objeto de tipo T para escribir en la lista ligada *buffer*. De igual forma, el método *read()* devuelve un objeto de tipo T.

Los métodos *wait()* y *notifyAll()* de la superclase *Object*, bloquean o desbloquean

a los procesos que hayan invocado al método que los contiene. Cuando un proceso invoca un método que ejecuta la sentencia *wait()*, El proceso queda bloqueado hasta que otro proceso ejecute un método que contenga la instrucción *notifyAll()*.

En el método *write()*, cuando el buffer está lleno (*buffer.size() >= maxSize*), el método *wait()* bloquea la ejecución del proceso que manda llamar al método *write()*. Cuando algún otro proceso ejecute el método *read()*, éste ejecuta también la sentencia *notifyAll()* que "despierta" al proceso que ejecuta el método*write()*. En cierto sentido, cuando un proceso ejecuta el método *read()* del *SyncBuffer*, **avisa** a los otros procesos que el buffer no está lleno y pueden escribir en él. De la misma forma, cuando el método *write()* agrega un elemento al buffer, se ejecuta la sentencia *notifyAll()* que **avisa** a los otros procesos que el buffer no esta vacío y pueden leer datos de él.

El método *size()* devuelve el número de elementos de la lista ligada *buffer*. Finalmente, la clase *SyncBuffer* queda definida por el código 4.3

Código 4.3: Clase *SyncBuffer*

```
1  public class SyncBuffer<T> {
2
3      private LinkedList<T> buffer;
4      private int maxSize;
5
6      public SyncBuffer(int size){
7          this.buffer = new LinkedList<T>();
8          this.maxSize = size;
9      }
10
11     public synchronized void write(T incoming){
12         while(buffer.size() >= maxSize){
13             try{
14                 wait();
15             }
16             catch(InterruptedException e){
17                 e.printStackTrace();
18                 return;
19             }
20         }
21         buffer.add(incoming);
22         notifyAll();
23     }
24
25     public synchronized T read(){
26         while(buffer.size() == 0){
27             try{
28                 wait();
```

```
29              }
30              catch(InterruptedException e){
31                  e.printStackTrace();
32                  return null;
33              }
34          }
35          T res = buffer.remove();
36          notifyAll();
37          return res;
38      }
39
40      public synchronized void clear(){
41          buffer.clear();
42      }
43
44      public synchronized boolean contains(T element){
45          return buffer.contains(element);
46      }
47
48      public synchronized int size(){
49          return buffer.size();
50      }
51  }
```

4.2. Mensajes y tipos de mensajes

En la sección 3.3.2, hablamos sobre los mensajes que serán intercambiados entre las computadoras virtuales de la biblioteca J-MIPS. Estos pueden ser de dos tipos:

- Mensajes maestros. Que son intercambiados entre el objeto *Master* y los *Managers*
- Mensajes regulares. Que son intercambiados entre los *Managers*.

Los mensajes maestros están representados por la clase *MasterMessage* y son envoltorios de datos e instrucciones que deben ser interpretados por la computadora de destino. Mencionamos en la sección 3.3.2 los atributos de esta clase:

- El identificador del emisor del mensaje
- El dato que debe ser enviado en el mensaje
- La instrucción que se debe realizar sobre ese mensaje.

La clase *MasterMessage* se muestra en el código 4.4.

Código 4.4: Código de la clase *MasterMessage*

```
1  public class MasterMessage implements Serializable{
2
3      private String idProcess;
4      private Serializable data;
5      private int type;
6
7      public MasterMessage(String idProcess, Serializable data, int
           type){
8          this.idProcess = idProcess;
9          this.data = data;
10         this.type = type;
11     }
12
13     public String fromProcess(){
14         return this.idProcess;
15     }
16
17     public Serializable getData(){
18         return this.data;
19     }
20
21     public int getMessageType(){
22         return this.type;
23     }
24 }
```

Para que un objeto pueda ser enviado a través de la red en Java, primero debe ser convertido en un flujo de bytes. Esto se logra mediante la interfaz *Serializable* del paquete java.io. Así, la clase *MasterMessage* implementa a esta interfaz y el dato que debe ser enviado también debe ser de tipo *Serializable*. El identificador es una cadena de caracteres *String* que es *Serializable* y la instrucción está dada por un número entero *int* que también es *Serializable*.

El constructor debe conocer los tres atributos del mensaje al momento de su llamada. Por eso, al construir un *MasterMessage*, estos atributos deben pasarse como argumentos al constructor.

No es posible modificar un atributo una vez construido el mensaje; sin embargo, es posible obtener el valor de sus atributos mediante métodos **get()**.

Los mensajes regulares están definidos por medio de la clase *Message* cuyo código se muestra en el código 4.5. Son extremadamente parecidos a los *MasterMessage*, excepto que en vez de una instrucción de tipo *int*, llevan el identificador de tipo *String* del proceso al cual van dirigidos y un método de acceso **get()** a dicho

identificador. En resumen, la clase *Message* tiene los siguientes atributos:

- El identificador *idFrom* del emisor
- El identificador *idTo* del receptor
- El dato *data* que se quiere enviar.

Igual que en la clase *MasterMessage*, los mensajes *Message* se construyen inicializando todos sus atributos, los cuales no son modificables, pero existen métodos *get()* de acceso a ellos.

Código 4.5: Código de la clase *Message*

```
1  public class Message implements Serializable{
2
3      private Serializable data;
4      private String idFrom;
5      private String idTo;
6
7      public Message(String idFrom, String idTo, Serializable data)
           {
8          this.idFrom = idFrom;
9          this.idTo = idTo;
10         this.data = data;
11     }
12
13     public String getIdFrom(){
14         return this.idFrom;
15     }
16
17     public String getIdTo(){
18         return this.idTo;
19     }
20
21     public Serializable getData(){
22         return this.data;
23     }
24 }
```

4.3. RemoteJob: la clase abstracta

La clase *RemoteJob* representa a los procesos en J-MIPS que el usuario desea que se ejecuten en el sistema distribuido. Según la sección 3.3.4, un objeto de tipo *RemoteJob* debe cumplir las siguientes características:

- Debe poderse transmitir entre computadoras físicas, pues el objeto *Master* debe poder enviar los procesos a los *Managers*.
- El usuario debe poder especificar el comportamiento del *RemoteJob*; es decir, las instrucciones que deben ser ejecutadas.
- A pesar del punto anterior, todos los procesos *RemoteJob* deben realizar algunas operaciones como enviar mensajes mediante los *Managers* o administrar los mensajes que lleguen dirigidos a ellos (el método *adminMessage()*).

Para poder cumplir los requerimientos anteriores, hacemos uso de las clases abstractas y las interfaces *Runnable* y *Serializable* para definir la clase *RemoteJob*:

```
public abstract class RemoteJob  implements Serializable,Runnable{
...
}
```

Al implementar a la interfaz *Serializable*, los objetos *RemoteJob* pueden ser convertidos en flujos de bytes y enviados a través de la red. La implementación a la interfaz *Runnable* obliga a todos los objetos *RemoteJob* a implementar el método:

```
  public void run()
```

El método anterior define el comportamiento del *RemoteJob*, es decir, dentro del método *run()*, están las instrucciones que el programador desea que se ejecuten.

Para poder permitir que el programador defina el comportamiento del método *run()*, hacemos de la clase *RemoteJob* una clase abstracta. De esta forma, el programador puede crear sub-clases de *RemoteJob* donde es obligado a implementar dicho método. Los objetos *RemoteJob*, que recibe el objeto *Master*, deben ser entonces instancias de sub-clases que heredan de *RemoteJob* y que por tanto implementan al método *run()*.

Un *RemoteJob* definido por el usuario debe verse como el siguiente bloque de código:

```
public class MyRemoteJob extends RemoteJob{
  ...
  public void run(){
    ...
  }
}
```

La sección 4.8 detalla la implementación de sub-clases de *RemoteJob*.

Los atributos de la clase *RemoteJob* son los siguientes:

```
1    protected String id;        //El identificador de este proceso
2    private Manager man;       //El Manager que ejecuta este proceso
3    private LinkedList<String> connectedProcesses;         //Lista
        de procesos conectados a este proceso
4    private Hashtable<String,SyncBuffer<Serializable>>
        incomingMessages;    //Mensajes entrantes
```

La lista de procesos conectados *connecterProcesses* sirve para poblar la tabla *incomingMessages* con los buffers necesarios para recibir mensajes desde otros procesos. Esta lista es fijada por el objeto *Master* antes de enviar el proceso al *Manager* que le corresponde. Dado que el *RemoteJob* debe tener comunicación con el *Manager* en el que se ejecuta al enviar mensajes, el proceso debe tener acceso a algunos métodos del *Manager*, por lo cual uno de los atributos de la clase *RemoteJob* es el *Manager* en el que se ejecuta. Este atributo es fijado por el propio *Manager* cuando este recibe al proceso desde el objeto *Master*.

Al construir una instancia de una sub-clase de *RemoteJob*, el único atributo que debe y puede conocerse es el identificador del *RemoteJob*. El resto de los atributos deben permanecer nulos hasta que el objeto *Master* y el *Manager* en donde reside el *RemoteJob* cambien su valor. El constructor de la clase *RemoteJob* es como sigue:

```
1   public RemoteJob(String id){
2          this.id = id;
3          man = null;
4          connectedProcesses = null;
5          incomingMessages = null;
6   }
```

La clase *RemoteJob* tiene un método no tratado en el capítulo 4 llamado *initializeBuffers()* que es invocado por el *Manager* donde el objeto *RemoteJob* reside. Este método inicializa los buffers de la tabla *incomingMessages* para poder recibir mensajes desde otros procesos. El método *initializeBuffers()* es como sigue:

```
1   protected void initializeBuffers(){
2      incomingMessages = new Hashtable<String, SyncBuffer<
          Serializable>>();
3
4      Iterator<String> consIter = connectedProcesses.iterator();
5
6      while(consIter.hasNext()){
```

```
7           String idCon = consIter.next();
8           SyncBuffer<Serializable> buff = new SyncBuffer<
                Serializable>(10000);
9           incomingMessages.put(idCon, buff);
10      }
11  }
```

Comenzamos inicializando la tabla Hash en la línea 2. La línea 4 obtiene un iterador para los elementos de la lista de identificadores de los procesos conectados al *RemoteJob*. Entre las líneas 6 y 10, para cada identificador *idCon* en la lista *connectedProcesses*, se construye un nuevo buffer *SyncBuffer* de datos *Serializables* con capacidad para diez mil elementos[1]. Este buffer es agregado a la tabla Hash, identificándolo como el buffer de datos para el proceso cuyo identificador es *idCon*.

Cuando un mensaje que lleva como destino un proceso es recibido por el *Manager* que lo ejecuta, éste realiza una llamada al método *adminMessage()* del procesos receptor. Este método se escribe como sigue:

```
1   protected void adminMessage(Serializable data, String from){
2           incomingMessages.get(from).write(data);
3   }
```

El método *adminMessage()* recibe un objeto *Serializable* como parámetro, junto con el identificador del proceso emisor. Con esto se obtiene el buffer de mensajes de la tabla *incomingMessages* asociado al proceso *from* y se escribe en él el objeto *data*.

Para enviar datos, un proceso debe primero crear un objeto *Message* si el dato a enviar está destinado a otro proceso *RemoteJob*, o un objeto *MasterMessage* si está destinado al objeto *Master*. Una vez creados los mensajes, el proceso que los emite delega la tarea de enviarlos al *Manager* donde reside. El siguiente código muestra la implementación de los métodos *sayToMaster()* y *sendMessage()*:

```
1   protected void saytoMaster(Serializable data){
2           MasterMessage mm = new MasterMessage(this.id, data,
                Constants.DATA);
3           man.sendtoMaster(mm);
4   }
5
6   protected void sendMessage(String id, Serializable data){
7           Message m = new Message(this.id, id, data);
8           man.sendMessage(id, m);
```

[1]La capacidad del buffer es dispuesta arbitrariamente; sin embargo, consideramos que diez mil elementos son suficientes.

```
9   }
```

Dado que los datos que otros procesos envían están almacenados en un buffer de la tabla *incomingMessages*, cuando se desea recibir un mensaje proveniente de un proceso emisor, basta con buscar dicho mensaje en el buffer asociado al proceso emisor como muestra el siguiente código:

```
1   protected Serializable receiveMessage(String id){
2           return incomingMessages.get(id).read();
3   }
```

El método *receiveMessage()* se invoca con el identificador del proceso del cual se desea recibir un mensaje. Este método busca en la tabla *incomingMessages* el buffer asociado al proceso con identificador *id* e intenta leer un dato de dicho buffer, el cual sera el valor de retorno del método. La construcción del buffer (sección 4.1) bloquea el método *receiveMessage()* hasta que exista un dato que leer.

Finalmente, cuando un *RemoteJob* ha terminado su trabajo, debe informar de este hecho al *Manager* donde reside mediante el método *finished()*. Este método invoca a su vez al método *jobFinished()* que revisaremos más adelante en este capítulo. La invocación del método *finished()* es responsabilidad del programador de aplicaciones paralelas y debe ser la última instrucción del método *run()* implementado por el programador.

4.4. Sobre sockets y comunicación en Java

Para poder conectar dos computadoras virtuales en Java, hacemos uso de *sockets* y de las clases del paquete *java.net*.

> Dentro de un programa, un **socket** es una abstracción para el software de red, que permite la comunicación hacia adentro y hacia afuera del programa [7].

Así, para comunicar dos programas, cada uno debe "abrir" un socket que este ligado al socket del otro programa. El primer programa debe esperar a que el segundo intente abrir un socket para poder abrir su propio socket.

En Java, un socket esta representado por la clase *Socket*, cuyo constructor recibe como parámetros una dirección de red y un puerto de enlace. Para establecer comunicación con otro programa, consideremos la siguiente línea:

```
Socket s = new Socket("192.168.1.59",3630);
```

Esta línea abre un nuevo *Socket* para establecer comunicación con un programa que se ejecuta en la computadora cuya dirección es *192.168.1.59* y que transfiere datos a través del puerto 3630. Una vez creado, el *Socket s* contiene un stream de datos de entrada llamado *InputStream* del cual se pueden leer datos enviados desde el otro lado del socket y un stream de datos de salida llamado *OutputStream* en el cual se pueden escribir datos para enviarlos al programa del otro lado del socket. Estos streams de datos son accesados por medio de los métodos *getInputStream()* y *getOutputStream()*.

En este trabajo, utilizamos una técnica ligeramente diferente. Consideremos las siguientes líneas:

```
1  Socket s = new Socket();
2  s.connect(new InetSocketAddress("192.168.1.59",3630), 1000);
```

En la primer línea, se construye el *Socket s* sin realizar ninguna conexión al momento de su construcción. La segunda línea intenta conectar el *Socket s* a la dirección *192.168.1.59* por medio del puerto 3630 utilizando un objeto *InetSocketAddress* y un parámetro adicional que representa los milisegundos que el programa debe esperar una respuesta a la petición de conexión antes de lanzar una *InterruptedIOException*.

En el lado del programa que debe esperar a que algún otro programa intente establecer conexión con él, se deben considerar las siguientes líneas de código:

```
1  ServerSocket ss = new ServerSocket(3630);
2  Socket incSocket = ss.accept();
```

La línea 1 establece que el programa que la ejecuta recibe conexiones con sockets que quieran conectarse con el puerto 3630 de la computadora local. En la línea 2, cuando un socket en otro programa intenta abrir un socket con el puerto 3630, la llamada *ss.accept()* devuelve un socket que representa la conexión con el otro programa y lo almacena en la variable *incSocket*.

Cuando se quiere terminar la conexión con otro programa, basta con ejecutar el método *close()* del objeto *Socket* que se desea cerrar.

4.5. *Connection*: La conexión entre computadoras virtuales

Basamos la conexión entre computadoras virtuales en J-MIPS por medio del modelo de comunicación con *Sockets* revisado en la sección anterior. Para conectar

componentes en J-MIPS (*Master-Manager, Manager-Manager*), hacemos uso de la clase *Connection*, de la cual hablamos en la sección 3.3.5.

Básicamente, un objeto *Connection* almacena los elementos necesarios tanto para realizar como para esperar una petición de conexión con otra computadora virtual. Cuando un objeto *Connection* es construido, éste sabe exactamente con quién se va a conectar o de quién debe esperar conexión. Dado que muchas cosas pueden salir mal al intentar conectar dos computadoras, la mayoría de los métodos de la clase *Connection* pueden lanzar excepciones de tipo *ConnectionException*.

Los atributos de la clase *Connection* son los siguientes:

```
1  private String idLocal;
2  private String idToConnect;
3  private String ip;
4  private int reqPort;
5  private int waitPort;
6  private int timeout;
7
8  private Socket bound;
9  private ServerSocket ss;
10 private ObjectInputStream inStream;
11 private ObjectOutputStream outStream;
```

- La cadena de caracteres **idLocal** es el identificador de la computadora virtual donde se construye el objeto *Connection*.
- La cadena de caracteres **idToConnect** representa el identificador de la computadora virtual a la cual se conecta el objeto *Connection*.
- La cadena de caracteres **ip** es la dirección a la cual el objeto *Connection* debe conectarse.
- El entero **reqPort** es el puerto hacia el cual el objeto *Connection* debe conectarse en caso de que deba realizar una petición de conexión.
- El entero **waitPort** es el puerto por el cual el objeto *Connection* debe recibir peticiones de conexión en caso de que deba hacerlo.
- El entero **timeout** representa el tiempo de espera en milisegundos para establecer una conexión. Cuando se realiza una petición de conexión o se espera una conexión durante **timeout** milisegundos y la conexión no ha sido establecida, se lanza una excepción de tipo *ConnectionException*.

- El *Socket* **bound** representa el socket establecido entre el objeto *Connection* y otro objeto *Connection* en otra computadora virtual.
- El *ServerSocket* **ss** representa el objeto que espera peticiones de conexión con el objeto *Connection* de ser el caso.
- El *ObjectInputStream* **inStream** representa el stream de datos de entrada, del cual se pueden leer datos que otro objeto *Connection* haya enviado. De este stream se pueden leer objetos de tipo *Serializable* y se construye a partir del stream *InputStream* del *Socket bound*.
- El *ObjectOutputStream* **outStream** representa el stream de datos de salida, en el cual se pueden escribir datos que son enviados a otro objeto *Connection*. En este stream se pueden escribir objetos de tipo *Serializable* y se construye a partir del objeto *OutputStream* del *Socket bound*.

Mostramos el constructor de la clase *Connection* con el siguiente código:

```
1  public Connection(String idLocal, String idToConnect,String ip,
       int reqPort, int waitPort, int timeout){
2          this.ip = ip;
3          this.idLocal = idLocal;
4          this.idToConnect = idToConnect;
5          this.reqPort = reqPort;
6          this.timeout = timeout;
7          this.waitPort = waitPort;
8
9          bound = null;
10         ss = null;
11         inStream = null;
12         outStream = null;
13 }
```

En el constructor se inicializan todos los atributos de la clase excepto por el *Socket bound*, el *ServerSocket ss* y los streams de datos *inStream* y *outStream*, cuyo valor inicial, es *null*.

El primer método importante de la clase *Connection* permite establecer comunicación con otro objeto *Connection*. Llamamos a este método *requestConnection()*, cuya implementación es como en el siguiente código:

```
1  public void requestConnection() throws ConnectionException{
2      try{
3           bound = new Socket();
4           bound.connect(new InetSocketAddress(ip, reqPort),
               timeout);
```

```
5
6           outStream = new ObjectOutputStream(bound.getOutputStream
                ());
7           outStream.flush();
8
9           inStream = new ObjectInputStream(bound.getInputStream())
                ;
10
11          outStream.writeObject(idToConnect);
12          outStream.flush();
13
14          String idConf = (String) inStream.readObject();
15
16          if(!idConf.equals(idLocal)){
17              bound.close();
18              bound = null;
19              throw new ConnectionException("Remote␣connection␣was
                    ␣expecting:␣" + idConf +
20                      ".␣This␣connection␣is:␣" + idLocal);
21          }
22      }
23      catch(InterruptedIOException e){
24          bound = null;
25          throw new ConnectionException(idLocal + ":␣
                TimeoutReached", e);
26      }
27      catch(ConnectionException e){
28          throw e;
29      }
30      catch(Exception e){
31          try{
32              bound.close();
33          }
34          catch(IOException k){ }
35          bound = null;
36          throw new ConnectionException(idLocal + ":␣Can't␣connect
                ␣with␣" + idToConnect,e);
37      }
38  }
```

Se inicia la operación en las líneas 3 y 4. El *Socket bound* es construido e intenta realizar la conexión. Si ésta es exitosa, se construyen los streams de objetos *outStream* e *inStream* entre las líneas 6 y 9, utilizando los streams de datos *OutputStream* e *InputStream* del *Socket bound*. Entre las líneas 11 y 21 se lleva a cabo un procedimiento de verificación de que el objeto *Connection* está conectado con quien se supone que debe estarlo. Al establecer la comunicación, el objeto *Connection* envía el identificador de la computadora virtual con la que espera estar conectado. A su vez, desde el otro lado de la conexión se envía el identificador

de la computadora virtual remota conectada al objeto *Connection* local. Si el identificador recibido es igual al identificador de la computadora virtual local, eso quiere decir que el objeto remoto está conectado con quien se supone que debe estar conectado. De no ser así, el *Socket* se cierra y se lanza una excepción de tipo *ConnectionException*.

Las instrucciones anteriores están encerradas en un bloque *try-catch* en el cual deben atraparse varias excepciones:

- Una excepción de tipo *ConnectionException* lanzada en la línea 19.
- Una excepción de tipo *InterruptedIOException* lanzada en la línea 4.
- Cualquier otro tipo de excepción que pueda generarse en el proceso.

En cualquier caso, si existe un error al realizar la petición de conexión con otra computadora virtual, se lanza una excepción de tipo *ConnectionException*.

El método *waitConnection()* espera hasta que otro objeto *Connection* en otra computadora virtual intente establecer conexión por medio de su método *requestConnection()*. Este método esta implementado en el siguiente código:

```
1  public void waitConnection() throws ConnectionException{
2      try{
3          ss = new ServerSocket(waitPort);
4          Thread listenerT = new Thread(){
5            public void run(){
6                try{
7                    bound = ss.accept();
8                }
9                catch(IOException e){
10               }
11           }
12         };
13
14         listenerT.start();
15         listenerT.join(timeout);
16         ss.close();
17
18         if(bound == null){
19             throw new ConnectionException(idLocal +": Can't
                   Connect with " + idToConnect);
20         }
21
22         outStream = new ObjectOutputStream(bound.getOutputStream
               ());
```

```
23          outStream.flush();
24          inStream = new ObjectInputStream(bound.getInputStream());
25
26          outStream.writeObject(idToConnect);
27          String idConf = (String) inStream.readObject();
28
29          if(!idConf.equals(idLocal)){
30              bound.close();
31              bound = null;
32              throw new ConnectionException("Remote␣connection␣was␣
                    expecting:␣" + idConf +
33                          ".␣This␣connection␣is:␣" + idLocal);
34          }
35      }
36      catch(ConnectionException e){
37          throw e;
38      }
39      catch(IOException e){
40          try{
41              bound.close();
42          }
43          catch(IOException k){ }
44          bound = null;
45          throw new ConnectionException(idLocal + "␣Can't␣connect␣
                with␣" + idToConnect,e);
46      }
47      catch(Exception e){
48          throw new ConnectionException(e);
49      }
50  }
```

La función del método comienza en la línea 3 donde crea un *ServerSocket* que escucha a través del puerto *waitPort*. Entre las líneas 4 y 12 se implementa un Thread cuya única función consiste en hacer que el *ServerSocket ss* espere una petición de conexión desde la computadora virtual *idToConnect*. Cuando esta petición llega, la conexión se establece por medio del *Socket bound*. Las líneas 14 y 15 ponen en ejecución este Thread y esperan durante *timeout* milisegundos a que se establezca la conexión. Si después de este tiempo la computadora virtual *idToConnect* no ha realizado una petición de conexión, entonces el valor de *bound* se establece en *null*. En cualquier caso, el *ServerSocket ss* es cerrado en la línea 16. Si la conexión no pudo establecerse y el valor de *bound* es *null*, se lanza una excepción de tipo *ConnectionException*. En las líneas 22 a 24 se crean los streams de objetos *outStream* e *inStream* como en el método *requestConnection()* y al igual que en este método, se verifica que la conexión se haya establecido con la computadora virtual correcta entre las líneas 26 a 34.

