UNIVERSIDAD NACIONAL AUTONOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DESARROLLO DE UN API PARA CÓMPUTO PARALELO

TESIS

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTA:
JOSÉ DANIEL RODRÍGUEZ HERRERA

DIRECTORA:
ING. LAURA SANDOVAL MONTAÑO

CIUDAD UNIVERSITARIA MÉXICO, D.F. 2006

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

ÍNDICE

PRÓLOGO.......................................1

INTRODUCCIÓN..................................3

CAPÍTULO I **SUPERCÓMPUTO**.....................10

INTRODUCCIÓN.............................11

ARQUITECTURAS DE COMPUTADORAS.............12

CONDICIONES DE CARRERA...................22

GRANULARIDAD.............................24

MODELOS DE CONEXIÓN DE REDES.............27

ALGORITMOS...............................35

CAPÍTULO II **PASO DE MENSAJES**................43

INTRODUCCIÓN.............................44

CLUSTERS.................................45

FUNDAMENTOS TEÓRICOS DE PASO DE MENSAJES..51

PASO DE MENSAJES ASÍNCRONO...............58

PASO DE MENSAJES SÍNCRONO................62

CONDICIONES DE EXCEPCIÓN.................70

CAPÍTULO III **DESARROLLO**.....................73

PROCESO DE DESARROLLO....................74

ANÁLISIS.................................75

DISEÑO...................................86

CONSTRUCCIÓN.............................96

CAPÍTULO IV **PRUEBAS**........................105

APLICACIÓN DE EJEMPLO...................106

DESCRIPCIÓN DEL PROGRAMA................107

LISTADO COMPLETO........................113

CONCLUSIONES................................116

BIBLIOGRAFÍA................................118

PRÓLOGO

El cómputo paralelo ha evolucionado hasta llegar a, lo que a la fecha es, miles de procesadores trabajando como uno solo, sistemas de memoria compartida que a su vez forman parte de sistemas más grandes de memoria distribuida, todo esto para ejecutar programas que tal vez requieren un poder de cómputo mayor.
La evolución no fue rápida, pero definitivamente se presentó; por desgracia no todo lo relacionado con el súper cómputo evolucionó de la misma manera, como casi siempre sucede, el software se quedó detrás del hardware, los lenguajes que se usan para programas secuenciales son ya muy distintos a los que se usan para el cómputo paralelo, uno de los casos más concretos, es el de Java, el cual ha ido tomando terreno y hoy por hoy es usado por todas las empresas que se dedican al desarrollo del software, al grado que los servidores de aplicaciones se están desarrollando en Java, a pesar de ser tan popular en los programas secuenciales, en el cómputo paralelo no se usa. Una de las razones, es el hecho de ser muy reciente, los programas en paralelo se hacen mediante bibliotecas propietarias, que en su mayoría son caras, o en su defecto tomaron mucho tiempo en ser desarrolladas y son complicadas de instalarse y usarse.

Las ventajas de Java son bastantes, si bien no todas son relevantes para el cómputo paralelo, la gran mayoría sí lo son, por lo que el desarrollo de bibliotecas que permitan hacer programas en paralelo usando Java toma

todas las ventajas del lenguaje y una más sobre todos los demás, Java es muy popular y la gente que puede realizar programas en paralelo, no necesita ser un programador experto en las bibliotecas MPI, PVM o en Open MP, basta con que apliquen sus conocimientos en Java, no tienen que aprender un lenguaje nuevo y complejo que fue diseñado hace 20 años, así como adecuarse a un nuevo sistema operativo para poder usar las herramientas; esto es una excelente opción para todos los científicos e investigadores que en ocasiones no pueden sacar provecho de equipos grandes, con un poder de cómputo considerable, lo que repercute en un atraso directo en sus investigaciones.

No sólo los investigadores pueden tomar ventaja, la gente que se dedica a las ciencias computacionales puede enriquecer cada vez más las bibliotecas con nuevas rutinas, cada vez más optimizadas y robustas, o bien bibliotecas específicas para casos de investigación especiales, el objetivo de todo esto es dejar de reinventar la rueda cada que se comienza un proyecto nuevo, es decir enfocarse al proyecto y no a las herramientas.

Un ejemplo de aplicaciones que requieren un gran poder de cómputo, es el procesamiento digital de imágenes, por lo que es un excelente medio para probar todo lo que se ha mencionado y poder ver con pruebas las ventajas antes mencionadas así como las que se verán de manera más extensa.