Las excepciones capturadas para las instrucciones anteriores son similares a las del método *requestConnection()*. La línea 32 puede lanzar una excepción de tipo *ConnectionException*, las líneas 7,16,22-27 y 30 pueden lanzar excepciones de tipo *IOException*. Todas estas excepciones son capturadas para lanzar una única excepción de tipo *ConnectionException*.

Los métodos *sendMessage(Serializable s)* y *receiveMessage()* hacen uso de los streams *outStream* e *inStream* para enviar y recibir objetos de tipo *Serializable*. Ambos métodos pueden lanzar excepciones de tipo *ConnectionException*.

Finalmente, la clase *Connection* implementa el método *closeConnection()* que invoca al método *close()* del *Socket bound* y puede lanzar una *ConnectionException*.

4.6. La computadora virtual esclava *Manager*

En la sección 3.3.6 vimos el diseño de la clase *Manager*, que representa a las computadoras virtuales esclavas en J-MIPS, encargadas de ejecutar los procesos de ejecución remota especificados en el archivo de configuración.

Para hablar sobre la implementación de la clase *Manager* y durante el resto de este capítulo, hacemos uso de valores constantes que se encuentran en una clase llamada *Constants*. Sin embargo, en vez de presentar al lector la implementación de esta clase, solo hacemos referencia a los nombres de los valores constantes que contiene.

En esta sección, presentamos la descripción de los componentes de la clase *Manager* conforme son requeridos por otros componentes de la misma clase. Esto es, presentamos primero los componentes que son independientes entre sí y después los componentes que mantienen interacciones entre ellos.

La clase *Manager* tiene los siguientes atributos:

```
1      private String localId;
2      private int masterPort;
3      private int timeout;
4
5      private Connection masterCon;
6
7      private Hashtable<String,Connection> physicalCons;
8      private Hashtable<String,String> pMapping;
9      private Hashtable<String,RemoteJob> procs;
```

```
10
11      private SyncBuffer<MasterMessage> messagesToMaster;
12      private int finished;
```

El valor entero *finished* no lo tratamos en la sección 3.3.6 pero lleva la cuenta de cuantos procesos *RemoteJob* han terminado su ejecución. Mas adelante en esta sección veremos por qué es importante conocer este valor. De igual forma, el atributo *timeout* representa el tiempo en milisegundos que el *Manager* debe esperar a que las computadoras remotas establezcan conexión con él antes de lanzar una *ConnectionException*. El buffer de mensajes *messagesToMaster* representa al atributo *buf* de la sección 3.3.6.

El constructor de un objeto *Manager* recibe dos argumentos:

- El puerto por el cual se conecta con el objeto *Master*
- El tiempo de espera en milisegundos *timeout*.

Estos son los únicos valores que el *Manager* debe conocer para iniciar su operación.

```
1   public Manager(int masterPort, int timeout){
2           this.localId = "Manager";
3           this.masterPort = masterPort;
4           this.timeout = timeout;
5           this.finished = 0;
6
7           masterCon = new Connection(localId,"Master","",0,
                masterPort,timeout);
8           physicalCons = new Hashtable<String, Connection>();
9           pMapping = null;
10          procs = new Hashtable<String, RemoteJob>();
11
12          messagesToMaster = new SyncBuffer<MasterMessage>(Integer.
                MAX_VALUE - 10);
13  }
```

El atributo *localId* se inicia con el valor *"Manager"* en todos los constructores hasta que el objeto *Master* envíe el identificador real. Las tablas Hash *physicalCons* y *procs* se inician como tablas vacias, la tabla *pMapping* inicia con el valor *null* y el buffer de mensajes *messagesToMaster* inicia como un buffer vacío. Además, el constructor inicializa el objeto *masterCon* con los parámetros necesarios para establecer la conexión con el objeto *Master*.

Cuando el objeto *Master* envía al *Manager* la tabla de conexiones físicas que

éste debe establecer con otros *Managers*, esta tabla contiene objetos de tipo *ConnectionData*. El *Manager* que recibe esta tabla y a través de ella, debe poblar la tabla *physicalCons* que contiene objetos de tipo *Connection*. El método *initConnections()* realiza esta labor mediante el siguiente código:

```
1  private void initConnections(Hashtable<String,ConnectionData>
       cons){
2
3          Collection<ConnectionData> consCol = cons.values();
4          Iterator<ConnectionData> consIter = consCol.iterator();
5
6          while(consIter.hasNext()){
7              ConnectionData nextCon = consIter.next();
8
9              Connection newCon = new Connection(localId,nextCon.
                   getId(),nextCon.getIp(),
10                       nextCon.getPort(),masterPort,timeout);
11
12             physicalCons.put(nextCon.getId(), newCon);
13         }
14 }
```

El método *initConnections()* recibe como argumento una tabla Hash de objetos de tipo *ConnectionData* que es enviada por el objeto *Master*. Las líneas 3 y 4 construyen un iterador para recorrer los elementos de dicha tabla Hash. Para cada uno de estos elementos la línea 9 construye un objeto de tipo *Connection* con los datos necesarios para establecer una conexión con el *Manager* representado por el objeto *ConnectionData* en turno, utilizando información de este último y del propio *Manager* que ejecuta el método *initConnections()*. La línea 12 agrega la conexión creada a la tabla *physicalCons*[2].

Para establecer las conexiones con otras computadoras virtuales esclavas, los *Managers* utilizan los métodos *connectToAll()* y *waitConsFor()* introducidos en la sección 3.3.6. La implementación del método *connectToAll()* es como sigue:

```
1  private void connectToAll(){
2      Collection<Connection> cons = physicalCons.values();
3      Iterator<Connection> iterCons = cons.iterator();
4
5      boolean ok = true;
6
7      while(iterCons.hasNext()){
8          Connection con = iterCons.next();
```

[2]El lector debe recordar que cuando un objeto *Connection* es creado, no se establece la conexión, solo se le indican al objeto los elementos necesarios para hacerlo.

```
 9
10          if(!con.isConnected()){
11              try{
12                  con.requestConnection();
13              }
14              catch(ConnectionException e){
15                  ok = false;
16                  String strEx = this.localId + "␣could␣not␣connect
                        ␣with␣" + con.getIdToConnect() + "\n" + e.
                        getMessage();
17                  MasterMessage m = new MasterMessage(localId,
                        strEx, EXCEPTION);
18                  messagesToMaster.write(m);
19                  break;
20              }
21          }
22      }
23
24      if(ok){
25          MasterMessage m = new MasterMessage(localId,null,
                FULLY_CONNECTED);
26          messagesToMaster.write(m);
27      }
28  }
```

Las líneas 2 y 3 construyen un iterador para recorrer todas las conexiones asociadas al *Manager*. La variable booleana*ok* es una bandera que sirve para decidir si todo el proceso de conexión es exitoso y avisar de esto al objeto *Master*; su valor inicial es *true*. La operación principal del método se encuentra dentro del ciclo que comienza en la línea 7. Para cada *Connection* en la tabla *physicalConnections*, si dicha conexión aun no ha sido establecida, se invoca a su método *requestConnection()* para establecer la conexión con otro *Manager* en la línea 12. Si la petición de conexión es exitosa, se realiza lo mismo con el siguiente elemento de la tabla *physicalConnections*; de no ser así, la línea 12 lanza una excepción que es capturada en la línea 14 y manejada entre las líneas 15 y 19. El primer paso en el manejo de esta excepción es avisar al mismo método que algo no salio bien asignando a la variable *ok* el valor *false*. Lo siguiente es avisar del error al objeto *Master*, lo cual se hace entre las líneas 16 y 18. Finalmente, se ejecuta la instrucción *break* para salir del ciclo e interrumpir el proceso de conexión. Si ningún error ocurre durante el proceso de conexión, se avisa al objeto *Master* sobre este hecho en las líneas 25 y 26.

El código del método *waitConsFor()* requiere más atención en las excepciones que pueden ocurrir que en el trabajo que hay que realizar:

```
1  private void waitConsFor(String id){
2      Connection con = physicalCons.get(id);
3
4      if(con != null){
5          if(!con.isConnected()){
6              try{
7                  con.waitConnection();
8              }
9              catch(ConnectionException e){
10                 String strEx = localId + "␣failed␣to␣connect␣with
                       ␣" + id + "\n" +
11                         e.getMessage();
12                 MasterMessage m = new MasterMessage(localId,
                       strEx, EXCEPTION);
13                 messagesToMaster.write(m);
14             }
15         }
16     }
17     else{
18         String message = "Id␣" + id + "␣does␣not␣exist␣on␣
               connections␣table␣in" +
19                 localId;
20         MasterMessage m = new MasterMessage(localId, message,
               EXCEPTION);
21         messagesToMaster.write(m);
22     }
23 }
```

Este método recibe como argumento el identificador del *Manager* del cual se debe esperar una petición de conexión. La línea 2 obtiene de la tabla de conexiones *physicalCons* el objeto *Connection* asociado al *Manager* del cual se espera una petición. Para que pueda ser invocado el método *waitConnection()* de la línea 7 son necesarias dos condiciones:

- El objeto *Connection* no debe ser nulo.
- El objeto *Connection* debe estar desconectado.

Si estas dos condiciones se cumplen se llama al método *waitConnection()* del objeto *Connection*, el cual puede lanzar una *ConnectionException* que es capturada en la línea 9 y manejada entre las líneas 10 y 13, donde si esta excepción ocurre, se avisa al objeto *Master*. La verificación de la línea 4 es necesaria pues el método *get()* de la tabla *physicalCons* regresa *null* si se pasa como argumento un identificador que no esté contenido en ella. En este caso, se avisa del error al objeto *Master* entre las líneas 18 y 21.

Hemos mencionado que para un *Manager*, por cada conexión de la tabla *physicalCons*, existe un thread encargado de recibir y entregar los mensajes que lleguen desde otros *Managers*. La clase *NormalReceiver* representa a estos threads y está implementada de la siguiente forma:

```
1  private class NormalReceiver extends Thread{
2
3      Connection conrec;
4
5      public NormalReceiver(Connection conrec){
6          this.conrec = conrec;
7      }
8
9      public void run(){
10         try{
11             while(true){
12                 Message m = (Message) conrec.receiveMessage();
13                 String to = m.getIdTo();
14                 String from = m.getIdFrom();
15                 Serializable data = m.getData();
16                 RemoteJob recP = procs.get(to);
17
18                 recP.adminMessage(data, from);
19             }
20         }
21         catch(ConnectionException e){
22             String erStr = Manager.this.localId + "␣can't␣receive
                   ␣data␣from␣"+
23                         conrec.getIdToConnect() + "␣"+ e.getMessage()
                               ;
24             MasterMessage mm = new MasterMessage(Manager.this.
                   localId, erStr, Constants.EXCEPTION);
25             sendtoMaster(mm);
26         }
27     }
28 }
```

La clase *NormalReceiver* es una sub-clase de *Thread* y contiene como atributo un objeto *Connection* llamado *conrec* que es la conexión con otro *Manager*. El constructor en la línea 5 recibe como argumento dicha conexión. El trabajo que este thread debe realizar se muestra en el método *run()* entre las líneas 10 y 26 y consiste en un ciclo infinito. En este ciclo primero se recibe un mensaje de tipo *Message* en la línea 12. Entre las líneas 13 y 15 se obtiene del mensaje recibido la siguiente información:

- El identificador del proceso de destino *to*

- El identificador del proceso emisor del mensaje *from*
- El dato serializable enviado por el emisor *data*

Además, se obtiene en la línea 16 el proceso *RemoteJob* receptor del mensaje mediante la tabla *procs* y utilizando el identificador *to*. Lo único que falta es entregar los datos enviados al proceso receptor mediante la llamada *adminMessage* indicándole quién los envía. Esto se hace en la línea 18. Al recibir un mensaje en la línea 12, se puede producir una *ConnectionException* que es capturada en la línea 21 y manejada entra las líneas 22 y 25 en las cuales se manda un mensaje al objeto *Master* avisándole sobre el error. Esta excepción interrumpe el flujo del ciclo infinito terminando así la ejecución del thread *NormalReceiver*.

Existe en la clase *Manager* un método llamado *startReceivers()*, el cual crea y ejecuta un *NormalReceiver* por cada conexión de la tabla *physicalCons*. Su implementación es la siguiente:

```
1  private void startReceivers(){
2      Collection<Connection> cons = physicalCons.values();
3      Iterator<Connection> consIter = cons.iterator();
4
5      while(consIter.hasNext()){
6          NormalReceiver nr = new NormalReceiver(consIter.next());
7          nr.start();
8      }
9  }
```

El método anterior es invocado por otro método de la clase *Manager* llamado *startProcesses*. Este último es invocado por el *Manager* cuando el objeto *Master* le instruye iniciar la ejecución de todos los procesos *RemoteJob*. Es decir, el método *startProcesses()* permite que el *Manager* reciba mensajes de otros *Managers* y pone en ejecución los procesos asignados por el programador. La implementación de este método es como en el código siguiente:

```
1  private void startProcesses(){
2      startReceivers();
3
4      Collection<RemoteJob> jobs = procs.values();
5      Iterator<RemoteJob> iterJobs = jobs.iterator();
6
7      while(iterJobs.hasNext()){
8          Thread t = new Thread(iterJobs.next());
9          t.start();
10     }
11 }
```

En la línea 2 se inician los procesos receptores de mensajes del *Manager*. Las líneas 4 y 5 construyen un iterador para recorrer los procesos *RemoteJob* de la tabla *procs*. En la línea 8, se crea un thread que recibe como argumento un objeto de tipo *Runnable*. Afortunadamente, los objetos *RemoteJob* son de este tipo y por tanto contienen un método llamado *run()*. Este método es utilizado por el *Thread* t de la línea 8 como el trabajo que debe realizar. Así, la instrucción *Thread* t = *new Thread(RemoteJob rj)* quiere decir:

> Crea un nuevo thread llamado t y utiliza el método *run()* del *RemoteJob rj* como el trabajo que debe realizar.

La línea 9 inicia la ejecución del nuevo thread creado y ejecuta un proceso que en principio, fue definido en otra computadora. Este proceso es repetido por cada *RemoteJob* de la tabla *procs*.

Cuando un mensaje es recibido desde el objeto *Master*, el método *processInstruction()* lo examina y decide que acción tomar respecto a ese mensaje. La implementación de este método es la siguiente:

```
1  private void processInstruction(MasterMessage m){
2     switch(m.getMessageType()){
3          case WAIT_CONNECTION:
4              waitConsFor((String)m.getData());
5              break;
6          case DO_CONNECTION:
7              connectToAll();
8              break;
9          case START_PROCESSES:
10             startProcesses();
11             break;
12         case KILL:
13             System.exit(0);
14             break;
15    }
16 }
```

Este método recibe como argumento un mensaje *MasterMessage* y examina el tipo de este mensaje en la línea 2, que en realidad, es una instrucción que el *Manager* debe ejecutar. Las instrucciones pueden ser las siguientes:

- **WAIT_CONNECTION**. El *Manager* debe ejecutar el método *waitConsFor()* para recibir una petición de conexión desde el *Manager* indicado en el atributo *data* del mensaje.
- **DO_CONNECTION.** El *Manager* debe ejecutar el método *connectToAll()* para conectarse con otros *Managers*.

- **START_PROCESSES.** El *Manager* debe ejecutar el método *startProcesses()* para iniciar la ejecución de los procesos *RemoteJob*.
- **KILL.** El *Manager* debe ejecutar la instrucción de Java *System.exit(0)* para terminar la ejecución del programa principal en la computadora física local.

Las instrucciones anteriores están definidas como números enteros (int) en la clase *Constants*.

Mencionamos en la sección 3.3.6 que los *Managers* ejecutan dos Threads importantes para mantener comunicación con el objeto *Master*: *MasterSender* y *MasterReceiver*. El primero envía a *Master* los mensajes almacenados en el buffer *messagesToMaster* y el segundo recibe y procesa los mensajes que lleguen desde la conexión *masterCon*.

El trabajo del Thread *MasterSender* es como sigue:

```
1  public void run(){
2     try{
3          while(true){
4               MasterMessage m = messagesToMaster.read();
5               masterCon.sendMessage(m);
6          }
7     }
8     catch(ConnectionException e){
9          if(masterCon.isConnected()){
10              try{
11                   masterCon.closeConnection();
12              }
13              catch(ConnectionException k){}
14              System.exit(0);
15         }
16    }
17 }
```

En realidad, el trabajo principal de este Thread se lleva a cabo entre las líneas 3 y 6 del código anterior. La línea 4 lee un mensaje del buffer *messagesToMaster* y lo envía al objeto *Master* a través de la conexión *masterCon* en la línea 5; estas dos operaciones se repiten en un ciclo hasta que se termine la ejecución del programa principal. Sin embargo, la instrucción *masterCon.sendMessage(m)* puede lanzar una *ConnectionException* que es capturada en la línea 8 y manejada entre las líneas 9 y 15. Si ocurre un error de este tipo, el programa intenta cerrar la conexión con el objeto *Master* antes de terminar la ejecución de todo el sistema en la línea 14. Es muy probable que si hubo una excepción al intentar enviar un mensaje al objeto *Master*, exista otra del mismo tipo al intentar cerrar la conexión

con ésta. Al capturar esta excepción, el *MasterSender* simplemente la ignora y termina la ejecución del programa.

La implementación del trabajo que realiza el thread *MasterReceiver* es como sigue:

```
1  public void run(){
2     try{
3          while(true){
4               MasterMessage m = (MasterMessage) masterCon.
                    receiveMessage();
5               processInstruction(m);
6          }
7     }
8     catch(ConnectionException e){
9          System.exit(0);
10    }
11 }
```

El trabajo de este thread se lleva a cabo entre las líneas 3 y 6. En la línea 4 se recibe un mensaje desde el objeto *Master* a través de la conexión *masterCon* que se procesa en la línea 4; estas dos operaciones se llevan a cabo en un ciclo que termina hasta que finaliza la ejecución del programa principal. La línea 4 puede lanzar una *ConnectionException* al intentar recibir un mensaje desde la conexión con el objeto *Master*. Esta excepción es capturada en la línea 8 y manejada en la línea 9 terminando la ejecución del programa principal.

Después de ser creados, los *Managers* deben esperar a que el objeto *Master* se conecte con ellos. Los *Managers* hacen esto por medio del método *connectToMaster()* cuya implementación es la siguiente:

```
1  public void connectToMaster() throws ConnectionException{
2     try{
3          if(!masterCon.isConnected()){
4               masterCon.waitConnection();
5
6               MasterMessage idMessage = (MasterMessage)masterCon.
                    receiveMessage();
7               this.localId = (String) idMessage.getData();
8
9               MasterMessage consMessage = (MasterMessage)masterCon.
                    receiveMessage();
10              initConnections( (Hashtable<String,ConnectionData>)
                    consMessage.getData());
11
12              MasterMessage mapMessage = (MasterMessage) masterCon.
                    receiveMessage();
```

```
13              pMapping = (Hashtable<String,String>)mapMessage.
                    getData();
14
15              MasterMessage procsMessage = (MasterMessage) masterCon
                    .receiveMessage();
16              procs = (Hashtable<String,RemoteJob>)  procsMessage.
                    getData();
17
18              Collection<RemoteJob> procesos = procs.values();
19              Iterator<RemoteJob> procsIter = procesos.iterator();
20
21              while(procsIter.hasNext()){
22                  RemoteJob job = procsIter.next();
23                  job.setMannager(this);
24                  job.initializeBuffers();
25              }
26
27              MasterReceiver mr = new MasterReceiver();
28              mr.start();
29
30              MasterSender ms = new MasterSender();
31              ms.start();
32          }
33      }
34      catch(ConnectionException e){
35          throw new ConnectionException("Can␣not␣connect␣to␣Master",
                    e);
36      }
37  }
```

La línea 3 de este método se asegura de que el procedimiento de conexión con el objeto *Master* se ejecute solo si no se ha ejecutado antes. Este procedimiento inicia en la línea 4 donde se le instruye al objeto de conexión con el objeto *Master* llamado *masterCon* que espere una petición de conexión desde esta última. Una vez establecida la conexión con el objeto *Master*, el *Manager* recibe el identificador que le corresponde en las líneas 6 y 7, las conexiones que debe establecer con otros *Managers*, la tabla *pMapping* y la tabla de procesos *RemoteJob* que deben ser ejecutados se reciben y procesan entre las líneas 9 y 16. Al momento de la recepción de los procesos, los siguientes puntos deben ser tomados en cuenta:

- Los procesos no saben quién es el *Manager* que los va a ejecutar. Este dato es indispensable en los procesos para que puedan enviar mensajes. Cuando son creados por el programador, el constructor de los *RemoteJob* inicia la variable *man* (el *Manager* donde deben residir) con el valor *null*. Hasta el momento de la recepción de los procesos en el *Manager*, este valor no ha cambiado y el proceso es incapaz de enviar mensajes a otros procesos.

- Los buffers de mensajes entrantes en los procesos *RemoteJob* no han sido inicializados apropiadamente. Al igual que la variable *man*, la variable *incomingMessages* también es inicializada con el valor *null* al momento de la creación de los *RemoteJob* y hasta el momento de recepción de estos procesos en el *Manager*, este valor no ha cambiado. Implementamos este comportamiento para hacer que los procesos *RemoteJob* ocupen menos memoria y su envío desde el objeto *Master* hasta los *Managers* sea más rápido.

El primer punto es solucionado en la línea 23, donde el *Manager* avisa a cada *RemoteJob* que es *él* con quien hay que hablar para enviar un mensaje.

El segundo punto es resuelto en la línea 24 donde se invoca al método *initializeBuffers()* de cada *RemoteJob* para crear los buffers de mensajes que lleguen desde otros procesos. Finalmente, entre las líneas 27 a 31 se crean y ponen en ejecución los procesos *MasterReceiver* y *MasterSender* para interactuar con el objeto *Master*. Dado que hasta este momento no hay ningún mensaje que entregar, el *Manager* queda a la espera de una instrucción por parte del objeto *Master* que sea recibida por el thread *MasterReceiver*.

El proceso anterior de conexión con el objeto *Master* puede lanzar excepciones de tipo *ConnectionException* que son capturadas en la línea 34 y re-lanzadas en la 35.

Hemos mencionado en esta sección que los mensajes que son enviados por otros *Managers*, son recibidos por el thread *NormalReceiver* y entregados por este a los procesos *RemoteJob*. Para enviar un dato a otro proceso, un *RemoteJob* debe hacerlo por medio del método *sendMessage()* del *Manager* donde reside. La implementación de este método está dada por el siguiente código:

```
1  protected void sendMessage(String id, Message mes){
2     try{
3         if(procs.containsKey(id)){
4             RemoteJob job = procs.get(id);
5             job.adminMessage(mes.getData(), mes.getIdFrom());
6         }
7         else if(pMapping.containsKey(id)){
8             String idMap = pMapping.get(id);
9             physicalCons.get(idMap).sendMessage(mes);
10        }
11        else{
12            throw new IOException("Process␣" + id + "␣not␣found␣in
                 ␣mannager␣" + localId);
13        }
14    }
15    catch(ConnectionException e){
```

```
16          String erStr = localId + "␣can't␣send␣Message␣to␣" + id +
                "␣in␣" + pMapping.get(id)
17                  + "\n" + e.getMessage();
18          MasterMessage mm = new MasterMessage(this.localId,erStr,
                Constants.EXCEPTION);
19          sendtoMaster(mm);
20      }
21      catch(IOException e){
22          String erStr = e.getMessage();
23          MasterMessage mm = new MasterMessage(this.localId,erStr,
                Constants.EXCEPTION);
24          sendtoMaster(mm);
25      }
26  }
```

El método *sendMessage()* recibe como argumentos el identificador del mensaje de destino y el mensaje *Message* que debe ser enviado. Como mencionamos en la sección 3.3.6, el *Manager* debe decidir si el mensaje debe ser enviado a otro *Manager* o entregado a alguno de sus procesos. Esta decisión se toma en el método *sendMessage()* evaluando el argumento *id* del proceso de destino, el cual puede caer en los siguientes casos:

- El identificador del proceso de destino se puede encontrar en la tabla *procs* (línea 3). Esto quiere decir que el proceso de destino es parte del conjunto de procesos que ejecuta el *Manager* local, en cuyo caso se entrega el mensaje a dicho proceso mediante su método *adminMessage()* (líneas 4 y 5).

- El identificador del proceso de destino no se encuentra en la tabla *procs* pero se encuentra en la tabla *pMapping* (línea 7). Esto significa que el proceso de destino se encuentra en otro *Manager*. En este caso, el método *sendMessage()* busca en cuál *Manager* se ejecuta el proceso de destino por medio de la tabla *pMapping* y envía el mensaje a este *Manager* a través de la conexión *Connection* establecida con él (líneas 8 y 9).

- No ocurre ninguno de los casos anteriores, lo cual significa que el identificador proporcionado es erróneo o el archivo de configuración no concuerda con la especificación de los procesos *RemoteJob*. En cualquier caso, es lanzada una *IOException*.

El procedimiento para enviar un mensaje puede lanzar una excepción de tipo *IOException* en la línea 12 o una *ConnectionException* en la línea 9. En cualquier caso, la respuesta a estas excepciones es enviar un mensaje al objeto *Master* indicándole el error generado (líneas 15 a 25).

En la sección 4.3 se ve que cuando un proceso *RemoteJob* termina su trabajo, debe avisar al *Manager* este hecho mediante el método *jobFinished()* de la clase *Manager*. La implementación de este método es como sigue:

```
1  protected synchronized void jobFinished(){
2     finished++;
3
4     if(finished == procs.size()){
5         MasterMessage message = new MasterMessage(localId, null,
              DONE);
6         messagesToMaster.write(message);
7     }
8  }
```

Recordemos que la variable *finished* cuenta el número de procesos que han terminado su trabajo. De esta forma, la primer acción que debe tomarse cuando un *RemoteJob* termina es incrementar el valor de la variable *finished* en la línea 2. En la línea 4, si el número de procesos que han terminado su trabajo es igual al número de procesos que se ejecutan en el *Manager*, entonces se envía un mensaje al objeto *Master* indicándole que el trabajo total del *Manager* ha terminado. Esta acción se realiza en las líneas 5 y 6. El método *jobFinished()* debe ser *synchronized* para evitar corrupción de la variable *finished*.

4.7. Master: El administrador

En la sección 3.3.7 de este trabajo se revisaron los aspectos de diseño del objeto *Master*, así como sus interacciones con el resto de los componentes del sistema. En esta sección, revisamos la implementación del *Master*, para la cuál utilizamos una clase en Java con el mismo nombre. Particularmente, tratamos el análisis del archivo de configuración del que hemos hablado desde el capítulo 4 sin especificar como es que se realiza. Desafortunadamente, la clase *Master* cuenta con cerca de 1000 líneas de código, por lo cual hemos optado por mostrar al usuario solo las instrucciones de los métodos que consideramos más importantes para comprender el funcionamiento del objeto *Master*.

La clase *Master* contiene los siguientes atributos:

```
1      private String configFile;
2      private int timeout;
3      private int bufferSize;
4
5      private Hashtable<String,ManagerData> mannagerDataTable;
6      private Hashtable<String,ManagerConnection> mannagerTable;
```

```
7       private Hashtable<String,RemoteJob> processTable;
8       private Hashtable<String,ProcData> procDataTable;
9       private Hashtable<String,SyncBuffer<Serializable>>
           procsMessages;
10
11      private SyncBuffer<MasterMessage> incomingMessages;
12
13      private int finished;
14
15      private boolean done;
```

Hemos revisado casi todos los atributos anteriores en la sección 3.3.7. Sin embargo, en esta sección se muestran algunos otros que involucran detalles puramente técnicos:

- **configFile** es una cadena de caracteres que especifica el nombre del archivo de configuración.
- **timeout** es el tiempo de espera para establecer las comunicaciones del sistema antes de lanzar una excepción.
- **bufferSize** es el tamaño máximo que puede alcanzar un buffer de datos.
- **finished** es un entero que indica el número de *Managers* que han terminado su trabajo. Su utilización es muy similar a la variable del mismo nombre de la clase *Manager*.
- **done** es una variable booleana cuyo valor inicial es *false* y cambia a *true* cuando todos los *Managers* terminan la ejecución de sus procesos.

El constructor de la clase *Master* es como sigue:

```
1   public Master(String configFile,int timeout, int bufferSize){
2           this.timeout = timeout;
3           this.configFile = configFile;
4           this.bufferSize = bufferSize;
5           this.finished = 0;
6           this.done = false;
7           ...
8   }
```

Este constructor acepta como argumentos el nombre del archivo de configuración, el tiempo de tolerancia para las conexiones y el tamaño máximo de los buffers de datos. El número de *Managers* que han terminado su trabajo es fijado en 0 (cero) y se asigna el valor *false* a la variable *done*. El resto de los atributos (tablas Hash y buffers) son inicializados como estructuras vacías.

Alternativamente, también existe un constructor que recibe únicamente el nombre del archivo de configuración:

```
1  public Master(String configFile){
2          this(configFile,10000,(Integer.MAX_VALUE-10));
3  }
```

4.7.1. Análisis del archivo de configuración

El primer paso para obtener información a partir del archivo de configuración es revisar que esté bien escrito. Nuestro objetivo es entonces implementar un aceptador que pueda decidir si las líneas del archivo son válidas.

Recordemos que en el archivo de configuración, los *Managers* se especifican por medio de líneas como la siguiente:

$$\text{Manager} = (\text{dirección,puerto,identificador})$$

Donde *dirección* e *identificador* son cadenas de caracteres y *puerto* es un número entero. Además, la cadena *dirección* puede estar construida por un identificador alfanumérico de cualquier longitud o por una serie de subcadenas de caracteres separadas por un punto. Centrando nuestra atención únicamente en la subcadena *(dirección,puerto,identificador)*, las líneas marcadas como *Manager* en nuestro archivo de configuración deben entonces ser aceptadas por la siguiente expresión regular:

$$(ww^*(.ww^*)^*,dd^*,ww^*)$$

Donde en esta expresión regular, w es un carácter alfanumérico y d es un dígito.

De igual forma, en el archivo de configuración, las líneas que especifican procesos son como la siguiente:

$$\text{Process} = (\text{identificador,Manager,[p1,p2,...,pn]})$$

Donde *identificador*, *Manager*, *p1*, ... , *pn* son cadenas de caracteres. Al igual que con las líneas de *Manager*, si centramos nuestra atención en la subcadena *(identificador,Manager,[p1,p2,...,pn])*, la expresión regular que acepta estas cadenas es la siguiente:

$$(ww^*,ww^*,[ww^*(,ww^*)^*])$$

Utilizando estas expresiones regulares sobre las líneas del archivo de configuración, podemos decidir si éste está bien escrito para especificar los *Managers* y procesos que se ejecutan en ellos.

Una vez que se ha verificado que el archivo de configuración está correctamente escrito, el siguiente paso es obtener información de él y almacenarla en objetos *ManagerData* y *ProcData*. Inicialmente, únicamente separando los componentes de una línea *Manager*, se puede crear un objeto *ManagerData* que encapsule la dirección, el puerto y el identificador de cada *Manager*. El método *hostAn()* se encarga de esta tarea mediante el siguiente código:

```
1  private void hostAn(String manager) throws ConFileException{
2      String [] comps = manager.split(",");
3      try{
4          String ip = comps[0].substring(1).trim();
5          int port = Integer.parseInt(comps[1].trim());
6          String id = comps[2].substring(0, comps[2].length()-1).
               trim();
7
8          ManagerData newMan = new ManagerData(id,ip,port);
9
10         if(managerDataTable.containsKey(id)){
11             throw new ConFileException("Manager␣duplicated:␣" +
                   id);
12         }
13         else{
14             mannagerDataTable.put(id, newMan);
15         }
16     }
17     catch(ConFileException e){
18         throw new ConFileException("Exception␣in␣line:␣" +
               manager, e);
19     }
20 }
```

El método *hostAn()* evalúa una cadena de la forma *(dirección,puerto,identificador)*. La línea 2 divide la cadena *manager* en tres subcadenas:

- Dirección.
- Puerto.
- Identificador.

Estas sub-cadenas son almacenadas en el arreglo *comps[]*. Las líneas 4 a 6 extraen la información de las cadenas del arreglo *comps[]* con la cual se crea un nuevo

objeto *ManagerData* en la línea 8. Finalmente, el objeto creado es agregado a la tabla Hash *managerDataTable* en la línea 14, si es que no existe en esa tabla otro objeto con el mismo identificador, en cuyo caso se lanza una *ConfFileException* en la línea 11.

De igual forma, con un algoritmo muy similar, se puede poblar la tabla *procDataTable* con objetos *ProcData* obtenidos a partir de líneas *Process* del archivo de configuración. Dichos objetos tienen asignado un identificador, el *Manager* en el que residen y una lista de identificadores de otros procesos con los cuales están conectados. Todo esto dentro de un método que llamamos *processAn()*.

Tres pasos son necesarios para completar la información de los objetos *ManagerData* y *ProcData*:

1. Los procesos *RemoteJob* deben saber con cuáles otros procesos están conectados para inicializar los buffers de mensajes recibidos.
2. La topología de la red de *Managers* y las tablas *pMapping* de estos deben ser determinadas y almacenadas en los correspondientes objetos *ManagerData*.
3. Para cada objeto *ManagerData*, se debe determinar la lista de procesos que debe ejecutar el correspondiente *Manager*.

Para el punto 1, basta con pasar a cada *RemoteJob* la lista de identificadores almacenada en el objeto *ProcData* correspondiente. En el punto 3, se debe realizar una revisión sobre los objetos *ProcData* como la del siguiente código:

```
1  for(ProcData pr: procDataTable){
2       String manager = pr.getHost();
3       ManagerData man = mannagerDataTable.get(manager);
4       man.addJobToDo(pr.getId());
5  }
```

En línea 1 del algoritmo anterior se revisan todos los objetos *ProcData*. En la línea 3, se obtiene el *ManagerData* que representa al *Manager* que va a ejecutar el proceso representado por el *ProcData pr*. Finalmente, en la línea 4, se notifica este hecho al *ManagerData*.

El punto número 2 representa un problema más difícil. Para poder abordarlo, consideremos nuevamente el principio de conexión planteado en la sección 3.3.8:

> Sean p y q procesos de tipo *RemoteJob*; y sean M y N *Managers* tales que p se ejecuta en M y q en N. Si p y q intercambian información por medio de paso de mensajes, entonces **existe una conexión**

virtual entre M y N, la tupla (q,N) aparece en la tabla *pMapping* de M y la tupla (p,M) aparece en la tabla *pMapping* de N.

Siguiendo este principio, podemos generar el siguiente código para solucionar el problema del punto 2:

```
1  Collection<ProcData> procesos = procDataTable.values();
2  Iterator<ProcData> iterProcs = procesos.iterator();
3
4  while(iterProcs.hasNext()){
5      ProcData evaluado = iterProcs.next();
6      String hostEvaluado = evaluado.host;
7
8      LinkedList<String> consEvaluado = evaluado.getConnections();
9      Iterator<String> consevIt = consEvaluado.iterator();
10
11     while(consevIt.hasNext()){
12         String condeEvaluado = consevIt.next();
13         ProcData toConnect = procDataTable.get(condeEvaluado);
14
15         if(toConnect != null){
16             String hostToConnect = toConnect.host;
17
18             if(!hostEvaluado.equals(hostToConnect)){
19                 ManagerData hostEvMan = mannagerDataTable.get(
                       hostEvaluado);
20                 ManagerData hostToCon = mannagerDataTable.get(
                       hostToConnect);
21
22                 if( (hostEvMan != null) && (hostToCon != null)){
23                     ConnectionData evto = new ConnectionData(
                           hostToCon.getId(),hostToCon.getIp(),
                           hostToCon.getPort(),timeout);
24                     ConnectionData toev = new ConnectionData(
                           hostEvMan.getId(),hostEvMan.getIp(),
                           hostEvMan.getPort(),timeout);
25
26                     hostEvMan.addConnection(evto);
27                     hostToCon.addConnection(toev);
28                 }
29                 else{
30                     throw new ConFileException("Mannager␣"+
                           hostEvaluado + "␣or␣"+hostToConnect+"␣not␣
                           found");
31                 }
32                 hostEvMan.addProcessMap(toConnect.getId(),
                       toConnect.getHost());
```

```
33                  hostToCon.addProcessMap(evaluado.getId(),
                        evaluado.getHost());
34              }
35          }
36          else{
37              throw new ConFileException("Process␣" + condeEvaluado
                    + "␣not␣found");
38          }
39      }
40  }
```

El código anterior puede ser difícil de entender a primera vista, pero la idea principal es la siguiente:

1. Se toma un proceso p y su lista de procesos conectados (líneas 5-10).

2. Se toma cada proceso q en la lista y se evalúan el *Manager* donde se ejecuta p, digamos M y el *Manager* donde se ejecuta q, digamos N (líneas 11-16).

3. Si el *Manager* M es diferente al *Manager* N, entonces se le agrega a M una conexión hacia N y viceversa (líneas 18-28).

4. Cuando p envíe un mensaje a q, debe hacerlo a través de M, por lo cual éste debe saber que q se ejecuta en N. Así, el mapeo (q,N) es agregado a la tabla *pMapping* de M. El caso recíproco obliga a incluir el mapeo (p,M) a la tabla *pMapping* de N (líneas 32-33).

5. El procedimiento anterior se ejecuta para cada proceso p en el objeto *Master* (líneas 1-4).

Al ejecutarse en el objeto *Master*, el algoritmo anterior no trabaja con los componentes reales del sistema, sino con sus representantes *ProcData* y *ManagerData*.

Todos los procedimientos que hemos visto en esta sección sobre analizar el archivo de configuración son integrados en el método *processConfFile()* de la clase *Master*.

4.7.2. Ejecución del sistema

En la sección 3.3.7 revisamos un método de la clase *Master* llamado *communicateManagers()*, el cual organiza a las computadoras virtuales esclavas para implementar la topología de estas. La implementación de este método es como sigue:

```
1  private void communicateMannagers()...{
2      Collection<ManagerData> mannagers = mannagerDataTable.values
           ();
3      Iterator<ManagerData> mdIterator = mannagers.iterator();
4
5      while(mdIterator.hasNext()){
6          ManagerData md = mdIterator.next();
7
8          Hashtable<String,ConnectionData> connections = md.
               getConnectionsTable();
9          Collection<ConnectionData> conCol = connections.values();
10         Iterator<ConnectionData> connectIterator = conCol.
               iterator();
11
12         while(connectIterator.hasNext()){
13             ConnectionData dat = connectIterator.next();
14
15             MasterMessage m = new MasterMessage("Master", md.
                   getId(), WAIT_CONNECTION);
16             ManagerConnection mc = mannagerTable.get(dat.getId())
                   ;
17             mc.sendMessage(m);
18
19             MasterMessage confirm = mc.receiveMessage();
20             ...
21         }
22
23         MasterMessage mCon = new MasterMessage("Master",null,
               DO_CONNECTION);
24         ManagerConnection mdConnection = mannagerTable.get(md.
               getId());
25         mdConnection.sendMessage(mCon);
26
27         MasterMessage confConnection = mdConnection.
               receiveMessage();
28         ...
29     }
30 }
```

Recordemos que los objetos de tipo *ManagerData* almacenan, entre otras cosas, una lista de los identificadores de los *Managers* con los cuales están conectados. La idea principal entonces consiste en tomar a un *Manager* M y a los de su lista de conexiones. A estos se les debe enviar un mensaje diciéndoles que esperen a que M intente establecer conexión con ellos y mandar otro mensaje a M diciéndole que debe realizar la petición de conexión con los otros *Managers*.

Las líneas 2 y 3 construyen un iterador sobre los elementos *ManagerData*. Entre

las líneas 6 y 10 se toma uno de estos elementos (*md*)y se construye un iterador sobre la lista de conexiones del *ManagerData* evaluado. Digamos que el *Manager* representado por *md* tiene el identificador *man_k*. Entonces, a cada *Manager* que deba estar conectado a *man_k*, se le manda un mensaje pidiéndole que espere una petición de conexión de éste y se espera un mensaje de confirmación entre las líneas 12 y 21. A *man_k* se le envía a su vez una petición para establecer conexión con todos los elementos de su lista de conexiones en la línea 25. Finalmente, se espera a que *man_k* regrese un mensaje indicando que la conexión ha sido exitosa en la línea 27. En este momento, el *Manager man_k* esta completamente conectado con todos los *Managers* con los que debe estarlo. Este procedimiento se repite para cada *ManagerData* en la tabla *managerDataTable*; es decir, para cada *Manager* del sistema.

Una vez en ejecución, el programador debe esperar a que el sistema completo termine su ejecución. Esto se hace mediante el método *waitToFinish()* de la clase *Master*. Su implementación es la siguiente:

```
1  public synchronized void waitToFinish(){
2      try{
3          wait();
4      }
5      catch(InterruptedException e){
6          System.out.println("InterrupdedException␣calling␣wait(),␣
               system␣will␣shut␣down␣");
7          e.printStackTrace();
8          killManagers();
9          System.exit(1);
10     }
11 }
```

El método *waitToFinish()* bloquea la ejecución de quién lo llame mientras exista un *Manager* que aún no haya terminado su ejecución (línea 3). La llamada al método *wait()* puede generar una *InterruptedException* que provoca que el sistema detenga toda su ejecución (líneas 6-9). Para provocar que el proceso que llama a *waitToFinish()* se desbloquee, existe en la clase *Master* el método *managerDone()* que se ejecuta cuando un *Manager* avisa a la computadora *Manager* el fin de la ejecución de sus procesos:

```
1   private synchronized void managerDone(){
2      finished++;
3      if(finished == mannagerTable.size()){
4          done = true;
5          notifyAll();
6      }
7  }
```

La función principal de este método es incrementar el valor de la variable *finished* en la línea 2, la cual lleva la cuenta de los *Managers* que han terminado. Cuando el número de estos es igual al número de *Managers* registrados en el sistema (línea 3), el estado de la variable *done* cambia a *true* (línea 4) y se avisa a los procesos en espera que continúen su ejecución (línea 5). La interacción entre *wait()* en el método *waitToFinish()* y *notifyAll()* en el método *managerDone()* funciona como sigue:

1. Después de realizar la llamada a *startSystem()*, el programador debe invocar al método *waitToFinish()* antes de ejecutar otra instrucción. Este método bloquea el flujo del proceso en que el programador controla la ejecución del sistema.

2. Progresivamente, los *Managers* avisan al objeto *Master* el final en la ejecución de sus procesos.

3. Cuando el último *Manager* termina y avisa al *Master*, el método *notifyAll()* es invocado restaurando el flujo en el proceso de control del programador.

La sección 4.8 muestra de forma más clara el comportamiento y utilidad del método *waitToFinish()* en el proceso de control del programador.

En algunas ocasiones, es necesario detener la ejecución de los *Managers*. Esto se logra mediante un método de la clase *Master* llamado *killManagers()* cuya implementación es como sigue:

```
1  public void killManagers(){
2      MasterMessage message = new MasterMessage("Master", null,
           Constants.KILL);
3      Collection<ManagerConnection> mannagers = mannagerTable.
           values();
4      Iterator<ManagerConnection> manIter = mannagers.iterator();
5
6      while(manIter.hasNext()){
7          ManagerConnection man = manIter.next();
8          try{
9              man.sendMessage(message);
10         }
11         catch(ConnectionException e){}
12     }
13 }
```

Para detener la ejecución de los *Managers*, primero se crea un mensaje que contenga la señal *KILL* de la clase *Constants* en la línea 2. Después, se envía a cada *Manager* el mensaje creado si es que esto es posible en las líneas 6 a 12.

La clase *Master* cuenta con un método llamado *processMessage()* encargado de procesar los mensajes recibidos desde los *Managers*. La implementación de este método es la siguiente:

```
1  private void processMessage(MasterMessage message){
2      switch(message.getMessageType()){
3          case DONE:
4              managerDone();
5              break;
6          case DATA:
7              String procId = message.fromProcess();
8              Serializable data = message.getData();
9
10             SyncBuffer<Serializable> buf = procsMessages.get(
                   procId);
11             buf.write(data);
12             break;
13         case EXCEPTION:
14             String error = (String) message.getData();
15             System.out.println(error);
16
17             killManagers();
18             System.exit(1);
19             break;
20     }
21 }
```

El método *processMessage()* recibe como argumento un mensaje *MasterMessage*, y realiza una acción determinada por el tipo del mensaje (la variable *type* de la clase *MasterMessage*). El tipo del mensaje puede ser alguno de los siguientes valores definidos en la clase *Constants*:

- **DONE**. Este tipo de mensaje es enviado por los *Managers* para avisar al objeto *Master* que han terminado la ejecución de su conjunto de procesos. Al recibir un mensaje de este tipo, *Master* ejecuta el método *managerDone()* (línea 4).
- **DATA**. Un mensaje de tipo *DATA* contiene información proveniente de un proceso *RemoteJob* destinado al programador. Al recibir un mensaje de este tipo, el objeto *Master* almacena su atributo *message* en el buffer de mensajes asociado al proceso emisor en la tabla *procsMessages*.
- **EXCEPTION**. Es el tipo de mensaje recibido cuando ocurre un e-rror en algún *Manager*, generalmente en la comunicación. El mensaje contiene el

mensaje de error en forma de un objeto *String*. Ante este mensaje, el objeto *Master* detiene la ejecución de todo el sistema en las líneas 17 y 18.

Para procesar los mensajes del buffer *incomingMessages*, el objeto *Master* cuenta con un thread llamado *MessageProcessor* encargado de leer mensajes del buffer y procesarlos con el método *processMessage()*. Su implementación es la siguiente:

```
1  private class MessageProcessor extends Thread{
2      ...
3      public void run(){
4          while(true){
5              MasterMessage incoming = incomingMessages.read();
6              processMessage(incoming);
7          }
8      }
9  }
```

La clase *MessageProcessor* es una clase interna de la clase *Master*. Debido a ello, el acceso al método *processMessage()* es inmediato.

En principio, el *Master* maneja una tabla de procesos *RemoteJob* con todas las tareas que deben ejecutarse en el sistema. Sin embargo, estos procesos deben ser separados para enviarse a los *Managers* que los ejecutan. Para esto, existe un método en la clase *Master* llamado *calcJobsFor()* que acepta como entrada un objeto *ManagerData* y regresa una tabla Hash con los procesos que debe ejecutar el *Manager* representado por el objeto de entrada. Su implementación está dada por el código siguiente:

```
1  public Hashtable<String,RemoteJob> calcJobsFor(ManagerData dm){
2      LinkedList<String> procslist = dm.getJobsToDo();
3      Iterator<String> listIter = procslist.iterator();
4
5      Hashtable<String,RemoteJob> jobsTab = new Hashtable<String,
           RemoteJob>();
6
7      while(listIter.hasNext()){
8          String pr = listIter.next();
9          RemoteJob job = processTable.get(pr);
10         jobsTab.put(pr, job);
11     }
12     return jobsTab
13 }
```

El algoritmo anterior recorre la lista de cadenas del objeto *dm*, que son los identificadores de los procesos que el *Manager* real debe ejecutar. Por cada uno de

estos identificadores, se obtiene de la tabla *ProcessTable* el proceso real que tiene dicho identificador (línea 9) y se agrega a una tabla (línea 10) que es devuelta al final del método.

Cuando se crea una instancia de la clase *Master*, se le asigna el archivo de configuración que contiene los datos de ejecución del sistema. El siguiente paso es darle un conjunto de procesos de tipo *RemoteJob* mediante el método *addRemoteJob()* de la clase *Master*. Finalmente, el método *startSystem()* ejecuta la operación de todo el sistema. Su implementación es como sigue:

```
1  public void startSystem(){
2      try{
3          processConfFile();
4
5          Collection<ManagerData> dataMannager = mannagerDataTable.
               values();
6          Iterator<ManagerData> mdIterator = dataMannager.iterator
               ();
7
8          while(mdIterator.hasNext()){
9              ManagerData dm = mdIterator.next();
10
11             ManagerConnection mc = new ManagerConnection(dm,
                   timeout, incomingMessages);
12             managerTable.put(dm.getId(), mc);
13
14             mc.stablishConnection();
15
16             MasterMessage idMessage = new MasterMessage("Master",
                   dm.getId(),DATA);
17             mc.sendMessage(idMessage);
18
19             MasterMessage consMes = new MasterMessage("Master",
                   dm.getConnectionsTable(),DATA);
20             mc.sendMessage(consMes);
21
22             MasterMessage mapMes = new MasterMessage("Master",dm.
                   getProcessMapping(),DATA);
23             mc.sendMessage(mapMes);
24
25             Hashtable<String,RemoteJob> jobsTab = calcJobsFor(dm)
                   ;
26
27             MasterMessage jobsMes = new MasterMessage("Master",
                   jobsTab,DATA);
28             mc.sendMessage(jobsMes);
29         }
30
```

```
31          communicateMannagers();
32
33          Collection<ManagerConnection> consMannager =
                mannagerTable.values();
34          Iterator<ManagerConnection> conIterator = consMannager.
                iterator();
35
36          while(conIterator.hasNext()){
37              ManagerConnection con = conIterator.next();
38              con.startMasterReceiver();
39          }
40
41          MessageProcessor mp = new MessageProcessor();
42          mp.start();
43
44          MasterMessage initSig = new MasterMessage("Master", null,
                START_PROCESSES);
45          spreadMessage(initSig);
46      }
47      catch(ConFileException e){
48          String str = "Bad␣configuration␣file␣" + "\n" +this.
                configFile + e.getMessage();
49          System.out.println(str);
50          System.exit(1);
51      }
52      catch(ConnectionException e){
53          killManagers();
54          String str = "Unable␣to␣initialize␣system" + "\n" + e.
                getMessage();
55          System.out.println(str);
56          System.exit(1);
57      }
58  }
```

Al iniciar el sistema, se procesa el archivo de configuración en la línea 3 con la instrucción *processConfFile()*. Al terminar la ejecución de este método, la tabla *dataManager* queda lista con la configuración que cada *Manager* debe tener. Así, se toma cada uno de los elementos *ManagerData* de la tabla y se construye un objeto *ManagerConnection* utilizando la información del *ManagerData* y pasándole además el buffer *incomingMessages* de mensajes de llegada en la línea 11. Esta *ManagerConnection* es agregada a la tabla correspondiente *ManagerTable* en la línea 12.