INTRODUCCIÓN

Los desarrollos de supercómputo han sido motivados por las simulaciones numéricas de sistemas complicados como el clima, los dispositivos mecánicos, los circuitos electrónicos y reacciones químicas. Sin embargo, lo que manda el desarrollo de este tipo de operaciones que están apareciendo, son las aplicaciones comerciales que requieren que una computadora pequeña pueda procesar las cantidades de datos y operaciones como computadoras grandes más rápidas.

Otra tendencia importante que cambia la cara de la computación es el aumento en las capacidades de las redes que conectan computadoras.

Una computadora de Von Neumann comprende una unidad de procesamiento central (CPU) conectada con una unidad de almacenamiento (la memoria) . La CPU ejecuta un programa almacenado que especifica una secuencia de la lectura y las operaciones de escritura sobre la memoria. Este modelo simple ha probado ser robusto. Su perseverancia sobre más de cuarenta años ha permitido que el estudio de tales temas importantes como los algoritmos y los lenguajes de programación, siga en gran parte independiente de los desarrollos en la arquitectura de computadora.

El modelo de computadora debe ser tanto simple como objetivo: simple para facilitar, comprender y programar, y objetivo para asegurar que los programas desarrollados para el modelo se ejecutan con la eficiencia razonable.

Un modelo de máquina paralelo es en la que cada computadora ejecuta su propio programa. Este programa puede acceder a la memoria local y puede enviar y recibir los mensajes sobre la red. Los mensajes permiten leer y escribir memorias lejanas. En la red idealizada, el costo de enviar un mensaje entre dos nodos es independiente tanto de la ubicación de nodo como el tráfico de la red, pero depende de la duración del mensaje.
Un atributo definiendo el modelo es que los accesos para la memoria en el mismo nodo local son menos costosos que los accesos para la memoria en diferente nodo. Es decir lectura y escritura son menos costosas que el envío y recepción.

MIMD

La computadora de MIMD (múltiples instrucciónes múltiples datos) quiere decir que cada procesador puede ejecutar un proceso separado del flujo de instrucciones sobre sus propios datos; la memoria distribuida quiere decir que la memoria es distribuida entre los procesadores. La diferencia principal entre una computadora de varios procesadores y un *cluster* en MIMD es que en el último, el costo de enviar un mensaje entre dos nodos no podría ser independiente de la ubicación de nodo y la red de tráfico.

Otra clase importante de computadora paralela es la multiprocesador, o la computadora MIMD de memoria compartida. En la cual, todos los procesadores comparten el acceso para una memoria común, típicamente vía un bus o una jerarquía de buses. En el modelo de acceso aleatorio paralelo idealizado, que a menudo se usa en estudios teóricos de los algoritmos paralelos, cualquier procesador puede acceder a cualquier elemento de memoria en la misma cantidad de tiempo. Para acceso, el caché es mucho más rápido que el acceso a la memoria compartida; por lo tanto, la localidad es generalmente importante, y las diferencias entre *clusters* y supercomputadoras son realmente sólo las cuestiones de velocidad y precio. Programas desarrollados para *clusters* también pueden ejecutar eficientemente sobre supercomputadoras, porque la memoria compartida permite una puesta en funcionamiento eficiente del paso de mensaje.

Lo anterior es la base de la tesis propuesta ya que la aplicación que se desarrolla puede ser adaptada para funcionar en supercomputadoras.

Existen varios modelos de programación paralela, siendo diferente en su flexibilidad, mecanismos de interacción de tarea, granularidad de tarea, y soporte para localidad, escalabilidad, y la modularidad. Aquí, examinamos algunas alternativas.

ENVÍO DE MENSAJES

El paso de mensajes es probablemente lo más ampliamente usado para la programación paralela. Cada tarea es identificada por un nombre único, y las tareas interactúan transmitiendo y recibiendo los mensajes hacia y desde las tareas nombradas. Con respecto a esto, el paso de mensaje es realmente sólo una diferencia menor sobre el otro modelo llamado de tarea-canal, siendo diferente solamente en el mecanismo usado para la transferencia de datos, más que enviar un mensaje de la tarea sobre el canal, podemos enviar un mensaje por ejemplo a la tarea número dos.