Creada la conexión y mediante ésta, se establece la comunicación con el *Manager* real en la línea 14. El identificador del *Manager* es enviado en las líneas 16 y 17. Entre las líneas 19 a 23, se crean y envían al *Manager* las conexiones que de-

be establecer con otros *Managers*, así como la tabla *pMapping* que le corresponde.

Como último paso para el *Manager* en turno, se determina el conjunto de procesos *RemoteJob* que debe ejecutar en la línea 25 y se envía en las líneas 27 y 28.

El procedimiento anterior es repetido para cada representante *ManagerData* en la tabla *dataManager*.

La línea 31 ejecuta el método *communicateManagers()* que organiza a los *Managers* para establecer conexiones entre ellos. Hasta este punto, los *Managers* reales tienen toda la información para iniciar su operación. Sin embargo, algunos procedimientos son necesarios para que el objeto *Master* realice su trabajo correctamente: Entre las líneas 33 y 39, se ponen en marcha los threads de los objetos *ManagerConnection* que reciben mensajes de los *Managers* y los depositan en el buffer *incomingMessages*.

El thread *MessageProcessor* encargado de leer y procesar mensajes de los *Managers* almacenados en el buffer *incomingMessages* es iniciado y puesto en ejecución en las líneas 41 y 42.

Finalmente, se envía a todos los *Managers* la señal que les indica iniciar la ejecución de sus procesos en las líneas 44 y 45 mediante el método *spreadMessage()*. Este método envía el mismo mensaje recibido como argumento a todos los *Managers* registrados en el sistema.

Al iniciar la ejecución del sistema, dos excepciones pueden ser lanzadas:

- **ConfFileException**. Que es lanzada si se detecta un error en el archivo de configuración, y en cuyo caso, se imprime el error en la pantalla y se termina la ejecución del sistema en las líneas 47-51.
- **ConnectionException**. Que es lanzada si ocurre un error en la comunicación con los *Managers* y en cuyo caso se realizan tres acciones: Se intenta terminar la ejecución de los *Managers* en la línea 53, se imprime el mensaje de error en la pantalla en la línea 55 y se termina la ejecución del objeto *Master* en la línea 56.

4.7.3. Comunicación con el usuario

Además de los métodos que ya hemos revisado y que involucran la interacción con el programador, existen otros métodos que permiten a este controlar la ejecución del sistema.

El método *isMessageFrom()* de la clase *Master* recibe un identificador de proceso como argumento y regresa el valor *true* si existe por lo menos un elemento en el buffer de mensajes dirigidos al usuario, asociados a dicho proceso. Su implementación es como sigue:

```
1  public boolean isMessageFrom(String id) throws
       IDNotFoundException{
2       if(!procsMessages.containsKey(id)){
3           throw new IDNotFoundException("ID␣" + id + "␣not␣found␣in
                ␣process␣table");
4       }
5
6       SyncBuffer<Serializable> buf = procsMessages.get(id);
7       return (buf.size() != 0);
8  }
```

Este método puede lanzar una *IDNotFoundException* si el identificador proporcionado no existe en la tabla de procesos registrados en el sistema.

Para obtener los mensajes recibidos desde determinado proceso, existe el método *messageFrom()* en la clase *Master*, cuya implementación es la siguiente:

```
1  public Serializable messageFrom(String id) throws
       IDNotFoundException{
2       if(!procsMessages.containsKey(id)){
3           throw new IDNotFoundException("ID␣" + id + "␣not␣found␣in
                ␣process␣table");
4       }
5       SyncBuffer<Serializable> buf = procsMessages.get(id);
6       return buf.read();
7  }
```

Este método acepta como argumento el identificador del proceso del cual se quieren recuperar mensajes y regresa el primer elemento del buffer asociado a dicho proceso. Puede lanzar una *IDNotFoundException* si el identificador proporcionado no se encuentra en la tabla de procesos registrados. El método *waitToFinish()* se bloquea hasta que exista un mensaje en el buffer, por lo cual es recomendable ejecutar el método *isMessageFrom()* antes de intentar recuperar un mensaje, para asegurarse que éste existe.

Finalmente, existe el método *shutdown()* en la clase *Master* que detiene la ejecución de todos los *Managers*, así como del mismo objeto *Master*. Su implementación se muestra en el siguiente segmento de código:

```
1  public void shutdown(){
2      killManagers();
3      System.exit(0);
4  }
```

En resumen, el usuario puede comunicarse con el sistema por medio del objeto *Master* en los siguientes puntos:

- Antes de iniciar la ejecución del sistema al proporcionarle los procesos que deben ser ejecutados mediante el método *addRemoteJob()*.
- Al iniciar la ejecución del sistema con el método *startSystem()*.
- Al esperar a que los *Managers* terminen su ejecución con el método *waitToFinish()*.
- Al recuperar datos provenientes de los procesos *RemoteJob* mediante los métodos *isMessageFrom()* y *messageFrom()*.
- Al finalizar la ejecución del sistema mediante la instrucción *shutdown()*.

4.8. Puesta en marcha

Para ilustrar al lector la forma en la que opera J-MIPS, damos un ejemplo de un sistema distribuido muy sencillo. Consideremos cuatro procesos con identificadores $P_1 ... P_4$ conectados en forma de *Pipeline* como en la sección 1.5.1. El trabajo de cada proceso es recibir una cadena de caracteres del proceso anterior, concatenarla con su identificador y enviar la concatenación al siguiente proceso. Por supuesto, el primer proceso en el *Pipeline* no recibe nada e inicia su operación únicamente enviando su identificador al siguiente proceso. El último proceso envía la concatenación de cadenas al objeto *Master* de donde el usuario puede revisar el resultado. La figura 4.1 muestra la operación del *Pipeline* descrito, así como la cadena enviada por cada filtro. Al final, el objeto *Master* recibe del proceso P_4 la cadena *P1P2P3P4*.

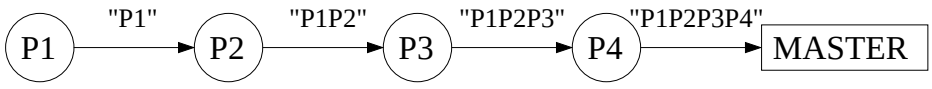

Figura 4.1: Operación del *Pipeline*

Además, supongamos que estos procesos deben ejecutarse en un cluster de 4 computadoras con las siguientes direcciones IP, a las cuales asociamos un identificador:

Direccion IP	Identificador
192.168.1.101	M101
192.168.1.102	M102
192.168.1.103	M103
192.168.1.104	M104

Asumimos que podemos utilizar el puerto 3305 de todas las computadoras del cluster para conectar nuestros procesos. Queremos utilizar la computadora M101 como aquélla en la que el usuario controle la operación de todo el sistema; es decir donde resida la computadora virtual *Master*. Además, queremos que la computadora M102 ejecute el proceso P_1, la computadora M103 debe ejecutar al proceso P_2 y la computadora M104 a los procesos P_3 y P_4. Para esto, utilizamos un *Manager* para cada computadora donde se ejecute un proceso P_i. Así, utilizamos tres *Managers*, uno en la computadora M102, otro en la computadora M103 y uno más en la computadora M104. Utilizamos los identificadores de la tabla anterior para identificar también a los *Managers*.

Los siguientes componentes son necesarios para operar el sistema:

- La implementación de la clase *RemoteJob* que defina la operación de cada proceso P_i.
- La clase Main de los *Managers*
- La clase Main del objeto *Master*
- El archivo de configuración del sistema.

El trabajo de los procesos está representado por la clase *Job* definida como sigue:

```
1  public class Job extends RemoteJob{
2      int id;
3
4      public Job(int id){
5          super("P" +id);
6          this.id=id;
7      }
8
9      public void run(){
10         if(id == 1){
11             sendMessage("P2", getProcessId());
12         }
13         else{
14             String cadena = (String)receiveMessage("P" + (id-1));
15             String data = cadena + getProcessId();
```

```
16              if(id == 4){
17                  saytoMaster(data);
18              }
19              else{
20                  sendMessage("P"+(id+1), data);
21              }
22          }
23          finished();
24      }
25  }
```

La clase *Job* hereda de la clase *RemoteJob*. En el constructor, el enunciado *super("P"+id)* asigna al proceso creado el identificador *Pi*, donde *i* es un número entero que identifica a un proceso de todos los demás. En el método *run()*, se considera el trabajo de los 4 procesos, diferenciándolo mediante el índice *id* de cada proceso. Así, el trabajo del proceso *P1* se encuentra en la línea 11 y consiste únicamente en enviar su identificador *getProcessId()* al proceso *P2*. Para todos los otros procesos, su primer tarea es recibir una cadena de caracteres del proceso que les precede en la línea 14. Después, en la línea 15, se construye otra cadena de caracteres concatenando la cadena recibida con el identificador del proceso que la recibe. El último proceso del *Pipeline* es enviar la cadena construida al objeto *Master* en la línea 17, mientras que el trabajo de los demás, es enviarla al siguiente proceso. Finalmente, la instrucción *finished()* de la línea 23 es necesaria en el proceso para avisar al *Manager* que lo ejecuta que el trabajo ha concluido.

La clase Main de cada *Manager* debe simplemente esperar conexión por parte del objeto *Master* mediante el método *connectToMaster()*. Su implementación es como sigue:

```
1   public class ManagerMain {
2
3       public static void main(String [] argv){
4           Manager m = new Manager(3305,10000);
5           try{
6               m.connectToMaster();
7           }
8           catch(ConnectionException e){
9               e.printStackTrace();
10          }
11      }
12  }
```

En principio, se crea un *Manager* que se conecta a través del puerto 3305 y que tiene un tiempo de espera de 10 segundos en la línea 4. La instrucción *connectToMaster()* es invocada en la línea 6 dentro de un bloque *try-catch*.

La clase Main del objeto *Master* requiere iniciar la ejecución del sistema, esperar a que termine, interactuar con el objeto *Master* y terminar la ejecución del sistema. La implementación de esta clase es como sigue:

```
1  public class MasterMain {
2
3      public static void main(String [] argv){
4          Master master = new Master("Config.txt");
5          try{
6              for(int i = 1; i <= 4; i++){
7                  master.addRemoteJob(new Job(i));
8              }
9
10             master.startSystem();
11             master.waitToFinish();
12
13             while(master.isMessageFrom("p4")){
14                 System.out.println(master.messageFrom("p4"));
15             }
16         }
17         catch(IDNotFoundException e){
18             e.printStackTrace();
19         }
20         catch(DuplicatedElementException e){
21             e.printStackTrace();
22         }
23         master.shutdown();
24     }
25 }
```

En la línea 4 se crea un objeto *Master* que recibe la cadena *Config.txt* como nombre del archivo de configuración. Entre las líneas 6 y 8 se agregan al objeto *Master* 4 procesos de tipo *Job* con identificadores *P1,P2,P3* y *P4*. La llamada *master.startSystem()* de la línea 10 inicia la operación del sistema distribuido. Antes de ver mensajes provenientes de los procesos, se debe esperar a que la ejecución de los procesos termine, lo cual se hace mediante la llamada *master.waitToFinish()*. Una vez que los procesos han finalizado, es seguro revisar los mensajes provenientes del proceso *P4*, lo cual se hace entre las líneas 13 y 15. La línea 7 puede lanzar una excepción de tipo *DuplicatedElementException* y las líneas 13 y 14 excepciones de tipo *IDNotFoundException*. Estas excepciones son capturadas en las líneas 17 y 20. Finalmente, el sistema debe ser terminado mediante la instrucción *master.shutdown()* de la línea 23.

El último componente que se requiere es el archivo de configuración *Config.txt*, el cual queda definido de la siguiente forma:

```
1   #Managers
2   Manager = (192.168.1.102,3305,M102)
3   Manager = (192.168.1.103,3305,M103)
4   Manager = (192.168.1.104,3305,M104)
5
6   #Procesos
7   Process = (P1,M102,[P2])
8   Process = (P2,M103,[P1,P3])
9   Process = (P3,M104,[P2,P4])
10  Process = (P4,M104,[P3])
```

En las líneas 2, 3 y 4 se definen los *Managers* M102, M103 y M104 tal como lo especificamos al principio de esta sección. Las líneas 7 a 10 especifican el mapeo de los procesos. Por ejemplo, el proceso P3 definido en la línea 9 se debe ejecutar en el *Manager* M104 y está conectado a los procesos P2 y P4.

Al ejecutar el sistema, debemos mencionar nuevamente que los *Managers* deben ser puestos en marcha **antes** que el objeto *Master*.

4.9. Consideraciones finales

El usuario de J-MIPS debe tener en cuenta los siguientes puntos para mejorar su desempeño:

- Para ejecutar varios *Managers* en una misma computadora física, se requiere la implementación de threads adicionales que ejecuten un *Manager* cada uno, así como la especificación de un puerto diferente para cada *Manager*.
- Salvo para cuestiones de pruebas y depuración de un programa paralelo, no es recomendable utilizar más de un *Manager* en la misma computadora física. Esto es debido a que cada *Manager* requiere tiempo de procesamiento para realizar su función. En su lugar, recomendamos ejecutar varios procesos en un solo *Manager* por computadora física.
- En un sistema paralelo distribuido, las comunicaciones entre procesos (particularmente los que se ejecutan en distintas computadoras) representan la parte más lenta del sistema. Debido a ello, recomendamos que el programador implemente procesos de granularidad gruesa respecto a la comunicación. Mientras más se comuniquen los procesos, más tiempo requiere la ejecución del sistema.
- Aunque los procesos son enviados a los *Managers* desde el objeto *Master*, cada *Manager* debe saber qué tipo de procesos va a ejecutar. Esto es, cada

computadora física del sistema debe tener acceso a los componentes del programa paralelo, es decir, cada computadora debe poseer una copia del programa escrito por el usuario.

- Es recomendable que el objeto *Master* se ejecute en una computadora física dedicada exclusivamente a esta tarea. Esto es debido a que en el objeto *Master*, se ejecutan varios threads para supervisar las conexiones con cada *Manager*, lo cual puede incrementar considerablemente el tiempo de ejecución de otros *Managers* que se ejecuten en la misma computadora.

- Es imprescindible que el programador recuerde escribir la sentencia *finished()* al final de la ejecución de sus procesos. De otra forma, los *Managers* donde se ejecuten procesos que no tengan esta sentencia no se enteran que el trabajo de los procesos termina, por lo cual, tampoco avisan al objeto *Master* el final de la ejecución de sus procesos, lo cual provoca que el método *waitToFinish()* del objeto *Master* se bloquee indefinidamente.

- Se recomienda cierto tiempo de espera entre la puesta en marcha de los *Managers* y la ejecución del método *startSystem()* del objeto *Master*. Esto es para asegurar que cada *Manager* tiene suficiente tiempo para preparar un Socket de conexión que espere al objeto *Master*. Si esta última intenta conectarse con un *Manager* que no está preparado para recibir la conexión, *Master* lanza una excepción y detiene su ejecución sin esperar a los *Managers*.

- Los procesos no pueden mandarse mensajes a sí mismos. Una llamada de tipo *sendMessage("P",data)* o *receiveMessage("P")* hecha desde el proceso *P*, producirá una excepción de tipo *NullPointerException*.

4.10. Resumen del capítulo

En este capítulo, revisamos la implementación de la biblioteca J-MIPS en Java. Esto involucra la revisión del código fuente de los elementos principales que permiten la ejecución de la biblioteca según el diseño del capítulo 4, así como una explicación de las instrucciones de dicho código. El capítulo continua con un ejemplo de un programa paralelo distribuido sencillo utilizando J-MIPS, el cual sirve para ilustrar los pasos necesarios para utilizar correctamente la biblioteca. Finalmente, se provee al lector con algunas consideraciones especiales de carácter técnico que permiten mejorar el desempeño de un sistema distribuido que utiliza J-MIPS.

Capítulo 5

Ejemplos de aplicación

En este capítulo describimos la implementación de dos problemas comunes en computación paralela distribuida mediante J-MIPS. Dicha implementación nos ayuda a mostrar las capacidades y flexibilidad de la biblioteca que construimos en capítulos anteriores.

Los programas que mostramos son descritos de la siguiente forma:

- **Definición del problema**. Donde se introduce al lector en el problema que hay que resolver.
- **Algoritmo secuencial**. Para comprender el algoritmo paralelo distribuido, así como la implementación de este mediante J-MIPS, es importante que conozca el algoritmo secuencial que resuelve el problema tratado, pues el algoritmo paralelo distribuido surge a partir del análisis del algoritmo secuencial.
- **Paralelismo potencial**. En esta sección se analiza el algoritmo secuencial y se consideran los elementos de este que pueden ser realizados en paralelo.
- **Algoritmo paralelo distribuido**. En esta sección se construye un algoritmo paralelo que resuelve el problema planteado.
- **Implementación con J-MIPS**. Con los elementos anteriores, esta sección está dedicada a la construcción de un programa paralelo distribuido mediante la biblioteca propuesta en este trabajo, el cuál resuelve el problema planteado. Se detallan los elementos de configuración, arranque, ejecución y finalización del sistema.

Los ejemplos expuestos en este capítulo fueron ejecutados en un cluster de 5 computadoras con las siguientes características:

- **Computadora 1.** Procesador AMD Athlon 64x2 3000+, memoria RAM de 4 GB, disco duro 1000 GB, sistema operativo Ubuntu 11.04, IP 192.168.1.100.
- **Computadora 2.** Procesador Intel Pentium 4 2.4 GHz, memoria RAM de 2 GB, disco duro 120 GB, sistema operativo Ubuntu 11.04, IP 192.168.1.101.
- **Computadora 3.** Procesador Intel Centrino duo 1.6 GHz, memoria RAM de 1 GB, disco duro de 100 GB, sistema operativo Ubuntu 11.08, IP 192.168.1.102.
- **Computadora 4.** Procesador Intel dual core 3.2 GHx, memoria RAM de 2GB, disco duro de 160 GB, sistema operativo Ubuntu 11.04, IP 192.168.1.103.
- **Computadora 5.** Procesador AMD Sempron 145 2.8 GHz, memoria RAM de 4 GB, disco duro de 500 GB, sistema operativo Ubuntu 11.08, IP 192.168.1.104.

Conectadas en una LAN con topología de estrella a través de un router inalámbrico 2WIRE 1701HG.

5.1. Ejemplo 1: Caminos mínimos

5.1.1. Definición del problema

Dada una gráfica $G(V, E)$ que consiste de N vértices y K aristas, en la cual toda arista e tiene un peso entero positivo que representa la distancia entre los vértices de sus extremos (figura 5.1), el objetivo es encontrar la trayectoria más corta entre un vértice inicial y cada uno de los otros vértices; donde la longitud de una trayectoria P, $d(P)$ está definida como la suma de las distancias de las aristas que recorre [9].

El algoritmo del camino mínimo con una única fuente (o SSP por sus siglas en ingles Single Source Shortest Path) fue propuesto por Dijkstra en 1956 y toma tiempo de ejecución $O(N^2)$. En cada iteración el algoritmo selecciona el vértice con la distancia más corta desde el vértice origen y lo etiqueta para recordar que su distancia mínima ya ha sido establecida. En el siguiente ciclo, todos los vértices cuya distancia mínima aun no es conocida (que en adelante llamaremos *desconocidos*) son examinados para ver si hay una trayectoria más corta a ellos desde el vértice etiquetado más recientemente [9].

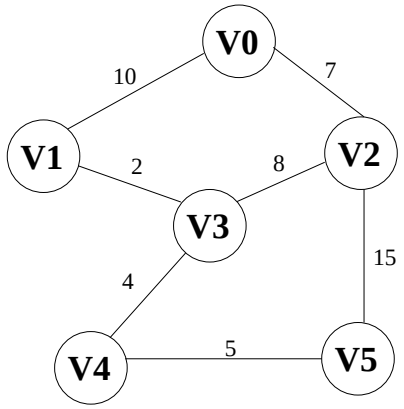

Figura 5.1: Gráfica con pesos positivos

5.1.2. Algoritmo secuencial

La gráfica puede ser representada por una matriz de adyacencias *g[][]*, en la cual, los elementos *g[i][j]* representan los pesos de las aristas entre los vértices. Se requieren dos estructuras de datos más: Un arreglo booleano *known[]*, para determinar aquellos vértices cuya distancia mínima ha sido establecida y un arreglo *dist[]* para recordar la distancia mínima establecida más recientemente entre el vértice origen y los de más [9].

Se requiere una función que llamaremos *minV*, que toma dos vértices como argumentos y regresa aquél con la distancia más corta que aun no se ha determinado como mínima. Si uno de estos vértice aparece marcado como conocido en el arreglo *known[]*, se regresa el otro. Se asume por la construcción del algoritmo que *minV* nunca se llama con dos vértices conocidos. Se asume también, que en la gráfica *g*, si no hay una arista con extremos (i,j), entonces *g[i][j] = infinito* [9].

El algoritmo es como sigue[9]:

Código 5.1: Algoritmo secuencial del problema SSP

```
1 /*Todos los vertices excepto el origen son desconocidos*/
2 for(int i = 1 ; i < N ; i++){
3   known[i] = false;
4 }
5
6 /*Solo el vertice 0 es conocido*/
7 known[0] = true;
8
9 /*Distancia inicial del vertice origen a todos los de mas*/
```

```
10  for(int i = 0 ; i < N ; i++){
11    dist[i] = g[0][i];
12  }
13
14  int lastknown = 0;  //El ultimo conocido es el vertice 0
15  int knowncount = 1; //Conocemos 1 vertice. Faltan N - 1
16
17  while(knowncount < N){
18    int minvertex = 0;
19    for(int i = 1 ; i < N ; i++){
20      /*Buscamos la distancia mas corta via el ultimo vertice
            etiquetado*/
21      if( ! known[i] ){
22        dist[i]= min(dist[i],dist[lastknown]+g[lastknown][i]);
23      }
24      minvertex = minV(minvertex , i);
25    }
26    /*Seleccionamos el proximo vertice conocido*/
27    lastknown = minvertex;
28    known[lastknown] = true;
29    knowncount++;
30  }
```

La figura 5.2 representa la ejecución del algoritmo de Dijkstra para un ejemplo. Los círculos representan los vértices, las líneas entre los círculos representan las aristas y los números a los lados de las aristas representan la distancia entre los vértices. Los vértices blancos son vértices desconocidos, los vértices con sombra obscura son aquéllos cuya distancia mínima ya ha sido establecida y los vértices con sombra clara son vértices desconocidos con distancia mínima que actualmente están siendo evaluados por el algoritmo.

5.1.3. Paralelismo Potencial

En cada ciclo del código 5.1, la distancia actual (*dist[i]*) a un vértice i, debe ser comparada con la distancia hacia ese mismo vértice pasando por el último vértice conocido (*dist[lastknown] + g[lastknown][i]*), y la mínima entre esas distancias debe ser recordada como la nueva distancia mínima. Estos cálculos dependen solamente del arreglo bidimencional *g[i][j]*. Así, la distancia mínima para cada vértice **puede ser calculada en paralelo**. Una vez calculada, la distancia mínima a todos los vértices puede ser marcada. Si hay N procesadores disponibles, el algoritmo tiene un tiempo de ejecución de $O(Nlog_2N)$. Aun se requieren $N-1$ ciclos para calcular la mínima distancia para todos los vértices. Sin embargo, cada ciclo requiere solo un paso para actualizar el mínimo para cada vértice y $O(log_2N)$ pasos para calcular el vértice mínimo [9].