El modelo paso de mensaje no impide la creación dinámica de las tareas, la ejecución de las tareas múltiples por procesador, o la ejecución de programas diferentes por las tareas diferentes. Sin embargo, en la práctica la mayoría de los sistemas de paso de mensaje crean un número fijo de las tareas idénticas en la nueva solicitud del programa y no permiten que las tareas sean creadas o destruidas durante la ejecución del programa. Este tipo de programas van a implementar un programa modelo de programación de datos (SPMD) múltiples datos para un simple programa, esto quiere decir que la tarea ejecuta el mismo programa pero afecta los múltiples datos diferentes. Esto dificulta algunos desarrollos de algoritmo paralelos.

La tarea del ingeniero de software es diseñar e implementar programas que satisfacen al usuario sus requisitos para el uso, corrección y el rendimiento.

Sin embargo, el "rendimiento" de un programa paralelo es un asunto complicado. Debemos considerar tiempo y escalabilidad de los *kernels* computacionales, además de la ejecución, los mecanismos por los que los datos son generados, guardados, transmitidos sobre redes, analizar si se movieron hacia y desde el disco, e intercambiaron etapas diferentes de un cálculo. Debemos considerar costos incurridos en las fases del ciclo vital de software, incluyendo el diseño, la puesta en funcionamiento, la ejecución y el mantenimiento. Por lo tanto, las métodos por los que medimos el rendimiento pueden ser tan diversos como el tiempo de ejecución, la eficiencia paralela, los requisitos de memoria, el tipo de proceso y transferencia de la red, los costos de diseño, los costos de puesta en funcionamiento, los costos de verificación, el potencial para el uso repetido, los requisitos de equipo físico, los gastos de equipo físico, los costos de mantenimiento, la portabilidad, y la escalabilidad.

La respectiva importancia de estas medidas variará de acuerdo con la naturaleza del problema. Una especificación puede suministrar los datos concretos para algunas mediciones, requerir que los otros sean optimizados, y hacer caso omiso de otros aún. Por ejemplo, la especificación de diseño para un sistema de pronóstico meteorológico (El cual es uno de los

principales clientes del supercómputo) en funcionamiento puede especificar tiempo de ejecución máximo, gastos de equipo físico, y gastos de puesta en práctica, y requerir que la fidelidad del modelo sea maximizada dentro de estas restricciones.

Por contraste, un grupo de ingenieros que desarrollan un programa de búsqueda de base de datos paralelo lo cual en la actualidad es usado por Oracle y por IBM. Aquí, la escalabilidad es menos crítica, pero la clave es que debe adaptarse a los cambios, tanto en el sistema de base de datos como tecnologías de computadora fácilmente.

CAPÍTULO 1

SUPERCÓMPUTO

1.1 INTRODUCCIÓN.

La necesidad de una mayor capacidad de procesamiento en las computadoras ha conducido al desarrollo de varias arquitecturas paralelas, usualmente del tipo SIMD o MIMD de acuerdo a la clasificación propuesta por *Flynn*.

Las computadoras del tipo MIMD pueden ser clasificados como máquinas de memoria compartida(multiprocesadores) o de memoria distribuida (multicomputadoras).

Aunque los primeros son más costosos de construir, tienen sin embargo la ventaja de que son más sencillos de programar.

Dicha línea de trabajo es común a la seguida en la mayoría de las universidades y centros de investigación, ya que de esta forma se puede aprender cómodamente a programar una máquina paralela, de tal forma que cuando el programador se enfrente con la máquina real tenga una mayor probabilidad de que su aplicación esté bien programada. Ello es debido a que el programar en paralelo es muy diferente a la programación secuencial. Por ello, en los últimos años ha proliferado el uso y desarrollo de entornos y herramientas que hagan más fácil la tarea de programar en paralelo.

El modelo de programación elegido no se ha restringido al SPMD (único código y múltiples datos), pues se permite que en cada procesador se ejecute un proceso diferente al del resto de procesadores. Es decir, no sólo permitimos un estilo de programación de paralelismo en los datos sino también de paralelismo en el flujo de control. Asimismo, permitimos que se trabaje con la idea de una zona de memoria privada, ya que un procedimiento puede tener variables locales que podrían no ser accesibles por el resto de procesos (a nivel lógico, ya que a nivel físico todas las variables se localizan en la memoria común de la máquina y por tanto direccionables por cualquier proceso).

1.2 ARQUITECTURAS DE COMPUTADORAS.

En 1966 Michael Flynn propuso un mecanismo de clasificación de las computadoras. La taxonomía de Flynn es la manera clásica de organizar las computadoras, y aunque no cubre todas las posibles arquitecturas, proporciona una importante penetración en varias arquitecturas de computadoras. El método de Flynn se basa en el número de instruccciones y de la secuencia de datos que la computadora utiliza para procesar información. Puede haber secuencias de instrucciones sencillas o múltiples y secuencias de datos sencillas o múltiples. Esto da lugar a 4 tipos de computadoras, de las cuales solamente dos son aplicables a las computadoras paralelas.

- *Single Instruction Stream, Single Data Stream* (SISD)
- *Single Instruction Stream, Multiple Data Stream* (SIMD)
- *Multiple Instruction Stream, Single Data Stream* (MISD)
- *Multiple Instruction Stream, Multiple Data Stream* (MIMD)

1.2.1 SISD (Single Instruction Single Data).

Éste es el modelo tradicional de computación secuencial donde una unidad de procesamiento recibe una sola secuencia de instrucciones que operan en una secuencia de datos (figura 1.1).

Figura1.1

Las computadoras seriales de uso común caen dentro del SISD en donde cada una de las instrucciones son realizadas de manera secuencial en una unidad de tiempo, éste es el modelo llamado Von Neumann, en donde una instrucción es ejecutada por unidad de tiempo para producir un resultado, en la actualidad existen procesadores que manejan estructuras de procesamiento paralelo interno, pero debido a que es dentro del procesador y que el hardware va más

adelante que el software no se utilizan desde el código de los programas, por lo que sólo se percibe una mayor velocidad en la ejecución de los programas seriales.

1.2.2 SIMD (Single Instruction Multiple Data).

A diferencia de SISD, en este caso se tienen múltiples procesadores que sincronizadamente ejecutan la misma secuencia de instrucciones, pero en diferentes datos. El tipo de memoria que estos sistemas utilizan es distribuida.

Aquí hay N secuencias de datos, una por procesador, así que diferentes datos pueden ser utilizados en cada procesador. Los procesadores operan sincronizadamente y un reloj global se utiliza para asegurar esta operación. Es decir, en cada paso todos los procesadores ejecutan la misma instrucción, cada uno en diferente dato.

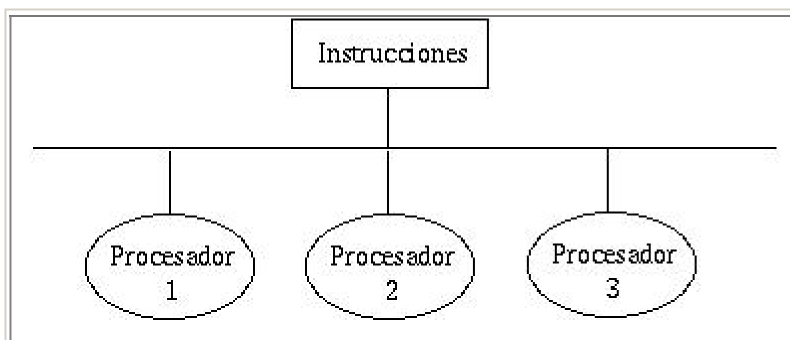


Figura 1.2 Modelo SIMD(Single Instruction Multiple Data).

Las máquinas SIMD son generalmente máquinas de múltiples procesadores las cuales cuentan con una unidad de control y una red, la unida de control emite instrucciones a todos los procesadores de manera simultánea, los procesadores realizan la tarea que se les manda de manera simultánea, todas las instrucciones son controladas por el controlador central el cual por medio de máscaras puede determinar cuáles procesadores están listos para procesar una nueva instrucción por lo que la realización de todo el proceso está generalmente sincronizada y esto evita las condiciones de carrera, ya que facilita mucho el saber el estado actual de cada uno de los procesadores y el estado de los datos.

1.2.3 MISD (Multiple Instrucion Single Data).

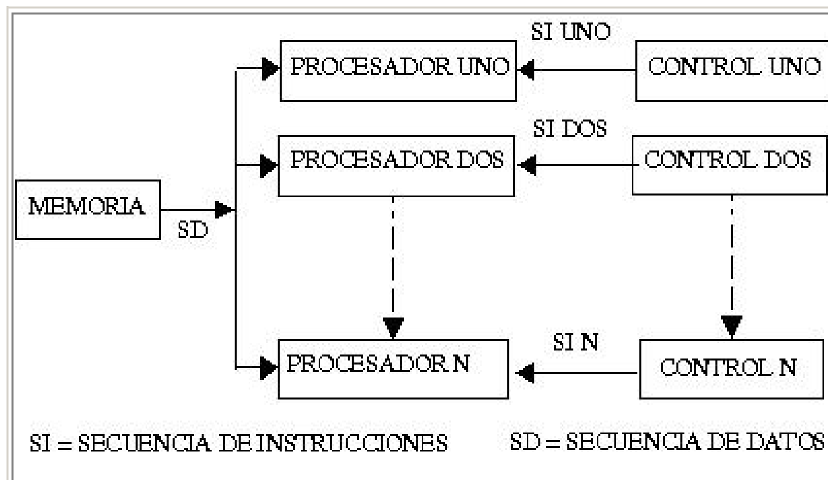


Figura 1.3 MISD (Multiple instruction Single Data.