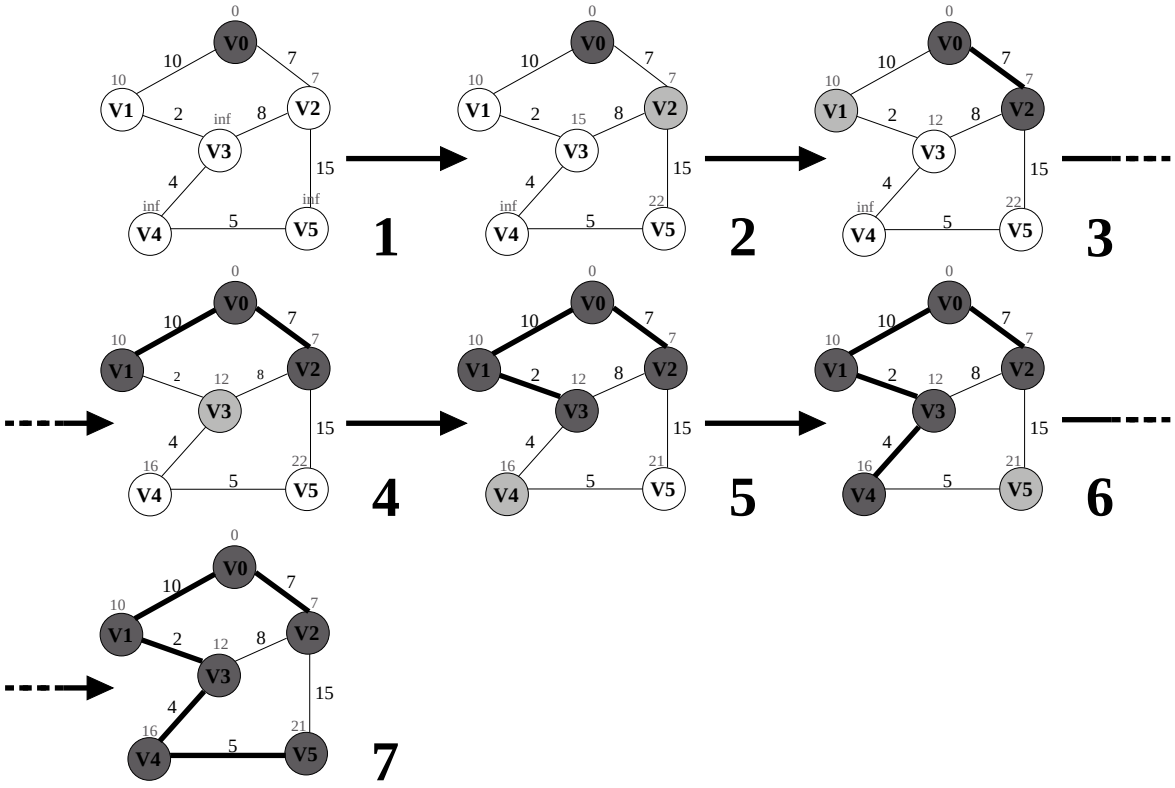

Figura 5.2: Ejecución del algoritmo de Dijkstra sobre la figura 5.1

5.1.4. Algoritmo paralelo distribuido

Para poder resolver este problema en un sistema de cómputo de memoria distribuida, iniciamos suponiendo que el sistema dispone de tantos procesadores como sean necesarios [1]. Mas precisamente, para N vértices en la gráfica, suponemos que el sistema de cómputo distribuido dispone de N procesadores. Un procesador para cada vértice[9].

Dos subproblemas deben ser abordados[9]:

1. Se debe determinar la topología de la red
2. Es necesario determinar la información que debe permanecer en cada procesador de la red, asi como la información que debe ser transmitida entre procesadores.

[1]En esta sección y por el resto de este capítulo, nos referimos a "procesador" como uno de los nodos del sistema distribuido

Debemos considerar que la topología de la red de tal forma que el número de pasos que se requieren para distribuir un mensaje entre los procesadores sea mínimo [9].

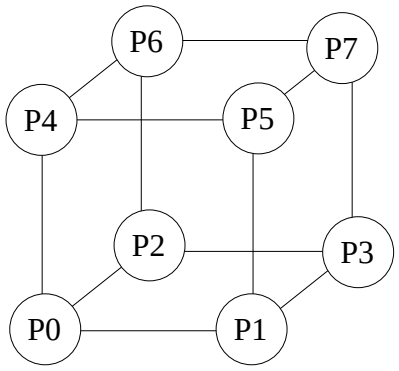

Figura 5.3: Hiper-cubo de 8 nodos

Utilizando una arquitectura de hipercubo como la de la figura 5.3, tanto la comunicación como el cálculo del vértice mínimo pueden realizarse en un tiempo $O(log_2 N)$. Con este esquema, cada procesador calcula su distancia mínima al vértice origen; después, los procesadores superiores (P4,...,P7) envían esta distancia hacia los procesadores inferiores (P0,...,P3) en un solo paso. A su vez, estos procesadores inferiores toman la mínima entre la distancia más corta local y aquélla enviada por los procesadores superiores. La mitad de los procesadores inferiores (P2 y P3) envían su distancia hacia los procesadores P0 y P1 en un segundo paso; el procesador P1 toma la distancia más corta entre su propia distancia mínima y aquélla enviada por el procesador P3, para luego enviarla al procesador P0 en el tercer paso. El procesador P0, toma la mínima distancia global comparando las distancias recibidas de los procesadores P1,P2,P4 y la suya propia. La comunicación y selección del vértice con distancia más corta puede realizarse en $O(log_2 N)$ pasos. Ya que este proceso debe realizarse para cada vértice, el rendimiento general del algoritmo es $O(N log_2 N)$ [9].

La arquitectura de hipercubo que se propone puede ser vista como el patrón de capas paralelas descrito en la sección 1.5.2 del capítulo 2, como lo muestra la figura 5.4.

El paso final en la implementación del algoritmo con N procesadores es determinar la información que debe ser transmitida entre ellos. El procesador controlador P0 debe calcular cuál de entre dos vértices tiene la distancia más corta aun no conocida. Para hacerlo, debe tener disponible la información de aquellos vértices

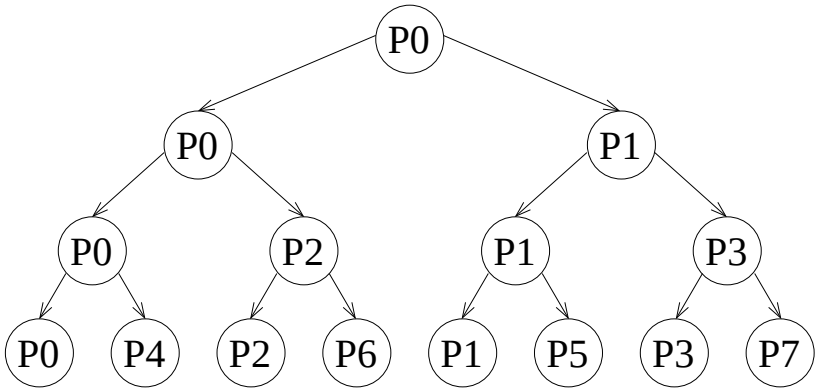

Figura 5.4: Representación del hipercubo de la figura 5.3 como el patrón de capas paralelas.

que ya han sido marcados como conocidos (el arreglo *known[]*), y la distancia e *id* de los vértices que compara [9].

Los procesadores esclavos, deben comparar su propia distancia con la distancia entre el ultimo vértice conocido y ellos mismos. Así, la gráfica *g[][]* y la información del último vértice conocido (*id* y distancia) deben ser accesibles por todos los procesadores. Además, algunos procesadores esclavos calculan el mínimo entre dos vértices, así que necesitan saber cuales de ellos son conocidos [9].

Los datos básicos que son enviados entre procesadores son el *id* del vértice y su distancia más reciente. Estos datos son usados para calcular el vértice con distancia mínima y para anunciar cuáles vértices son marcados como conocidos [9]. Así, un mensaje puede ser representado por medio de una estructura como la siguiente:

Código 5.2: Información de un vértice en una clase *Vertex* en java

```
1   public class Vertex{
2      private int id;
3      private int distance;
4
5      public Vertex(int id, int distance){
6        this.id = id;
7        this.distance = distance;
8      }
9
10     public int getId(){
11       return this.id;
12     }
```

```
13
14      public int getDistance(){
15        return this.distance;
16      }
17   }
```

Ya que todos los procesadores deben saber cuáles vértices son conocidos, cada procesador debe tener su propio arreglo *known[]* para recordar el estado de los vértices. Una vez que un procesador determine a un vértice como conocido, este procesador debe mandar un mensaje a todos los otros para avisar que el estado del vértice ha cambiado. Así, cada procesador debe almacenar y actualizar localmente el arreglo *known[]*. Asimismo, la gráfica *g[][]*, que no cambia durante la ejecución del algoritmo, debe ser distribuida a todos los procesadores y almacenada localmente antes del inicio de la ejecución del algoritmo de búsqueda [9].

Con estos cambios, la función *minV*, ya no tiene acceso al arreglo *dist[]*, para buscar las distancias de los vértices que deben ser comparados. Los parámetros deben cambiar de tal manera que las distancias de los vértices sean pasadas como argumentos, así como los identificadores de dichos vértices [9]. Gracias a la programación orientada a objetos y a la clase propuesta en el código 5.2, podemos cambiar los argumentos de la función *minV* para recibir dos objetos de tipo Vertex en vez de los identificadores de estos vértices. Finalmente, se requiere de una variable *distance* local, que representa la distancia desde el origen hasta los otros vértices.

Usando el esquema y las especificaciones anteriores, el algoritmo paralelo para resolver el problema SSP en un sistema de cómputo de memoria distribuida se divide en dos partes: El algoritmo para el procesador controlador P0 y el algoritmo para los otros procesadores. Esto es necesario, pues como mencionamos, la tarea de P0 consiste principalmente en comparar los resultados de sus vecinos e informar el resultado de la comparación [9].

El código para el procesador P0 que calcula el vértice con la distancia mínima global y determina al último vértice conocido es como sigue[9]:

Código 5.3: Código para el procesador P0

```
1   public void solveforP0(){
2      int i = 1;
3      while(i < N){
4        /*Recibimos la informacion de los vertices P1,P2,P4 y
             tomamos al minimo*/
5        Vertex minVertex = 0;
6        Vertex received;
```

```
 7        received = receibeMessage(Z);
 8        minVertex = minV(minVertex,received);
 9        received = receiveMessage(Y);
10        minVertex = minV(minVertex,received);
11        received = receiveMessage(X);
12        minVertex = minV(minVertex,received);
13
14        /*El ultimo vertice conocido es determinado y su estado es
              actualizado*/
15        known[minVertex.getId()] = true;
16        Vertex lastKnown = minVertex;
17
18        /*P1,P2 y P4 deben saber sobre el ultimo conocido*/
19        sendMessage(X, lastKnown);
20        sendMessage(Y, lastKnown);
21        sendMessage(Z, lastKnown);
22        i++;
23      }
24   }
```

El código para los procesadores Pk, donde $1 \leq k < N$ es como sigue[9]:

Código 5.4: Código para los procesadores distintos de P0

```
1  public void solveforK(){
2    /*Los procesadores restantes*/
3    int i = 1;
4
5    while( i < N){
6      Vertex localMinVertex = k;
7      Vertex received = null;
8      Vertex lastKnown = null;
9
10     /*Encontramos el vertice con la distancia minima desconocida
           */
11     /*procs P1 ... P3 reciben y calculan el minimo*/
12     if(k < 4){
13       received = receiveMessage(Z);
14       localMinVertex = minV(localMinVertex,received);
15     }
16     else{
17       sendMessage(Z,localMinVertex);
18     }
19
20     if(k < 4){
21       if(k == 1){
22         received = receiveMessage(Y);
23         localMinVertex = minV(localMinVertex,received);
24       }
```

```
25          else{
26            sendMessage(Y,localMinVertex);
27          }
28        }
29
30        if(k == 1){
31          sendMessage(X,localMinVertex);
32        }
33
34        /*Se recibe el minimo conocido del procesador 0 y se
              distribuye a todos los
35        demas procesadores*/
36        if(id == 1){
37          lastKnown = receiveMessage(X);
38          known[lastKnown.getId()] = true;
39          sendMessage(Y, lastKnown);
40          sendMessage(Z, lastKnown);
41        }
42        else if (id < 4){
43          lastKnown = receiveMessage(Y);
44          known[lastKnown.getId()] = true;
45          sendMessage(Z, lastKnown);
46        }
47        else{
48          lastKnown = receiveMessage(Z);
49          known[lastKnown.getId()] = true;
50        }
51
52        distance = min(distance, lastKnown.getDistance() +
53                       g[lastKnown.getId()][k]);
54        i++;
55      }
56    }
```

La sincronización se lleva a cabo a través de las conexiones entre procesadores. El método *receiveMessage()* se bloquea hasta que el correspondiente método *sendMessage()* es invocado desde otro procesador [2]. Así, el procesador P3 no puede calcular el vértice con distancia mínima entre sí mismo y el procesador P7 hasta que este último envíe su distancia. Una vez calculado el mínimo, éste es enviado al procesador P1, el cual a su vez, se bloquea hasta recibir el mensaje que contiene la distancia necesaria para calcular el mínimo entre los procesadores P3 y P1 [9].

[2]El algoritmo para memoria distribuida no forma parte del código ejecutable implementado en este trabajo. Este tema se aborda en la siguiente sección.

Casos donde $P > N$

Hasta ahora hemos trabajado suponiendo que el número de procesadores N es equivalente al número de vértices P. Más aun, hemos trabajado sobre un caso muy general en el que el número de vértices y por tanto procesadores es exactamente 8. La extensión de este caso particular al caso donde $P > N$ es en realidad simple. Cada procesador se encarga de calcular la distancia para N/P vértices. En cada iteración, cada procesador debe escoger el vértice con distancia mínima de entre sus vértices. Cuando el vértice mínimo local sea elegido, la distancia de todos los vértices desconocidos debe ser actualizada. Los métodos implementados *minV()* son todo lo que se necesita para tal tarea [9].

Hemos decidido dejar fijo el número de procesadores haciendo $N = 8$. Esto es para mantener el proceso de comunicación entre los procesadores en un tiempo $O(log_2 N)$. Este tiempo puede mantenerse si $N = 2^k$ con $k \in \mathbb{N}$ implementando estructuras de red virtual en hipercubo de k dimensiones.

5.1.5. Implementación con J-MIPS

Como hemos visto, para implementar una aplicación paralela distribuida en J-MIPS, debemos especificar los siguientes componentes (sección 4.8):

- Los procesos que deben ejecutarse.
- La clase Main de los *Managers*.
- La clase Main del objeto *Master*.
- El archivo de configuración.

En general, el trabajo consiste en especificar la tarea de los procesos. La clase Main de los *Managers* solo debe crearlos y decirles que esperen conexión desde el objeto *Master*. La clase Main de la computadora virtual *Master* debe crear los procesos, agregarlos al *Master* y evaluar la información proveniente de ellos.

Los procesos

En la aplicación que construimos, los procesos tienen dos tareas:

- Distribuir la gráfica a través de la red y guardar una copia de ella en un archivo de texto para uso futuro.
- Encontrar mediante el algoritmo visto en la sección pasada el camino más corto entre dos vértices dados por el programador.

Al distribuir la gráfica a través de la red, cada archivo la guarda en un archivo llamado *Grafica + idDelProceso + .dat*. Es decir, cada proceso guarda su propio archivo. Cuando los procesos son instruidos para encontrar el camino más corto entre dos vértices, su primer tarea es cargar la gráfica desde el archivo que le corresponde.

Para controlar el comportamiento de los procesos $P_0 ... P_7$, utilizamos otro proceso llamado *Constructor*, conectado a P_0, el cual indica a este último la tarea que debe llevar a cabo. P_0 distribuye al resto de los nodos la operación que deben llevar a cabo.

El programa que construimos es bastante extenso, por lo cual, omitimos la tarea de distribuir y guardar la gráfica y nos enfocamos en describir la búsqueda de caminos mínimos, de la cual solo describimos los elementos más importantes.

Antes de describir el trabajo de los procesos en sí, debemos describir la implementación de la gráfica sobre la que trabajan. Esta gráfica está representada por la clase *GraficaAleatoria*, cuya descripción está dada por el siguiente código:

```
1  public class GraficaAleatoria implements Serializable{
2
3      private int nVertex;
4      private int [][] ady;
5      private double connectivity;
6      private int maxVal;
7
8      public GraficaAleatoria(int nVertex, double connectivity, int
           maxVal){
9          this.nVertex = nVertex;
10         this.connectivity = connectivity;
11         this.maxVal = maxVal;
12         ady = new int [nVertex][nVertex];
13         createGraphic();
14     }
15
16     public GraficaAleatoria(int nVertex, int [][] ady){
17         this.nVertex = nVertex;
18         this.ady = ady;
19         connectivity = 0;
20         maxVal = 0;
21     }
22
23     private void createGraphic(){
24         Random r = new Random();
25         for(int i = 0; i < nVertex; i++){
26             ady[i][i] = 0;
```

```
27              int ip1 = i+1;
28
29              if(ip1 < nVertex){
30                  int val = r.nextInt(maxVal-1)+1;
31                  ady[i][ip1] = val;
32                  ady[ip1][i] = val;
33              }
34
35              for(int j = i+2; j < nVertex; j++){
36                  double t = r.nextDouble();
37
38                  if(t < connectivity){
39                      int val = r.nextInt(maxVal-1)+1;
40                      ady[i][j] = val;
41                      ady[j][i] = val;
42                  }
43                  else{
44                      ady[i][j] = -1;
45                      ady[j][i] = -1;
46                  }
47              }
48          }
49      }
50
51      public int [][] getG(){
52          return ady;
53      }
54
55      public int getNumVertices(){
56          return this.nVertex;
57      }
58  }
```

En esencia, la clase *GraficaAleatoria* contiene una matriz cuadrada de números enteros que representan la distancia entre los vértices de la gráfica, los cuales, a su vez, están representados por los índices de la matriz. Así, el elemento *ady[i][j]* se refiere a la distancia entre los vértices i y j en la gráfica. Para representar que no existe una arista entre los vértices a y b, se asigna a la entrada *ady[a][b]* el valor -1.

La clase *GraficaAleatoria* es capaz de crear una gráfica conexa de N vértices con aristas aleatorias entre ellos mediante el método *createGraphic()* de la línea 23. Para ello, el constructor de la línea 8 recibe como argumentos el número de vértices de la gráfica (*nVertex*), la conectividad entre dos vértices (*connectivity*) y la distancia máxima que puede existir entre dos vértices (*maxVal*). Básicamente, el método *createGraphic()* funciona de la siguiente forma:

1. Para cada vértice i, la distancia a sí mismo es 0, es decir *ady[i][i] = 0* (línea

26).

2. Cada vértice i es conectado con el vértice i+1 con una distancia aleatoria entre 1 y *maxVal* (líneas 29 a 33). Este paso asegura que la gráfica sea conexa.

3. Cada vértice tiene probabilidad *connectivity* de estar conectado con cada uno de los otros vértices. Cuando se determina que dos vértices deben estar conectados, se asigna entre ellos una distancia aleatoria entre 1 y *maxVal* (lineas 35 a 47).

El constructor de la línea 16 es utilizado cuando se requiere construir una gráfica ya existente (cuando se dispone de la matriz de adyacencias). Los métodos *getG()* y *getNumVertices()* regresan la matriz y el número de vértices de la gráfica respectivamente.

Una vez descrita la representación de la gráfica, podemos describir la tarea de los procesos. Como mencionamos anteriormente, existen dos tipos de procesos en nuestra solución:

- El proceso *Constructor* que define si el resto de procesos deben distribuir una gráfica creada por él, o buscar un camino mínimo entre dos vértices.
- Los procesos $P_0 ... P_7$ encargados de buscar dicho camino.

El proceso *Constructor* se define como sigue:

```
1  public class Constructor extends RemoteJob{
2
3     int instruction;
4
5     int nVertex;
6     double connectivity;
7     int maxVal;
8
9     int source;
10    int objective;
11
12    public static final int CONSTRUCT = 1;
13    public static final int SOLVE = 2;
14
15    public Constructor(){
16        super("Constructor");
17        ...
18    }
19
```

```
20      public void setInstruction(int instruction){
21          this.instruction = instruction;
22      }
23
24      public void setGraphParameters(int nVertex,double
            connectivity,int maxVal){
25          ...
26      }
27
28      public void setSourceObjective(int source, int objective){
29          ...
30      }
31
32      public void run(){
33          DataMessage m;
34
35          if(instruction == CONSTRUCT){
36              GraficaAleatoria ga = new GraficaAleatoria(nVertex,
                    connectivity, maxVal);
37              m = new DataMessage(0, ga);
38          }
39          else{
40              int [] data = new int[]{source,objective};
41              m = new DataMessage(1, data);
42          }
43          sendMessage("P0",m);
44          finished();
45      }
46  }
```

La clase *Constructor* posee un atributo llamado *instruction* que puede tomar el valor constante *CONSTRUCT*, en cuyo caso el método *run()* de la línea 32 crea una gráfica aleatoria de *nVertex* vértices, con conectividad *connectivity* y donde la distancia más larga posible entre dos vértices es *mxVal*, o el valor constante *SOLVE*, en cuyo caso se indica al resto del sistema que encuentre el camino más corto entre los vértices *source* y *objective*.

Para poder dar valores a las variables mencionadas, la clase *Constructor* posee los métodos *setInstruction(int instruction)* que indica la tarea del proceso, *setGraphParameters(int nVertex,double connectivity,int maxVal)* para establecer los parámetros bajo los cuales se crea una gráfica aleatoria, en caso de que esa sea la tarea deseada y *setSourceObjective(int source, int objective)* para establecer los vértices entre los cuales se desea obtener el camino más corto, de ser el caso.

En el método *run()*, el *Constructor* siempre termina enviando un mensaje al proceso *P0*, indicándole la tarea que debe realizar. Si la gráfica debe ser distri-

buida, se crea un mensaje *DataMessage* que contiene la gráfica y la instrucción 0, indicando el deseo del usuario de distribuir la gráfica. Si se desea encontrar un camino mínimo entre dos vértices, se crea un mensaje *DataMessage* que contiene un arreglo donde se almacenan los índices de los vértice origen y destino del camino, así como la instrucción 1, indicando el deseo del programador de encontrar un camino entre los vértices especificados.

Los procesos $P_0 ... P_7$ que encuentran caminos mínimos, están representados por la clase *ParallelSSP*, y su implementados como sigue:

```
1  public class ParallelSSP extends RemoteJob{
2      private int id;
3      private GraficaAleatoria gp;
4      private boolean [] known;
5      private int [][] g;
6      private int nVer;
7      private Vertex [] handledVert;
8
9      private static final int X = 0;
10     private static final int Y = 1;
11     private static final int Z = 2;
12
13     private int source;
14     private int objective;
15
16     public ParallelSSP(int id){
17         super("P"+id);
18         this.id = id;
19         handledVert = null;
20         gp = null;
21         g = null;
22         known = null;
23         nVer = 0;
24         source = 0;
25         objective = 0;
26     }
27     ...
28 }
```

Cada proceso almacena un identificador de tipo entero *id*. Los atributos *boolean [] known[], int [][] g* e *int nVer* son los parámetros necesarios para encontrar el camino más corto entre dos vértices. Estos atributos son los mismos utilizados en los códigos 5.3 y 5.4. Las constantes *X, Y* y *Z*, son utilizadas para decidir en qué dirección debe ser enviado un mensaje. Las variables *source* y *objective* son los extremos del camino que debe ser encontrado.

El arreglo *handledVert[]* representa los vértices manejados por el proceso *id*. Cada proceso se hace cargo de más o menos $nVer/8$ vértices, cuyos indicadores son de la forma $id + 8 * k$. Es decir, el proceso P_0 se hace cargo de los vértices $0, 8, 16, 24, ..., 8*k$, el proceso P_1 se hace cargo de los vértices $1, 9, 17, 25, ..., 1+8*k$, etc. Este arreglo, utiliza objetos de tipo *Vertex* que contienen información sobre los vértices de la gráfica, y son las entidades enviadas entre los procesos. Su implementación es como sigue:

```
1  public class Vertex implements Serializable{
2
3      private int id;
4      private int distance;
5      private int predecesor;
6
7      public Vertex(int id, int distance){
8          this.id = id;
9          this.distance = distance;
10         this.predecesor = -1;
11     }
12
13     public int getDistance(){
14         return this.distance;
15     }
16
17     public int getPredecesor(){
18         return this.predecesor;
19     }
20
21     public void setPredecesor(int predecesor){
22         this.predecesor = predecesor;
23     }
24
25     public int getId(){
26         return this.id;
27     }
28
29     public void setDistance(int distance){
30         this.distance = distance;
31     }
32 }
```

Es decir, cada objeto *Vertex* almacena el identificador de un vértice, el vértice que es actualmente el anterior en el camino más corto desde *source* hasta él, y la última distancia conocida hasta este vértice desde *source*. Dado que los procesos intercambian objetos de tipo *Vertex*, estos objetos son catalogados como *Serializables*.