En este modelo, secuencias de instrucciones pasan a través de múltiples procesadores. Diferentes operaciones son realizadas en diversos procesadores. N procesadores, cada uno con su propia unidad de control comparten una memoria común.

Aquí hay N secuencias de instrucciones (algoritmos o programas) y una secuencia de datos. El paralelismo es alcanzado dejando que los procesadores realicen diferentes cosas al mismo tiempo en el mismo dato.

Las máquinas MISD son útiles en cómputos donde la misma entrada está sujeta a diferentes operaciones.

1.2.4 MIMD (Multiple Instruction Multiple Data).

Este tipo de computadora es paralela al igual que las SIMD, la diferencia con estos sistemas es que MIMD es asíncrono. No tiene un reloj central. Cada procesador en un sistema MIMD puede ejecutar su propia secuencia de instrucciones y tener sus propios datos. Esta característica es la más general y poderosa de esta clasificación.

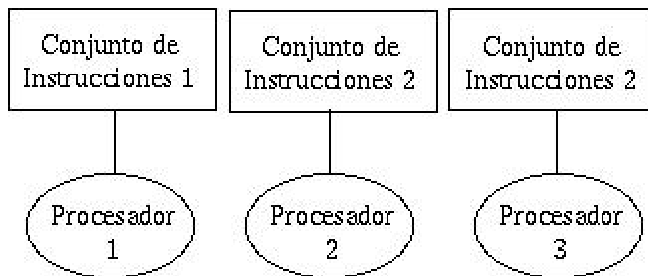


Figura 1.4 Modelo MIMD(Multiple Instruction Multiple Data).

Se tienen N procesadores, N secuencias de instrucciones y N secuencias de datos. Cada procesador opera bajo el control de una sencuencia de instrucciones, ejecutada por su propia unidad de control, es decir cada procesador es capaz de ejecutar su propio programa con diferentes datos. Esto significa que los procesadores operan asíncronamente, o en términos simples, pueden estar haciendo diferentes cosas en diferentes datos al mismo tiempo.

MIMD es una computadora de más de dos procesadores que tiene la posibilidad de ejecutar más de un programa a la vez, por lo que la principal diferencia con una máquina SIMD es que cada uno de los nodos puede ejecutar su propio programa, tal vez todos los

procesadores estén ejecutando el mismo programa pero en una parte diferente, este tipo de estilo de programación es llamado SPMP (*Simple Program Múltiple Data*), es muy poco usado porque en la realidad es poco probable que cada uno de los procesadores tenga un programa diferente, por lo que este estilo de programación así como la estructura SIMD simplifica mucho el razonamiento de los procesos así como el estatus de los procesadores y las estructuras de los datos.

Por lo anterior han surgido varias configuraciones de computadoras las cuales están conformadas por módulos SMP los cuales de manera lógica usan memoria compartida y de manera física son memoria distribuida y a su vez todos los módulos funcionan de manera lógica con una arquitectura de memoria distribuida.

Los sistemas MIMD se clasifican en:

- Sistemas de Memoria Compartida.
- Sistemas de Memoria Distribuida.
- Sistemas de Memoria Compartida Distribuida.

1.2.4.1 MEMORIA COMPARTIDA

En una máquina de memoria compartida existe sólo una memoria disponible para todos los procesadores en la máquina, los cuales generalmente son conectados mediante un *bus* común, este modelo es similar a un

grupo de estudiantes que toman una clase del mismo pizarrón en donde todo lo que se comenta se escribe y lo que cada uno aprende o requiere lo toma del mismo, en este caso el pizarrón es la memoria y los alumnos son los procesadores.

Las computadoras MIMD con memoria compartida son sistemas conocidos como de multiprocesamiento simétrico (SMP) donde múltiples procesadores comparten un mismo sistema operativo y memoria. Otro término con que se le conoce es máquinas fuertemente acopladas o de multiprocesadores.