El método *run()* de la clase *ParallelSSP* está implementado como en el siguiente código:

```
1  public void run(){
2      try{
3          DataMessage m;
4
5          if(id == 0){
6              m = (DataMessage) receiveMessage("Constructor");
7          }
8          else if(id == 1){
9              m = (DataMessage) receiveIn(X);
10         }
11         else if( id < 4){
12             m = (DataMessage) receiveIn(Y);
13         }
14         else{
15             m = (DataMessage) receiveIn(Z);
16         }
17
18         int inst = m.getInstruction();
19
20         switch(inst){
21             case 0:
22                 distributeGraphic(m);
23                 break;
24             case 1:
25                 int [] data = (int []) m.getData();
26                 this.source = data[0];
27                 this.objective = data[1];
28                 solveSSP(m);
29                 break;
30         }
31     }
32     catch(Exception e){
33         e.printStackTrace();
34     }
35     finished();
36 }
```

El programa inicia recibiendo un mensaje desde otro proceso. El proceso *P0* recibe el mensaje desde el proceso *Constructor* (línea 6). Los procesadores P1 a P7 reciben el mensaje desde uno de sus vecinos mediante el método *receiveIn()*. Si recordamos la Figura 5.3, cada proceso en el hipercubo representa un vértice de éste, y puede recibir un mensaje desde tres direcciones: X, Y y Z. El método *receiveIn()* recibe como parámetro la dirección desde la cual se desea recibir un mensaje y con base en esta dirección y en el identificador *id* del proceso, determina

el identificador del proceso desde el cual debe recibir el mensaje. Entre las líneas 8 y 16, los procesos P1 a P7 reciben el mensaje que les indica la tarea que deben realizar. Este mensaje es analizado en las líneas 20 y 30. Si la instrucción del mensaje es 0, el mensaje es retransmitido a través del hipercubo mediante el método *distributeGraphic()*. Si la instrucción del mensaje es 1, los atributos *source* y *objective* son desempacados y se invoca al método *solveSSP()* que inicia el algoritmo de caminos mínimos. Este método recibe como argumento el mismo mensaje obtenido anteriormente, pues antes de iniciar la ejecución del algoritmo principal, debe redistribuir el mensaje a sus vecinos para que a su vez, ellos inicien su ejecución. La implementación del método *solveSSP()* es como sigue:

```
1  public void solveSSP(DataMessage m){
2      mountGraphic();
3
4      if(id == 0){
5          sendIn(X, m);
6          sendIn(Y, m);
7          sendIn(Z, m);
8          solvefor0();
9      }
10     else{
11         if(id == 1){
12             sendIn(Y, m);
13             sendIn(Z, m);
14         }
15         else if(id == 2 || id == 3){
16             sendIn(Z, m);
17         }
18         solvefornon0();
19         }
20         constructPath();
21     }
22 }
```

La primer tarea del método *solveSSP()* consiste en leer del archivo *GraficaId.dat* la información de la gráfica con la cual debe trabajar el algoritmo de caminos mínimos, así como inicializar apropiadamente los arreglos *known[]* y *handledVert[]*. En el arreglo *known[]*, todos los vértices son desconocidos, excepto por el vértice *source*, cuya distancia mínima es 0. Como ya mencionamos, el arreglo *handledVert[]* contiene a los vértices cuyo identificador es de la forma $id + 8 * k$, donde k es el índice de cada elemento del arreglo. Esta tarea se lleva a cabo mediante el método *mountGraphic()*. Entre las líneas 5 a 7 y 11 a 17, el mensaje recibido en el método *run()* es redistribuido hacia los procesos vecinos vecinos.

Hasta este punto, todos los procesos están listos para iniciar su ejecución. Sin

embargo, el proceso *P0* realiza una labor ligeramente diferente a la realizada por el resto de los procesos. Debido a ello, implementamos dos métodos: *solvefor0()* que realiza la tarea del procesador *P0* (línea 8) y *solvefornon0()*, que realiza la tarea de los procesadores *P1 ... P7* (línea 18). Finalmente, implementamos un método llamado *constructPath()* que mediante una cadena de caracteres, fabrica una representación del camino más corto entre los vértices *source* y *objective*, y envía esta información al objeto *Master*. La implementación de estos métodos se revisa en los párrafos sucesivos.

Para revisar la implementación de los métodos *solvefor0()* y *solvefornon0()*, es necesario revisar dos métodos importantes de la clase *ParallelSSP*. El método *searchLocalMin()* encuentra el vértice con la la distancia local más corta a *source* de entre aquellos vértices locales manejados por este proceso, es decir, los del arreglo *handledVert[]*. Su implementación es la siguiente:

```
1  public Vertex searchLocalMin(){
2      Vertex min = handledVert[0];
3
4      for(int i = 1; i < handledVert.length; i++){
5          Vertex v = handledVert[i];
6
7          if(v != null){
8              min = minV(min,v);
9          }
10     }
11     return min;
12 }
```

En cada iteración, el vértice con la distancia más corta es actualizado en la línea 8 mediante el método *minV()*, el cual realiza la misma función que el método *minV()* de los códigos 5.3 y 5.4, excepto que en vez de recibir identificadores enteros como argumentos, recibe vértices de tipo *Vertex*. Así, después de recorrer toda la lista de vértices locales, se regresa el vértice con la distancia más corta a *source*, conocida hasta el momento.

El segundo método que debemos revisar es llamado *updateDistances()* y es utilizado para actualizar la distancia de todos los vértices locales al vértice *source* una vez que se determina la distancia mínima para un vértice dado, llamado *lastKnown*. Su implementación es la siguiente:

```
1  private void updateDistances(Vertex lastKnown){
2
3     for(int i = 0; i < handledVert.length; i++){
4          Vertex v = handledVert[i];
```

```
5
6           if(v != null && !known[v.getId()]){
7               int dista = valid(v.getDistance());
8               int distb = valid(valid(lastKnown.getDistance()) +
                    valid(g[lastKnown.getId()][v.getId()]));
9
10              if(dista <= distb){
11                  v.setDistance(dista);
12              }
13              else{
14                  v.setDistance(distb);
15                  v.setPredecesor(lastKnown.getId());
16              }
17          }
18      }
19  }
```

Suponemos que el vértice *lastKnown* pasado como argumento es el último para el cual su distancia mínima ha sido descubierta y establecida. Así, para cada vértice v del arreglo *handledVert[]*, se compara la distancia más corta actual de v, con la distancia entre *lastKnown* y v (recordemos que la distancia de *lastKnown* es mínima). Si la segunda es menor que la primera, eso quiere decir que existe un camino más corto a v que el que se conocía hasta el momento y pasa por *lastKnown*. De esta manera, en las líneas 14 y 15, se actualiza la distancia más corta hasta v, y se avisa a este último que su predecesor en el camino más corto desde *source* es *lastKnown*

La implementación del método *solvefor0()* surge directamente del código 5.3:

```
1   public void solvefor0(){
2       int i = 1;
3
4       while(i < nVer){
5           Vertex minVertex = searchLocalMin();
6           Vertex received;
7
8           received = (Vertex) receiveIn(Z);
9           minVertex = minV(minVertex,received);
10
11          received = (Vertex) receiveIn(Y);
12          minVertex = minV(minVertex,received);
13
14          received = (Vertex) receiveIn(X);
15          minVertex = minV(minVertex,received);
16
17          known[minVertex.getId()] = true;
18
```

```
19          Vertex lastKnown = minVertex;
20
21          sendIn(X, lastKnown);
22          sendIn(Y, lastKnown);
23          sendIn(Z, lastKnown);
24
25          updateDistances(lastKnown);
26          i++;
27      }
28  }
```

El vértice mínimo local es encontrado en la línea 5. Entre las líneas 8 y 5 se determina el vértice con la distancia mínima global, el cual es actualizado en el arreglo *known[]* de vértices conocidos, y enviado en las direcciones X, Y y Z. Finalmente, la distancia mínima de los vértices locales es actualizada en la línea 25.

El método *solvefornon0()* es implementado a partir del código 5.4:

```
1   public void solvefornon0(){
2       int i = 1;
3
4       while( i < nVer){
5           Vertex localMinVertex = searchLocalMin();
6           Vertex received = null;
7           Vertex lastKnown = null;
8           if(id < 4){
9               received = (Vertex) receiveIn(Z);
10              localMinVertex = minV(localMinVertex,received);
11          }
12          else{
13              sendIn(Z,localMinVertex);
14          }
15          if(id < 4){
16              if(id == 1){
17                  received = (Vertex) receiveIn(Y);
18                  localMinVertex = minV(localMinVertex,received);
19              }
20              else{
21                  sendIn(Y, localMinVertex);
22              }
23          }
24          if(id == 1){
25              sendIn(X, localMinVertex);
26          }
27          if(id == 1){
28              lastKnown = (Vertex) receiveIn(X);
29              known[lastKnown.getId()] = true;
30              sendIn(Y, lastKnown);
```

```
31              sendIn(Z, lastKnown);
32          }
33          else if (id < 4){
34              lastKnown  = (Vertex) receiveIn(Y);
35              known[lastKnown.getId()] = true;
36              sendIn(Z, lastKnown);
37          }
38          else{
39              lastKnown = (Vertex) receiveIn(Z);
40              known[lastKnown.getId()] = true;
41          }
42          updateDistances(lastKnown);
43          i++;
44      }
45  }
```

En la línea 5 se encuentra el vértice local con la distancia más corta. Entre las líneas 8 a 26, se encuentra el vértice mínimo entre el local y los de los vecinos. El resultado es enviado al proceso *P0*, quien determina el mínimo global y lo distribuye en la red, conocido ahora como *lastKnown*. Entre las líneas 27 a 41, se recibe y redistribuye el vértice *lastKnown*. Finalmente, es necesario actualizar las distancias mínimas de los vértices locales mediante el método *updateDistances()* en la línea 42.

Solo nos queda revisar el método *constructPath()* para terminar con el análisis de los procesos. La tarea de este método consiste en encontrar una representación de cadena de caracteres que indique el camino más corto encontrado entre los vértices *source* y *objective*, así como su distancia total, para después enviar estos datos al objeto *Master* para que puedan ser revisados por el usuario. Al igual que con el algoritmo paralelo distribuido de caminos mínimos, la operación del método *constructPath()* es diferente entre el proceso *P0* y el resto de procesos. Su implementación es como sigue:

```
1  public void constructPath(){
2      if(id == 0){
3          pathP0();
4      }
5      else{
6          pathPK();
7      }
8  }
```

El método *pathP0()* mantiene comunicación con todos los procesos del hipercubo. Su implementación es la siguiente:

```
1  public void pathP0(){
2      int last = objective;
3      String path = objective + "";
4
5      while( source != last){
6          int location = last % 8;
7          if(location != 0){
8              String procHost = "P" + location;
9              sendMessage(procHost, last);
10             last = (Integer) receiveMessage(procHost);
11         }
12         else{
13             int index = last/8;
14             last = handledVert[index].getPredecesor();
15         }
16         path = last + "␣" + path;
17     }
18     for(int i = 1; i < 8; i++){
19         sendMessage("P" + i, -1);
20     }
21     saytoMaster(path);
22     int proc = objective % 8;
23     int dist;
24     if(proc != 0){
25         sendMessage("P"+proc,objective);
26         dist = (Integer) receiveMessage("P"+proc);
27     }
28     else{
29         dist = handledVert[objective/8].getDistance();
30     }
31     for(int i = 1; i < 8; i++){
32         if(i != proc){
33             sendMessage("P"+i, -1);
34         }
35     }
36     saytoMaster(dist);
37 }
```

Recordemos que cada elemento *Vertex* tiene una referencia hacia el vértice anterior en el camino más corto hasta él. De esta forma, el proceso *P0* inicia comunicándose con el proceso que aloja al vértice *objective*, que es el último en el camino más corto desde *source* hasta *objective*, para preguntarle quién es el vértice anterior a éste. El apuntador obtenido se convierte en el nuevo "último" vértice del camino más corto. Este proceso es repetido para todos los nuevos "últimos" vértices hasta que uno de ellos señale al vértice origen (líneas 5 a 17). Una vez obtenido el camino más corto entre *source* y *objective*, el proceso *P0* envía a todos los procesos el número entero -1 para indicarles que esta tarea ha terminado (líneas 18 a

20). *P0* envía el camino construido al objeto *Master* (línea 21) y continua comunicándose con el proceso que aloja al vértice *objective*, para pedirle la distancia de éste (líneas 24 a 30). Una vez recibida esta distancia, *P0* la envía al objeto *Master* (línea 36) y manda a todos los otros procesos la señal -1 para indicarles que su tarea ha terminado (líneas 31 a 35).

Para todos los procesos distintos de *P0*, el código para construir el camino mínimo es como sigue:

```
1  public void pathPK(){
2      boolean flag = true;
3      while(flag){
4          int req = (Integer) receiveMessage("P0");
5          if(req != -1){
6              int index = (req - id) / 8;
7              int anterior = handledVert[index].getPredecesor();
8              sendMessage("P0", anterior);
9          }
10         else{
11             flag = false;
12         }
13     }
14     int k = (Integer) receiveMessage("P0");
15     if(k != -1){
16         int index = (k-id)/8;
17         int dist = handledVert[index].getDistance();
18         sendMessage("P0", dist);
19     }
20 }
```

El método *pathPK()* inicia esperando una instrucción del proceso *P0*. Si el dato recibido es distinto de -1, significa que *P0* está solicitando al predecesor del vértice recibido. Entre las líneas 6 y 8, el predecesor en el camino mínimo es encontrado y devuelto al proceso *P0*. Este procedimiento es repetido hasta que el dato recibido sea -1, indicando que no hay más peticiones de vértices anteriores. Una vez que esta señal es recibida, se recibe un dato más desde el proceso *P0*. Si el dato es diferente de -1, significa que *P0* desea saber la distancia almacenada en el vértice cuyo identificador es el dato recibido. Esta distancia es encontrada y enviada de vuelta a *P0* entre las líneas 16 y 18. Si el dato recibido es -1, significa que *P0* no desea nada y se puede terminar la ejecución del proceso local.

Master y Managers

Una vez definidos los procesos, debemos implementar programas ejecutables para el objeto *Master* y para los *Managers*. Sin embargo, la tarea de los *Managers*

siempre radica únicamente en esperar una petición de conexión del objeto *Master*. Debido a ello, el código para el método *main()* que ejecuta un *Manager* es como especificamos en la sección 4.8:

```
1  public static void main(String [] argv){
2      Manager m = new Manager(port, waitTime);
3
4      try{
5          m.connectToMaster();
6      }
7      catch(ConnectionException e){
8          e.printStackTrace();
9      }
10 }
```

Donde *port* y *waitTime* deben especificarse para cada *Manager*.

Para el objeto *Master*, sin embargo, el método *main()* que la ejecuta debe crearla, asignarle procesos, iniciar la ejecución del sistema y después leer los mensajes del proceso *P0* que especifican el camino más corto entre dos vértices, así como su longitud. La implementación es como sigue:

```
1  public static void main(String [] argv){
2      Master m = new Master("HiperCube.config");
3
4      try{
5          for(int i = 0; i < 8; i++){
6                  m.addRemoteJob(new ParallelSSP(i));
7          }
8          Constructor c = new Constructor();
9          c.setInstruction(Constructor.SOLVE);
10         c.setSourceObjective(10,60);
11         m.addRemoteJob(c);
12
13         Thread.sleep(1000);
14
15         m.startSystem();
16
17         m.waitToFinish();
18
19         String path = (String) m.messageFrom("P0");
20         System.out.println("El␣camino␣es:␣" + path);
21
22         int dist = (Integer) m.messageFrom("P0");
23         System.out.println("Su␣distancia␣es␣" + dist);
24
25         m.shutdown();
26     }
```

```
27      catch(Exception e){
28          e.printStackTrace();
29      }
30  }
```

Después de crear al objeto *Master* con el archivo de configuración *HiperCube.config*, entre las líneas 5 y 7 se le asignan los procesos *P0 ... P7*. El proceso *Constructor* es creado en la línea 8 y se le instruye para encontrar el camino más corto entre los vértices 10 y 60 en las líneas 10 y 11. Entre las líneas 13 y 17, se pone en ejecución el sistema y se espera a que termine su ejecución. Una vez terminada la ejecución del sistema, las líneas 19 a 23 obtienen del proceso *P0* el camino más corto entre los vértices 10 y 60, así como su distancia y muestran estos datos al usuario. Finalmente, la línea 25 detiene al sistema completo con la instrucción *shutdown()*.

El archivo de configuración

Disponemos de un cluster de 5 computadoras, en el cual debemos mapear 9 procesos: *P0* a *P7* y el *Constructor*. Siguiendo nuestras propias recomendaciones, reservamos una computadora física para uso exclusivo del objeto *Master*. Eso nos deja disponibles 4 computadoras para 9 procesos. Si analizamos un poco la ejecución del sistema, el proceso *Constructor* termina su ejecución justo cuando el proceso *P0* inicia la suya. Podemos entonces poner a este proceso en cualquier computadora sin afectar la ejecución del sistema, particularmente, en la misma computadora que *P0*. Para los procesos *P0* a *P7*, hemos decidido mapear dos procesos a cada una de las 4 computadoras físicas disponibles, donde los procesos *PX, PY* residen en la misma computadora si *PX* es vecino en la dirección *Z* de *PY*. Es decir, P_i y P_{i+4} residen en la misma computadora para $0 \leq i \leq 4$.

Respecto a las computadoras físicas, todas se conectan a través del puerto 3305 y tienen las siguientes direcciones:

```
192.168.1.100
192.168.1.101
192.168.1.102
192.168.1.103
192.168.1.104
```

De las cuales, el objeto *Master* debe ocupar la computadora física 192.168.1.100. Con estas especificaciones, podemos construir el archivo de configuración *HiperCube.config* como sigue:

```
#Hosts
Host = (192.168.1.101,3305,M1)
Host = (192.168.1.102,3305,M2)
Host = (192.168.1.103,3305,M3)
Host = (192.168.1.104,3305,M4)

#Processes
Process = (P0,M1,[P1,P2,P3,P4,P5,P6,P7,Constructor])
Process = (P1,M2,[P0,P3,P5])
Process = (P2,M3,[P0,P3,P6])
Process = (P3,M4,[P0,P1,P2,P7])
Process = (P4,M1,[P0,P5,P6])
Process = (P5,M2,[P0,P1,P4,P7])
Process = (P6,M3,[P0,P2,P4,P7])
Process = (P7,M4,[P0,P3,P5,P6])
Process = (Constructor,M1,[P0])
```

El proceso *P0* está conectado a todos los demás, pues establece comunicación con todos en el método *constructPath()* de la clase *ParallelSSP*. Sin embargo, el algoritmo principal no resulta afectado y respeta la estructura de hipercubo.

5.2. Ejemplo 2: La criba de Eratóstenes

5.2.1. Definición del problema

Sea $N \in \mathbb{N}$. El objetivo del programa es encontrar a todos los números primos menores o iguales a N [9]. Para resolver este problema, utilizamos varios resultados de algebra y teoría de números. No demostraremos aquí tales resultados.

Para verificar que un número K es primo, éste debe ser dividido por todos los primos menores que $\sqrt{K}$. No es necesario dividirlo por números compuestos, ya que todos los números compuestos tienen factores primos. Debido a la división por tentativa, tampoco es necesario dividir K por números mayores que $\sqrt{K}$ [9].

Así, para determinar los números primos menores que N primero se deben determinar los números primos menores o iguales que $\sqrt{N}$. Después, cada número K tal que $\sqrt{N} < K \leq N$ (a estos números los llamamos candidatos) debe ser dividido por todos los primos menores que $\sqrt{N}$. Si en algún momento una de estas divisiones tiene residuo 0, entonces K no es un número primo y debe probarse con el siguiente candidato. Este procedimiento se realiza de manera secuencial para todos los candidatos. Sabiendo que todos los números pares tienen a 2 como

factor, solo tomaremos en cuenta como candidatos a aquellos números impares entre $\sqrt{N}$ y N [9].

Siendo más formales, sean:

$$\Omega = \{p \,\|\, p \in \mathbb{P}, 1 < p \leq \sqrt{N}\}$$
$$\Gamma = \{k \,\|\, \sqrt{N} < k \leq N, k \neq 2n \ \forall n \in \mathbb{N}\}$$

Entonces, $k \in \Gamma$ es primo sii:

$$k \bmod p = 0 \ \forall p \in \Omega \tag{5.1}$$

5.2.2. Algoritmo secuencial

El código Listing 5.5 resuelve de forma secuencial la ecuación 5.1 [9]:

Código 5.5: Solución secuencial al problema de la criba de Eratóstenes

```
1  int lim = sqrt(N) ; //Se encuentran todos los primos menores que
       raiz de N
2  int index = 0;
3  int loc = 1; //Indice del siguiente primo
4  int next = 5; //Siguiente candidato
5  int [] primeFilters; //Primos menores que sqrt(N)
6  primeFilters[0] = 3; //Se conoce el primer primo
7
8  while(next < lim){
9      while(index < loc){
10       //Se verifica cada candidato contra todos los primos
             encontrados
11       if(next % primeFilters[index] != 0){
12         index++;
13       }
14       else{
15          index = 0;
16          salir de loop interno;
17       }
18     }
19     //Se agrega un nuevo primo
20     if(index == loc){
21         primeFilters[loc] = next;
22         loc++;
23         index=0;
24     }
25     next += 2;
26 }
```

```
27
28   //Hasta ahora, se tienen todos los primos menores que î N . Se
         calcularan todos
29   //los primos menores que N de la siguiente forma
30
31   int candidate = sqrt(N) ;
32   if(candidate % 2 == 0) candidate++; //Se toma el primer
         candidato >= sqrt(N) impar
33   int count;
34
35   //Para cada candidato impar menor que N
36   while(candidate < N){
37       count = 0;
38       //Se cuenta con count el n\'umero de pruebas pasadas
39       for(int i = 0; i < primeFilters.length; i++){
40           if(candidate % primeFilters[i] == 0){
41                break;
42           }
43           else{
44              count++;
45           }
46       }
47       //Si candidate pasa todas las pruebas, es un primo
48       if(count == primeFilters.length){
49           candidate es primo;
50       }
51       candidate += 2;
52   }
```

De las líneas 1 a 27, se calcula el conjunto Ω. Esto se hace dividiendo cada número p impar menor que $\sqrt{N}$ entre todos los primos encontrados hasta entonces. Si p resulta ser un primo, se añade a la lista de primos conocidos, almacenados en el arreglo *primeFilters[]*. El resto del algoritmo toma en cuenta a cada elemento k del conjunto Γ mediante la variable *candidate* y lo divide entre todos los elementos de Ω. Si en cada división el residuo es diferente de 0, entonces *candidate* es un primo.

Se puede observar que las verificaciones de cada candidato contra cada entrada del arreglo de primos *primeFilters[]* es independiente de cualquier verificación de otros candidatos. Conceptualmente, se puede pensar en un "flujo" de candidatos pasando concurrentemente en el arreglo de primos, haciendo las verificaciones en paralelo[9].

5.2.3. Paralelismo potencial

Hemos mencionado que las verificaciones de los candidatos son independientes entre sí. De hecho, la única dependencia entre las comparaciones en el algoritmo secuencial es la posibilidad de que un número compuesto sea descartado y no se requieran las comparaciones subsecuentes. Si se pasa por alto este hecho, se notan dos tipos de independencia entre las operaciones[9]:

1. Dado un candidato, las divisiones entre los primos de prueba pueden ser hechas en paralelo. Es decir, un candidato dado C puede ser dividido entre 3,5,7,... al mismo tiempo, verificando al final si alguna de las divisiones produjo residuo 0 [9].

2. Dado un primo de prueba, varios candidatos pueden ser verificados con este primo al mismo tiempo. Es decir, dados los candidatos 7,9,11,13,... y el primo 3, los candidatos pueden ser divididos entre 3 al mismo tiempo, eliminando los candidatos divisibles entre 3, y verificando los que queden contra el siguiente primo [9].

Aunque los dos puntos anteriores definen explícitamente las operaciones en paralelo, en el primer punto se llevan a cabo operaciones innecesarias, pues cuando se tiene que $C \bmod P = 0$ para un candidato C y un primo P, no es necesario probar a C contra más primos. Sin embargo, se usan las características de independencia de los puntos anteriores para realizar el algoritmo paralelo [9].

5.2.4. Algoritmo paralelo distribuido

El patrón de filtros y tuberías paralelas de la sección 2.5.1 se ajusta a las necesidades de paralelismo de este problema.

Como mencionamos, en un pipeline, un conjunto de datos fluye de un procesador a otro. Cada procesador recibe un dato de su vecino anterior, realiza una parte del cómputo total sobre ese dato y lo envía a su vecino siguiente. Este ciclo se repite para cada dato y todos los procesadores realizan cálculos al mismo tiempo, definiendo así, el cómputo total. En un pipeline ideal para este caso de estudio, se disponen de tantos procesadores (filtros) como números primos de prueba hay. Cada procesador filtra el flujo de candidatos de entrada, eliminando aquellos valores divisibles entre el primo asociado a cada procesador. Esto se muestra en la Figura 5.5. En la parte izquierda del pipeline, el primer proceso (G) genera la secuencia de datos: 3,5,7,9,11,13,15,..., y el último proceso (I) realiza el cómputo

final sobre el conjunto de primos resultantes (mostrarlos, almacenarlos o contarlos). Cada filtro en el pipeline divide a cada candidato que recibe entre su primo asignado. El candidato es eliminado del flujo de datos si el residuo de la división es 0. De otra forma, el candidato se pasa al siguiente filtro [9].

Figura 5.5: Pipe-Line de primos de prueba

En general, siempre hay más filtros que procesadores disponibles. En estos casos, se agrupan varios filtros en cada procesador y cada candidato se prueba localmente contra todos estos filtros. Cada procesador ejecuta un algoritmo muy similar al del código 5.5 utilizando únicamente un subconjunto de primos consecutivos de Ω. El cálculo de los filtros necesarios para establecer el pipeline también puede obtenerse como en el código 5.5 [9].

El proceso principal en cada procesador en el pipeline cuenta el número de filtros necesarios, los distribuye uniformemente (cada procesador se ocupa de K filtros, donde si F es el número de filtros totales y P es el número de procesadores, entonces $K = F/P$) y después realiza el filtrado de los candidatos hasta que el generador envía una señal de término a través del pipeline (Figura 5.6) [9].

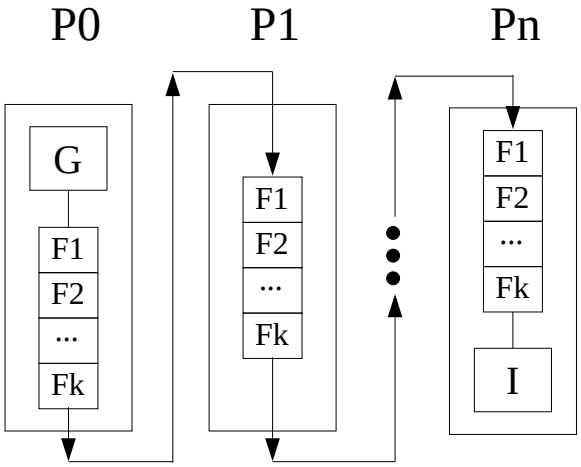

Figura 5.6: Agrupamiento de los filtros en los procesadores disponibles

El código 5.6 encuentra el subconjunto de Ω que cada procesador usa para filtrar a los candidatos. Para ello, se requiere acceso a la variable *NumProc*, que representa el número de procesadores del pipeline. Además, se requiere de la variable *ident*, que representa el identificador numérico de cada procesador en el pipeline. Tanto *NumProc* como *ident* deben ser especificados antes de la ejecución del algoritmo. Como valores de retorno, el programa calcula la variable *load*, en donde se almacena el número de primos que cada computadora utiliza para filtrar a los candidatos. También es calculada la variable *first*, que es la posición del arreglo *primeFilters[]* donde se encuentra el primer primo que cada computadora toma en cuenta para filtrar a los candidatos. En términos simples, los primos que cada procesador utiliza para filtrar candidatos están en el arreglo *primeFilters[]*, y van desde *primeFilters[first]* hasta *primeFilters[first + load - 1]* [9].

Código 5.6: Código para encontrar los filtros de cada procesador

```
1  public void findLoad(){
2
3      int lim = sqrt(N)
4      int index = 0;
5      //La siguiente posicion disponible en el arreglo de primos
6      int loc = 1;
7      int next = 5;
8      //El arreglo de primos
9      primeFilters = new int[lim / 2];
10
11     for(int i = 0; i < primeFilters.length; i++){
12         primeFilters[i] = -1;
13     }
14
15     primeFilters[0] = 3;
16     //Se encuentran todos los primos menores que sqrt(N)
           secuencialmente
17     while(next < lim){
18         while(index < loc){
19             if(next % primeFilters[index] != 0){
20                 index++;
21             }
22             else{
23                 index = 0;
24                 break;
25             }
26         }
27
28         if(index == loc){
29             primeFilters[loc] = next;
30             loc++;
31             index = 0;
32         }
```

```
33            next += 2;
34        }
35        //Se almacena en load el nÃºmero de primos que cada
              procesador
36        // maneja
37        load = loc / NumProc;
38
39        //Se almacena en first el indice del arreglo primeFilters en
              el cual se
40        //encuentra el primer primo a partir del cual esta
              computadora empieza a
41        // filtrar
42        first = (ident - 1) * load;
43
44        if(first >= loc){
45            load = 0;
46        }
47    }
```

El código 5.7 representa el trabajo individual que cada filtro debe realizar, comprobando a cada candidato contra el subconjunto de Ω asignado a dicho filtro y enviando a los candidatos que pasen las comprobaciones al siguiente filtro. El arreglo *prime[]* representa los primos de prueba locales y debe ser inicializado antes de la ejecución del programa. Para esto, utilizamos el código 5.6 [9].

Código 5.7: Trabajo de cada filtro del pipeline

```
1   public void filterNumbers(){
2
3      int next = receiveNext();
4
5      /*El ciclo principal divide cada candidato entre todos los
6       * primos asignados a este filtro.
7       */
8      while(next > 0){
9          int index = 0;
10         while(index < prime.length){
11             if(next % prime[index] != 0){
12                 index++;
13             }
14             else{
15                 break;
16             }
17         }
18         if(index == prime.length){
19             sendNext(next);
```

```
20          }
21          next = receiveNext();
22      }
23      sendNext(next);
24  }
```

El trabajo en el código 5.7 se lleva a cabo entre las líneas 8 y 22. La variable *next* simboliza el siguiente candidato. La comprobación de la línea 8 *next > 0* comprueba que siga habiendo candidatos para evaluar. En este sentido, *next = -1* sucede solo cuando el generador G de la figura 5.6 envía 0 como señal de terminación. Las líneas 10 a 17 realizan las comprobaciones necesarias para un candidato dado. La variable *index* apunta hacia la posición del arreglo *prime[]* donde se encuentra el primo contra el que debe ser comparado el candidato *next*. Si en algún momento la comprobación falla (esto es que el residuo de la división sea 0 para algún primo de *prime* y algún candidato) el programa se sale del ciclo interno. La comprobación de la línea 18 verifica si *next* pasó todas las pruebas. De ser así, *next* se envía al siguiente filtro. En caso contrario, *next* es ignorado y se espera hasta recibir el siguiente candidato.

Comunicación y Buffers

El uso de un algoritmo secuencial dentro de cada procesador hace surgir un problema en la solución paralela global: la coordinación de la comunicación. Para evitar desbordamientos de memoria, la comunicación entre procesadores debe ser síncrona. Cuando algún procesador llega a la instrucción *sendNext()*, el proceso debe esperar hasta que la instrucción *receiveNext()* sea llamada por el proceso receptor y viceversa. Cuando ambos procesos llegan al punto de comunicación, ésta se ejecuta. Este proceso provoca un paro total del pipeline y se pierde tiempo de procesamiento. Si un proceso del final del pipeline termina con un número candidato rápidamente, es forzado a esperar disponibilidad del proceso siguiente y hace esperar al proceso anterior. Este tiempo forzado de espera reduce la eficiencia del programa e incrementa el tiempo de ejecución [9].

Se puede evitar (o al menos reducir) este problema haciendo que la comunicación entre procesos sea asíncrona, es decir, introduciendo buffers. En cada procesador, se añade un proceso *Enlace*, dedicado a recibir mensajes del filtro anterior para almacenarlos en una "cola de mensajes" (buffer). El trabajo del proceso *Enlace* es simple: recibir información del filtro anterior y depositarla en el buffer. Todo esto dentro de un ciclo infinito hasta que se reciba la señal de terminación (figura 5.7) [9].

Con este nuevo esquema, si un filtro termina su trabajo antes que el si-guiente, el

primero envía al candidato sin tener que esperar a que el segundo finalice, pues el proceso *Enlace* y el buffer se encargan de recibir y guardar al candidato hasta que el segundo filtro esté disponible para procesarlo. El caso contrario es que el segundo filtro finalice antes que el primero, pero esto no es un problema, pues si existe un dato almacenado en el buffer, puede tomarlo inmediatamente sin tener que esperar a establecer comunicación con el filtro anterior.

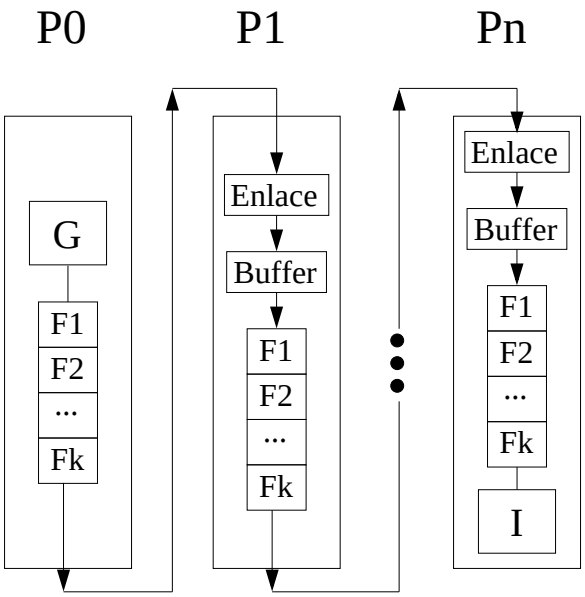

Figura 5.7: Pipeline con buffers. P_i son los procesadores físicos, G es el generador de candidatos, I es el proceso final donde se decide que hacer con los primos calculados, los F_i son los filtros locales para cada procesador.

Aún existe la posibilidad de que un proceso trate de leer datos de un buffer vacío, o que el *Enlace* trate de escribir datos en un buffer lleno. En ambos casos, los procesos involucrados deben esperar a que el buffer esté disponible. En el primer caso, el proceso principal trata de obtener un dato de un buffer vacío y tiene que esperar a que el *Enlace* reciba un dato y lo escriba en el buffer. En el segundo caso, el buffer está lleno y el *Enlace* tiene que esperar hasta que el proceso principal lea un dato del buffer y lo desocupe [9].

Dado que no es posible obtener comunicación directa entre el buffer y el proceso emisor (pues ambos procesos se encuentran en computadoras distintas), el emisor debe esperar una señal de confirmación por parte del *Enlace* que indica que el mensaje ha sido recibido y almacenado correctamente en el buffer. Después de recibir la confirmación, el emisor continúa con su trabajo. Por su parte, el

Enlace espera un dato del emisor. Cuando éste llega, lo almacena en el buffer y envía una señal de confirmación al emisor. Si el buffer estuviera lleno cuando llega un dato desde el emisor, el proceso receptor espera hasta que el buffer tenga un lugar disponible donde guardar el dato antes de enviar la señal de confirmación [9].

Aunque existe la posibilidad de buffers llenos que detienen el flujo de los datos, si se tienen los suficientes recursos computacionales para hacer un buffer de gran tamaño, este fenómeno es rara vez un problema. Mientras más grande es el tamaño del buffer, menores son los retrasos debido a esperas innecesarias [9].

Este sistema de buffers lleva al problema del "productor - consumidor", tratado en capítulos anteriores. El código 5.8 representa las acciones principales de cada filtro, desde que se inicia toda la operación [9].

Código 5.8: Trabajo principal de cada filtro

```
1  public void mainWork(){
2     //Se recibe el identificador de esta maquina
3     ident = receiveMessage();
4
5     //Se recibe el numero de procesadores
6     NumProc = receiveMessage();
7
8     //Se recibe el numero objetivo
9     target = receiveMessage();
10
11    //Se calcula la raiz cuadrada del objetivo
12    sqrtMax = sqrt(target);
13
14    /*Se envian los datos al siguiente filtro siempre y cuando no
         sea
15    *el ultimo procesador*/
16    if(ident != NumProc){
17        sendMessage(ident + 1);
18        sendMessage(NumProc);
19        sendMessage(target);
20    }
21
22    //Se calcula la carga de trabajo de este filtro
23    findLoad();
24
25    //Se inicia el buffer de X elementos (200 funciona bien para
         nuestro ejemplo)
26    Buffer buffer = new Buffer(200);
27    //Se inicia el proceso Enlace.
28    //Este Enlace requiere acceso al buffer y al manejador de
         mensajes local
```

```
29      Enlace en = new Enlace(buffer);
30      en.start();
31
32      //Se filtran todos los candidatos
33      filterNumbers();
34    }
```

En el código 5.8 desde la línea 3 hasta la 12, cada filtro recibe información necesaria para ejecutar el trabajo principal. Esta información es el nombre o identificador del proceso local, el número de procesadores que tiene el pipeline y el número *target* que es del cual queremos calcular todos los primos menores que él. En la línea 12 se calcula $\sqrt{\text{target}}$, necesaria para calcular el subconjunto de Ω que cada filtro utiliza para filtrar a los candidatos. Las líneas 16 a 20 distribuyen la información recibida a través del pipeline; la línea 23 calcula los filtros locales (primos de los cuales se encarga cada procesador) mediante el código 5.6. Las líneas 26 a 30 inician el buffer y el proceso *Enlace*. Dejamos los detalles de implementación de este proceso para la siguiente sección. Finalmente, la línea 33 inicia el proceso de filtrado mediante el código 5.7.

5.2.5. Implementación con J-MIPS

Tal como especificamos en la sección 4.8, para realizar un programa en J-MIPS requerimos de los siguientes componentes:

- La definición de los procesos *RemoteJob* que deben ser ejecutados.
- La clase Main de los *Managers*.
- La clase Main del objeto *Master*.
- El archivo de configuración.

Al resolver este problema, queremos que los números primos encontrados se muestren en la pantalla. Por último, el sistema debe mostrar el tiempo que se ha requerido para resolver el problema.

Los procesos

Existen dos tipos de procesos en el pipeline: Los filtros de números primos y el generador de candidatos. El trabajo del generador de candidatos lo representamos mediante la clase *NGenerator* como sigue:

```
1  public class NGenerator extends RemoteJob{
2
3      private int initial;
4      private int limit;
5      private int numprocs;
6
7      public NGenerator(int numprocs, int target){
8          super("Generador");
9          this.numprocs = numprocs;
10         this.limit = target;
11         this.initial = 0;
12     }
13
14     public void run(){
15         sendMessage("P1",numprocs);
16         sendMessage("P1",limit);
17         long time1 = System.currentTimeMillis();
18         saytoMaster(time1);
19         initial = (int) floor(sqrt(limit));
20
21         if(initial % 2 == 0){
22             initial++;
23         }
24
25         boolean gate = true;
26         int n;
27
28         while(gate){
29             n = generateNext();
30             if(n < 0){
31                 gate = false;
32             }
33             sendMessage("P1",n);
34         }
35         finished();
36     }
37
38     public int generateNext(){
39         int k = initial;
40         initial += 2;
41         if(initial > limit){
42             initial = -1;
43         }
44         return k;
45     }
46 }
```

La clase *NGenerator* tiene tres atributos:

- **initial**. Que representa el primer candidato para los filtros.
- **limit**. Que representa el número objetivo.
- **numprocs**. Que representa el número de filtros (procesos) en el pipeline.

El constructor de la clase *NGenerator* recibe el número objetivo y el número de procesos en el pipeline. Este mismo constructor asigna al proceso el identificador *Generador*. Esta es información suficiente para iniciar la operación del sistema.

El método *run()* del generador de candidatos inicia enviando el número objetivo y el número de procesos del pipeline al primer filtro. Más adelante veremos que estos datos son distribuidos a través de todos los procesos, recibiéndolos del filtro anterior y enviándolos al filtro siguiente. Las líneas 17 y 18 envían al objeto *Master* la cantidad de milisegundos que han pasado desde cierta fecha establecida por los programadores de Java. Esto sirve para calcular el tiempo de ejecución del sistema, como veremos más adelante en esta misma sección. Entre las líneas 19 y 23 se determina el primer candidato que es enviado al primer filtro. En principio, el primer candidato está definido como la raíz cuadrada del número objetivo. Las líneas 28 a 34 calculan y envían al primer filtro del pipeline todos los candidatos de prueba. El último número enviado es -1, que es tomado como una señal de que no hay más candidatos.

El trabajo de los filtros de candidatos está definido por la clase *FiltersJob*, la cual se define como sigue:

```
1  public class FiltersJob extends RemoteJob{
2
3      int ident;
4      int sqrtMax;
5      int numProc;
6      int load;
7      int first;
8      int [] primeFilters;
9      int target;
10     int [] prime;
11
12     public FiltersJob(int id){
13         super("P"+id);
14         ident = id;
15         sqrtMax = 0;
16         numProc = 0;
17         load = 0;
18         first = 0;
19         target = 0;
```

```
20          primeFilters = null;
21      }
22    ...
23  }
```

Cada objeto *FiltersWork* recibe un identificador de tipo entero *ident*. Veremos más adelante que en un pipeline, es conveniente tomar identificadores de este tipo para los filtros. Sin embargo, como *RemoteJob*, cada *FiltersWork* también posee un identificador de tipo *String*, formado por la letra "P", seguida del identificador numérico *ident*. Por ejemplo, el proceso cuyo identificador numérico es 5, es visto en el sistema como el proceso *P5*. Las variable *target* y *numProc* son enviadas por el filtro anterior en el pipeline. *target* es el número antes del cual se desean obtener todos los números primos y *numproc* es el número de filtros de números primos. La variable *sqrtMax* se define como la raíz cuadrada de la variable *target*. El número entero *load* representa el número de primos de prueba para cada filtro, es decir, los divisores de los candidatos. La variable *first* es el primero de estos divisores, o sea el primer número primo de este filtro contra el cual se verifican los candidatos. Dos arreglos de números primos son necesarios: el arreglo *primeFilters[]* contiene a todos los números primos menores que *sqrtMax*, mientras que el arreglo *prime[]* contiene solo a aquellos números primos que este filtro utiliza para verificar a los candidatos.

El trabajo principal de los filtros se realiza en el método *filterNumbers()*, cuya implementación es la siguiente:

```
1  public void filterNumbers(){
2      int count = 0;
3
4      for(int i = 0; i < primeFilters.length; i++){
5          if(primeFilters[i] > 0){
6              count++;
7          }
8          else{
9              break;
10         }
11     }
12     if(count - first >= load){
13         prime = new int[load];
14     }
15     else{
16         prime = new int[count - first];
17     }
18     for(int i = 0; i < prime.length; i++){
19         prime[i] = primeFilters[first + i];
20     }
```

```
21
22      int next = receiveNext();
23      while(next > 0){
24          int index = 0;
25          while(index < prime.length){
26              if(next % prime[index] != 0){
27                  index++;
28              }
29              else{
30                  break;
31              }
32          }
33          if(index == prime.length){
34              sendNext(next);
35          }
36          next = receiveNext();
37      }
38      sendNext(next);
39  }
```

Este método toma la idea del código 5.7. Entre las líneas 2 y 20, el método *filterNumbers()* determina y guarda los números primos con los que debe comparar a los candidatos en el arreglo *prime[]*. El resto del método recibe candidatos del filtro anterior y los compara con los primos del arreglo *prime[]*. El número candidato es enviado al siguiente filtro solo si pasa exitosamente todas las divisiones tal como en el código 5.7. Los métodos *sendNext()* y *receiveNext()* deducen en base al identificador del filtro, a quien deben enviar o pedir datos; por ejemplo, el primer filtro del pipeline recibe candidatos del proceso *Generador*, mientras que el último filtro envía datos al objeto *Master*.

El trabajo del método *mainWork()* del código 5.8 se toma como base para crear el método *run()* de la clase *FiltersJob* como sigue:

```
1   public void run(){
2           String sender;
3
4           if(this.ident == 1){
5               sender = "Generador";
6           }
7           else{
8               sender = "P" + (ident-1);
9           }
10
11          numProc = (Integer) receiveMessage(sender);
12          target = (Integer) receiveMessage(sender);
13          sqrtMax = (int) floor(sqrt(target));
14
```

```
15          if(sqrtMax % 2 == 0){
16              sqrtMax++;
17          }
18
19          if(ident != numProc){
20              String rec = "P" + (ident+1);
21              sendMessage(rec, numProc);
22              sendMessage(rec, target);
23          }
24
25          findLoad();
26
27          if(ident == numProc){
28              int index = 0;
29              int prim = primeFilters[index];
30
31              while(prim > 0){
32                  saytoMaster(prim);
33                  index++;
34                  prim = primeFilters[index];
35              }
36          }
37
38          if(load != 0){
39              filterNumbers();
40          }
41          else{
42              boolean gate = true;
43
44              while(gate){
45                  int next = receiveNext();
46
47                  if(next < 0){
48                      gate = false;
49                  }
50                  sendNext(next);
51              }
52          }
53          if(ident == numProc){
54              long time2 = System.currentTimeMillis();
55              saytoMaster(time2);
56          }
57          finished();
58  }
```

Sin embargo, hay diferencias notables que debemos señalar. Para empezar, omitimos el uso de un buffer para almacenar los candidatos recibidos. Esto es debido a que en la biblioteca J-MIPS, los mensajes recibidos son almacenados en buffers

que permiten un esquema de conexión de tipo "emisor no bloqueante - receptor bloqueante", que es el esquema deseado al incluir buffers para los filtros. Además, dado que cada filtro tiene un identificador, no es necesario que los procesos reciban un identificador por parte del filtro anterior como en el código 5.8.

El método *run()* de la clase *FiltersJob* considera además trabajo extra para el último filtro del pipeline, el cual entre las líneas 27 y 36, envía los primeros números primos menores que $sqrt(target)$ al objeto *Master* para ser desplegados al usuario. Después de haber probado al último candidato, el último filtro envía al *Master* la hora en la que el trabajo ha terminado en la línea 55.

También es considerado en el método *run()* el caso en el que el número de primos de prueba sea cero para el filtro actual. En este caso, la tarea se convierte en recibir números desde el filtro anterior y enviarlos al siguiente sin ningún cambio. Esta tarea se especifica entre las líneas 41 y 52.

Finalmente, si el número de primos de prueba para el filtro actual es distinto de cero, se ejecuta el método *filterNumbers()* en la línea 39.

Managers y Master

Una vez definidos los procesos, debemos especificar el trabajo de las computadoras virtuales *Manager* y de la computadora virtual *Master*. En general, en todas las aplicaciones paralelas programadas mediante J-MIPS, el trabajo de los *Managers* es muy similar al mencionado en el ejemplo de la sección 4.8; es decir, los *Managers* solo deben esperar conexión desde el objeto *Master*.

```
1  public class ManagerMain {
2
3      public static void main(String [] argv){
4          Manager m = new Manager(puerto,timeout);
5
6          try{
7              m.connectToMaster();
8          }
9          catch(ConnectionException e){
10             e.printStackTrace();
11         }
12     }
13 }
```

Para la computadora *Master*, el método *main()* de la clase que la ejecuta es como sigue:

```
1  public static void main(String [] argv){
2      int nprocs = 7;
3      int target = 10000;
4      Master m = new Master("Config.cfg",10000,200);
5
6      try{
7          for(int i = 1; i <=nprocs;i++){
8              FiltersJob fw = new FiltersJob(i);
9              m.addRemoteJob(fw);
10         }
11         NGenerator ng = new NGenerator(nprocs,target);
12         m.addRemoteJob(ng);
13         Thread.sleep(2000);
14         m.startSystem();
15         boolean flag = true;
16
17         while(flag){
18             int rec = (Integer) m.messageFrom("P" + nprocs);
19             if(rec > 0){
20                 System.out.println("Es␣primo␣" + rec);
21             }
22             else{
23                 flag = false;
24             }
25         }
26         long time1 = (Long) m.messageFrom("Generador");
27         long time2 = (Long) m.messageFrom("P" + nprocs);
28         long tTime = time2 - time1;
29
30         System.out.println("Tiempo␣" + tTime + "␣milisegundos");
31         m.shutdown();
32     }
33     catch(Exception e){
34         e.printStackTrace();
35     }
36 }
```

Este método inicia definiendo las variables *nprocs* y *target*, que son el número de filtros en el pipeline y el número objetivo para el cual se deben obtener todos los primos menores que él. En la línea 4 se crea un objeto *Master* cuyo archivo de configuración es llamado "Config.cfg"[3] y entre las líneas 7 y 10 se le proporcionan procesos *FiltersJob* con identificadores del 1 al 10. Las líneas 11 y 12 agregan al objeto *Master* el proceso generador de candidatos en el pipeline. La instrucción *startSystem()* que inicia la operación del programa paralelo se encuentra en la línea 14, precedida por un pequeño tiempo de espera que permite que los *Ma-*

[3]Se asume la existencia de dicho archivo, del cual se habla más adelante en esta sección.

nagers inicien su ejecución correctamente antes que el objeto *Master*. Entre las líneas 17 y 25, se imprimen en pantalla todos los números primos menores que *target* leyendo datos del buffer de mensajes del último filtro del pipeline. El ciclo se rompe cuando se lee un número negativo (el único de estos es -1, indicando que se han acabado).

Debemos recordar que el generador de candidatos envío al objeto *Master* la hora exacta (en milisegundos) en que inició su ejecución. De igual forma, el último filtro envió la hora en milisegundos en la que terminó su trabajo. En las líneas 26 y 27 se recuperan estos datos del objeto *Master* y se obtiene el tiempo de ejecución del sistema en la línea 28. Después de imprimir el tiempo de ejecución en la pantalla, se termina la ejecución del sistema mediante la llamada *m.shutdown()*.

El archivo de configuración

En la sección pasada asumimos la existencia de un archivo de configuración llamado *Config.cfg*. En esta sección especificamos dicho archivo. Como antes, disponemos de un cluster de cinco computadoras donde todas poseen una Maquina Virtual de Java versión 1.5 o superior con las siguientes direcciones y puertos:

Direccion IP	Puerto
192.168.1.100	3305
192.168.1.101	3305
192.168.1.102	3305
192.168.1.103	3305
192.168.1.104	3305

Al igual que en el ejemplo anterior, utilizamos un *Manager* por cada computadora disponible, que no sea donde reside la computadora virtual *Master*, la cual tiene la dirección 192.168.1.104. En las cuatro computadoras restantes, ejecutamos un *Manager* en cada una de ellas con los siguientes identificadores:

Identificador	Direccion IP	Puerto
M101	192.168.1.100	3305
M102	192.168.1.101	3305
M103	192.168.1.102	3305
M104	192.168.1.103	3305

Al analizar el trabajo de los procesos en el pipeline, es fácil darnos cuenta de que los últimos filtros tienen menos trabajo que los primeros. Esto es debido a que cada filtro considera igual número de primos de prueba y a que muchos candidatos son eliminados del flujo de datos que atraviesa el pipeline en los primeros filtros.

Para equilibrar un poco la carga de trabajo, creamos siete filtros mas el proceso generador distribuidos de la siguiente forma:

Identificador	Manager
Generador	M101
P1	M102
P2	M103
P3	M103
P4	M104
P5	M104
P6	M104
P7	M104

Es decir, el proceso generador y el primer filtro del pipeline se ejecutan en un *Manager* cada uno. El siguiente *Manager* en el pipeline se encarga del doble de primos de prueba al ejecutar dos filtros. El último *Manager* maneja cuatro veces más primos de prueba que el primero al ejecutar cuatro filtros. La figura 5.8 ilustra mejor la explicación que hemos proporcionado.

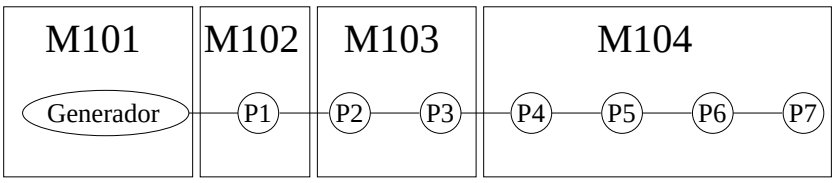

Figura 5.8: Mapeo de los filtros a los *Managers*.

Elegimos este mapeo simplemente para mostrar al lector la flexibilidad que J-MIPS permite al asignar procesadores a los procesos. Con estas especificaciones, el archivo de configuración queda como sigue:

```
1   #Hosts
2  Host = (192.168.1.100,3305,M101)
3  Host = (192.168.1.101,3305,M102)
4  Host = (192.168.1.102,3305,M103)
5  Host = (192.168.1.103,3305,M104)
6
7  #Processes
8  Process = (Generador,M101,[P1])
9  Process = (P1,M102,[P2,Generador])
10 Process = (P2,M103,[P1,P3])
```

```
11  Process = (P3,M103,[P2,P4])
12  Process = (P4,M104,[P3,P5])
13  Process = (P5,M104,[P4,P6])
14  Process = (P6,M104,[P5,P7])
15  Process = (P7,M105,[P6])
```

5.3. Resumen del capítulo

En esta sección se revisan dos ejemplos de programas paralelos distribuidos que utilizan J-MIPS como plataforma de ejecución. Primero se revisa el problema del *camino mínimo con pesos positivos*, el cual utiliza una estructura de hipercubo para conectar procesos. Después se revisa el problema de la criba de Eratóstenes que encuentra de forma paralela todos los números primos menores que cierto numero natural. La solución a este problema utiliza el patrón de filtros y tuberías paralelas.

La implementación de estos programas en J-MIPS demuestra su capacidad para mapear e interconectar procesos, así como para definir distintas topologías de conexión entre ellos.

Mostramos también el método que se sigue para implementar un programa paralelo distribuido mediante J-MIPS. El primer paso es definir los procesos que deben ser ejecutados. La tarea de los *Managers* es siempre la misma: esperar comunicación del objeto *Master*. Para esta última, primero se crea una instancia de *Master*, se le agregan procesos, se pone en ejecución el sistema y se leen los mensajes provenientes de ellos. Finalmente, el archivo de configuración siempre sigue el mismo orden para mapear los procesos a las computadoras físicas de la red: se definen los *Managers* y luego los procesos con un *Manager* asignado a cada uno y una lista de conexiones.

Conclusiones

En este capítulo, concluimos el trabajo de tesis. Comenzamos dando un breve resumen sobre las características más importantes de la biblioteca J-MIPS.

El capítulo continua con una revisión de J-MIPS. Como un sistema paralelo en memoria distribuida, podemos revisar sus características conforme a las mismas pautas con las que revisamos a las bibliotecas PVM y MPI. Esta revisión nos permite detectar las virtudes y limitaciones de J-MIPS respecto a otros sistemas paralelos de memoria distribuida.

Después, mencionamos la experiencia del escritor de este trabajo al utilizar J-MIPS para la ejecución de aplicaciones paralelas.

Mencionamos entonces algunas características del sistema J-MIPS que contribuyen al área del cómputo paralelo en memoria distribuida.

Finalizamos abordando algunas características que no son tratadas en esta tesis, pero que pueden ser abordadas en el futuro para complementar lo que hemos hecho hasta ahora.

Resumen del sistema

Hemos revisado el desarrollo de J-MIPS. Esta biblioteca, implementada en Java, permite a otros programadores desarrollar aplicaciones paralelas distribuidas en un clúster de computadoras. Los elementos principales del sistema son los siguientes:

- La computadora virtual *Master*, encargada de iniciar los componentes del sistema, mapear y conectar los procesos del programador y supervisar la ejecución de las computadoras virtuales que los ejecutan. La configuración de las conexiones entre procesos y el mapeo de éstos a los *Managers* es proporcionado por el programador al objeto *Master* mediante un archivo

de configuración. Además, el objeto *Master* mantiene comunicación con el usuario para entregarle los mensajes que provengan de los procesos en ejecución.

- Las computadoras esclavas virtuales o *Managers*, quienes ejecutan los procesos del usuario. Los *Managers* mantienen comunicación con otras computadoras virtuales esclavas para permitir la comunicación entre procesos. La característica más importante de este componente de J-MIPS es que cuando son creados, los *Managers* no tienen conocimiento sobre las conexiones que deben tener con otros *Managers*, no conocen los procesos que van a ejecutar y ni siquiera conocen su propio identificador. Esta información es proporcionada por la computadora virtual *Master*.

- Los procesos *RemoteJob* del programador que en principio son proporcionados al nodo maestro virtual *Master* y que después son mapeados por éste a los *Managers* de acuerdo al archivo de configuración del usuario. Estos procesos contienen primitivas de comunicación por paso de mensaje, que permiten al programador intercambiar información entre procesos que se ejecutan en computadoras físicas distintas.

Visto como un todo, un sistema J-MIPS es una red de computadoras virtuales (*Managers*) que ejecutan sus propios procesos (*RemoteJob*), los cuales intercambian información entre ellos. Las computadoras virtuales son administradas por un nodo virtual maestro (*Master*). Esta red es mapeada a una red física de computadoras. Sin embargo, la red virtual mantiene características topológicas propias e independientes de las de la red física.

Comparación con trabajo relacionado

Portabilidad

La portabilidad es una de las características de J-MIPS. A diferencia de PVM y MPI, en J-MIPS no es necesario recompilar un programa paralelo para ejecutarlo en otra arquitectura. Basta con ajustar el archivo de configuración para ejecutar el mismo programa proveniente de otro sistema. El único requisito es que el nuevo sistema posea en todas sus computadoras una Maquina Virtual de Java.

Comunicación entre procesos

Al igual que en PVM y MPI, J-MIPS utiliza paso de mensajes para enviar información de un proceso a otro. Sin embargo, a diferencia de PVM, la tarea de

empacar el mensaje e inicializar un buffer para enviarlo es realizada automáticamente por J-MIPS. El único requisito en este sentido, es que el dato que se desea enviar debe ser de tipo *Serializable*.

A diferencia de MPI y PVM, J-MIPS no posee primitivas que envíen un mensaje y esperen por una respuesta. Esta tarea debe ser programada por el usuario de J-MIPS.

Control de procesos

Este es quizá uno de los aspectos más débiles de J-MIPS frente a MPI y PVM. J-MIPS asume que la cantidad de procesos en el sistema durante toda su ejecución es estático. J-MIPS es incapaz de crear, iniciar o detener procesos en tiempo de ejecución. Además, salvo los mensajes recibidos en la computadora virtual *Master*, enviados por los procesos en ejecución en computadoras remotas, J-MIPS no tiene forma de obtener datos relacionados con la ejecución de los procesos, asumiendo simplemente que en algún momento van a terminar su ejecución, y cuando lo hagan, los *Managers* que los ejecutan avisen a la computadora virtual *Master*.

MPI y PVM pueden manejar procesos de una forma mucho más dinámica, permitiendo al programador crear, iniciar y detener procesos dinámicamente en el sistema. Sin embargo, J-MIPS permite especificar exactamente qué computadora ejecuta cual proceso, característica de la cual carece PVM.

Control de recursos

En este sentido, J-MIPS es muy parecido a MPI y distinto de PVM. J-MIPS fue pensado para ser de naturaleza estática respecto al control de recursos. Esto quiere decir que no existen funciones en J-MIPS para agregar o eliminar dinámicamente una computadora a la red y para reasignar un proceso a otro procesador.

PVM es mucho más dinámico en este sentido, pues permite agregar o eliminar computadoras a la red en cualquier momento, ya sea desde una consola de sistema o desde la misma aplicación paralela en ejecución.

Topología

Al igual que MPI, J-MIPS provee un alto nivel de abstracción en términos de la topología de paso de mensajes. J-MIPS considera tres niveles topológicos:

- Topología de procesos. Mediante paso de mensajes, los procesos pueden ser acomodados para interactuar dentro de una topología especificada a nivel

lógico. Así, la comunicación entre procesos se da dentro de la topología especificada, aunque ésta no corresponda con la topología física de la red.

- Topología de *Managers*. Al definir las conexiones entre procesos y el mapeo de éstos a las computadoras virtuales esclavas, se define indirectamente la topología de los *Managers*. Esta se da en un nivel lógico y es independiente de la topología física de la red. La red de *Managers* es determinada por la computadora virtual *Master* a partir del archivo de configuración.

- Topología física. Finalmente, los *Managers* son mapeados a la red física de computadoras, la cual se asume como estática.

En contraste, PVM no soporta la especificación de una topología virtual de interconexión de procesos. En PVM, el usuario crea grupos de procesos que deben ser ejecutados y especifica la organización de las comunicaciones.

Tolerancia a fallas

MPI y PVM soportan un sistema de notificación de falla de procesos. Esto es, cuando un proceso A espera un mensaje por parte de otro proceso B, y éste muere súbitamente debido a una falla, A recibe una notificación de falla en vez del mensaje original. A puede decidir entonces qué hacer.

Además, dado el alto grado en el control de recursos de PVM, ante una falla en un proceso, la carga de trabajo puede reasignarse y el sistema puede continuar su ejecución.

J-MIPS no posee un sistema de notificación como MPI y PVM. Además, el grado de control de recursos y procesos de J-MIPS impide la reasignación de trabajo cuando un fallo ocurre. Cuando un proceso *RemoteJob* falla, es responsabilidad del programador manejar el fallo desde otros procesos. Cuando un *Manager* falla, la computadora virtual *Master* detecta el error y detiene la ejecución del sistema completo. De igual forma, cuando la computadora virtual *Master* falla, los *Managers* detectan el error y detienen su ejecución.

Contribuciones

Como mencionamos al principio de esta tesis, nuestro trabajo permite desarrollar aplicaciones paralelas distribuidas. Otras bibliotecas como MPI y PVM han sido desarrolladas con el mismo propósito. Sin embargo, creemos que J-MIPS puede ser utilizada como una herramienta alternativa con características que facilitan la

programación y ejecución de sistemas distribuidos. Particularmente, los siguientes puntos pueden tomarse en cuenta:

- Las aplicaciones implementadas con J-MIPS alcanzan un alto grado de portabilidad, obtenida a partir de la Java Virtual Machine (JVM). Cada computadora física de la red debe ejecutar su parte del sistema distribuido a través de una JVM. Esto permite al programador migrar fácilmente sus aplicaciones entre una red física y otra, siempre y cuando los componentes de la nueva red posean una JVM.

- El mapeo de los procesos está dado por el programador. Lo cual es una característica muy importante, pues éste decide en todo momento en dónde se ejecuta un determinado proceso.

- El programador construye la topología virtual. Lo cual es una consecuencia del punto anterior, pues al tener el usuario la capacidad de decidir dónde se ejecutan sus procesos, también decide la topología de la red de computadoras virtuales.

- Es fácil modificar la configuración del sistema. A través del archivo de configuración, el usuario puede decidir cuántas computadoras físicas tiene a su disposición, así como especificar el mapeo de los procesos y las conexiones entre ellos. Los cambios en estos aspectos son especificados modificando únicamente el archivo de configuración, en el cual, tanto los procesos como las computadoras virtuales son definidos mediante tuplas de datos simples. Estos aspectos facilitan la tarea de modificar la configuración del sistema, tanto en el mapeo de los procesos como en la construcción de la red virtual.

- J-MIPS permite enfocarse en la definición del trabajo que debe realizarse. Una vez dado el archivo de configuración y los procesos que deben ser ejecutados, la computadora virtual *Master* se encarga de construir la red virtual sobre la red física y de poner en ejecución los procesos tal y como se especifica en el archivo. Esto permite al programador de aplicaciones paralelas enfocarse en la implementación de los procesos que deben ser ejecutados, o por lo menos, separar los dos problemas que surgen al programar aplicaciones distribuidas: la construcción de la red y la implementación de los procesos que se ejecutan en ella.

Experiencia

Nuestra experiencia al utilizar J-MIPS para ejecutar aplicaciones paralelas en memoria distribuida se limita a los ejemplos de aplicación descritos en el capítulo 5

y a algunas pruebas involucradas en el desarrollo de la biblioteca, utilizando una pequeña red de computadoras descrita en el capítulo 5.

En el capítulo dedicado a los ejemplos de aplicación, sin embargo, mostramos solo un mapeo por cada ejemplo. En el desarrollo de este trabajo, ejecutamos los ejemplos utilizando distintos mapeos de procesos. Para llevar a cabo esta tarea, no tuvimos que realizar modificaciones en el código de Java que describe el comportamiento de la aplicación paralela (los objetos *RemoteJob*, la ejecución de los *Managers* y la ejecución del objeto *Master*). Solo fue necesario modificar el archivo de configuración para cambiar el mapeo de los procesos a los *Managers* y de los *Managers* a las computadoras físicas de la red. Podemos decir que esta es una característica de todas las aplicaciones paralelas ejecutadas en J-MIPS, lo cual significa que una vez construida la aplicación paralela, no es necesario recompilar el código de la aplicación para modificar la configuración del sistema. Más aún, una vez teniendo la especificación de los procesos *RemoteJob*, el resto de la construcción de aplicaciones paralelas sigue un patrón fácil de implementar:

- **Construir el programa que ejecuta a los *Managers*.** Sin embargo, como vimos en los ejemplos de aplicación, este programa puede ser el mismo para todas las aplicaciones paralelas.
- **Construir el programa que ejecuta al objeto *Master*.** El cual, también constituye un patrón de instrucciones sencillo: Crear los procesos, crear el objeto *Master*, alimentarlo con los procesos creados, llamar a la ejecución del sistema y recopilar datos de la ejecución para desplegarlos al usuario.

En resumen, en nuestra experiencia J-MIPS nos ha permitido enfocarnos en la construcción de los procesos que intervienen en las aplicaciones paralelas, ofreciendo además reutilización de código entre ellas.

Trabajo futuro

J-MIPS no es una biblioteca completa. En realidad, podemos decir que cumple con los componentes básicos que nos permiten alcanzar nuestro objetivo: mapear y conectar procesos de software. Aún existen muchas características que pueden ser mejoradas o añadidas a J-MIPS para hacerla una plataforma más eficiente.

Una característica que consideramos importante añadir a J-MIPS es el manejo dinámico de procesos. Hasta ahora, solo los procesos especificados inicialmente en el objeto *Master* pueden ser ejecutados en el sistema. Ningún proceso puede ser creado o destruido durante la ejecución de un programa paralelo. Creemos que es

deseable que el sistema tenga la capacidad para crear nuevos procesos y mapearlos a una computadora virtual, lo cual implica reevaluar las conexiones establecidas inicialmente entre los *Managers*.

El manejo de excepciones en J-MIPS puede ser mejorado para evitar que el sistema completo se detenga cuando existe un error en alguno de sus componentes. Con el sistema desarrollado en este trabajo, un error producido en un *Manager* provoca que el objeto *Master* detenga la ejecución de los *Managers* restantes. Es deseable que ante un error así, el sistema pueda redistribuir el trabajo del *Manager* defectuoso mientras éste es reparado.

En el objeto *Master*, al igual que en los *Managers*, por cada conexión con otra computadora virtual, un proceso es creado para supervisar los mensajes que lleguen desde esa conexión, para así evitar tiempos de espera en la ejecución del sistema. Sin embargo, cada uno de estos procesos consume tiempo del procesador, disminuyendo el desempeño general de un programa en ejecución. Creemos que es posible mejorar esta situación sustituyendo a los procesos supervisores por oyentes de eventos o interrupciones cuando un mensaje es recibido.

Estas son solo algunas de las características que pueden ser mejoradas en J-MIPS, las cuales consideramos más importantes. Sin embargo, J-MIPS es una biblioteca en desarrollo y falta mucho por hacer para convertirla en una biblioteca completa. Sin embargo, consideramos que nuestro propósito y objetivo han sido cumplidos con el desarrollo que hemos hecho hasta ahora. Así, el desarrollo subsecuente de J-MIPS puede ser abordado en otros trabajos.

Bibliografía

[1] Barney, Blaise «*Introduction to parallel computing*», https://computing.llnl.gov/tutorials/parallel_comp/#ModelsThreads.

[2] Bokhari, Shahid H. «*On the Mapping Problem*», IEEE Transactions on computers, vol. c-30, No. 3, pages 207-214, March 1981.

[3] Foster, Ian «*Designing and Building Parallel Programs: concepts and tools for parallel software engineering*», Addison-Wesley, 1995.

[4] Geist, G. A.; Kohl, J. A.; Papadopoulos, P. M. «*PVM and MPI: a comparison of features*», http://www.csm.ornl.gov/pvm/PVMvsMPI.ps, 1996.

[5] Geist, Al; Beguelin, Adam; Dongarra, Jack; Jiang, Weicheng; Manchek, Robert; Sunderam,, Viady «*PVM: Parallel Virtual Machine. A users' guide and tutorial for network parallel computing*», The MIT Press, 5a ed., 2000.

[6] Horstmann, Cay S.; Cornell, Gary «*Core Java 2, Volumen I - Fundamentos*», 7a ed., Prentice Hall, 2006.

[7] Horstmann, Cay S.; Cornell, Gary «*Core Java 2, Volumen II - Características Avanzadas*», 7a ed., Prentice Hall, 2006.

[8] Juhasz, Zoltan; Turner, Stephen J. «*A new Heuristic for the Process-Processor Mapping Problem*», Distributed and parallel systems: from instruction parallelism to cluster computing, Kluwer Academic Publishers, pages 91-94, 2000.

[9] Nevison, Christopher H.; Hyde, Daniel C.; Schneider, G. Michael.; Tymann, Paul T. «*Laboratories for Parallel Computing*», 1a ed., Jones and Bartlett Publishers, 1994.

[10] Ortega Arjona, Jorge Luis «Architectural Patterns for Parallel Programming: Models for Performance Estimation», VDM Verlag, 2009.

[11] Ortega Arjona, Jorge Luis «*Design Patterns for Communication Components of Parallel Programs*», 12th European Conference on Pattern Languages of Programs, 2007.

[12] Ortega Arjona, Jorge Luis «*The parallel layers pattern, A Functional Parallelism Architectural Pattern for Parallel Programming.* », 6th Latin American Conference on Pattern Languages of Programming, 2007.

[13] Ortega Arjona, Jorge Luís «*The Parallel Pipes and Filters Pattern, A Functional Parallelism Architectural Pattern for Parallel Programming*», 10th European Conference on Pattern Languages of Programming and Computing, 2005.

[14] Parnot, Charles «*Xgrid Stanford - Using Xgrid to fit biochemical models*» , [en linea], Stanford University, [ref. del 3 de enero de 2012], Disponible en: *http://cmgm.stanford.edu/c̃parnot/xgrid-stanford/.*

[15] Peterson, Larry L.; Davier, Bruce S. «*Computer Networks, a system approach*», 4a ed., Elsevier, 2007.

[16] «*Concurrencia en java*», [en linea], Posgrado en Ciencias e Ingeniería de la Computación, [ref. 15 de marzo de 2011], Disponible en *http://www.matematicas.unam.mx/jloa/concurrencia.pdf.*

[17] Silverschatz, Abraham; Galvin, Peter Baer; Gagne, Greg «*Operating System Concepts*», 8a ed., John Wiley and sons, 2009.

[18] Snir, Marc; Otto, Steve; Huss-Lederman, Steven; Walker, David; Dongarra, Jack «*MPI: The Complete Reference, Volume 1 - The MPI Core*», 2a. ed., The MIT Press, 1996.

[19] Stallings, William «*Data and computer communications*», 7a ed., Prentice Hall, 2004.

[20] Stallings, William «*Sistemas operativos, aspectos internos y principios de diseño*», 5a ed., Prentice Hall, 2005.

[21] Tanenbaum, Andrew S. «*Modern operating systems*», 3a ed., Prentice Hall, 2008.

[22] Tanenbaum, Andrew S. «*Redes de computadoras*», 4a ed., Prentice Hall, 2003.

[23] Tanenbaum, Andrew S.; Van Steen Maarten «*Sistemas distribuidos, principios y paradigmas*» 2a ed., Prentice Hall, 2008.

[24] «*Xgrid Programming Guide: How It Works*», [en linea], Apple Inc., [ref. del 3 de enero de 2012], Disponible en: *http://developer.apple.com/library/mac/#documentation/MacOSXServer/Conceptual/Xgrid_Programming_Guide/Overview/Overview.html#//apple_ref/doc/uid/TP40006246-CH2-DontLinkElementID_13*.