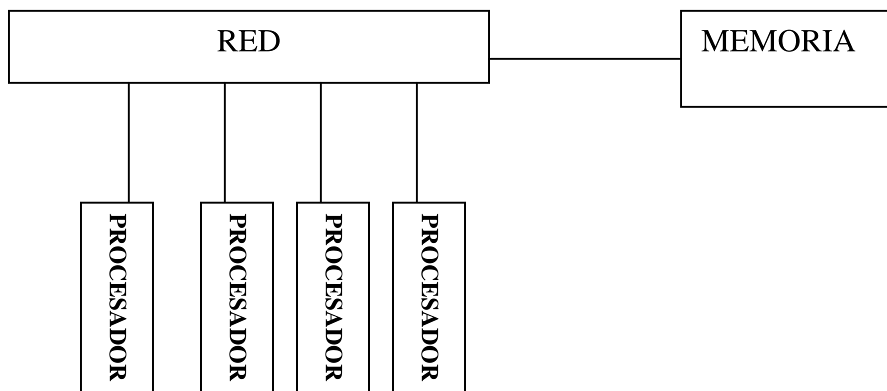


Figura 1.5 Memoria Compartida.

1.2.4.2 MEMORIA DISTRIBUIDA

En una máquina de memoria distribuida cada procesador cuenta con una memoria local y la comunicación entre ellos es por medio de mensajes, si se toma el ejemplo anterior se puede ver un salón con un maestro y un solo pizarrón por maestro en el cual cuando algún maestro se comunica con otro le manda un mensaje de lo

que requiere y el otro profesor manda un mensaje de respuesta, los mensajes son llevados por algún alumno, en este caso, cada uno de los profesores representan un procesador y cada uno de los alumnos un proceso, los pasillos de la escuela una red de comunicaciones y cada uno de los pizarrónes de los salones la memoria local de cada procesador.

Estos sistemas tienen su propia memoria local. Los procesadores pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos. Esta comunicación se le conoce como Paso de Mensajes.

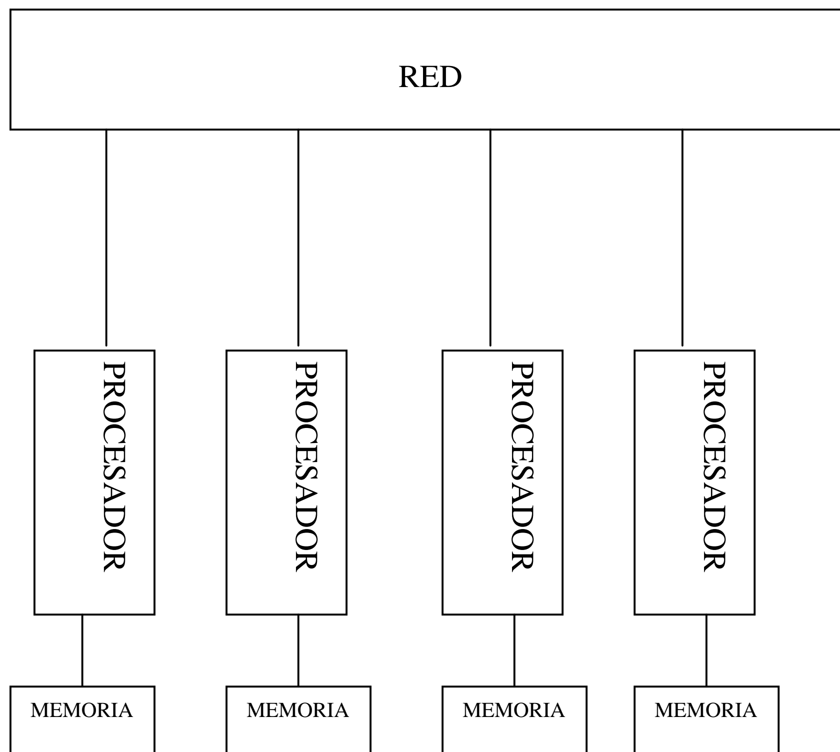


Figura 1.6 Memoria Distribuida.

1.2.4.3 Memoria Compartida Distribuida

Es un *cluster* o una partición de procesadores que tienen acceso a una memoria compartida común pero sin un canal compartido. Esto es, físicamente cada procesador posee su memoria local y se interconecta con otros procesadores por medio de un dispositivo de alta velocidad, y todos ven las memorias de cada uno como un espacio de direcciones globales.

El acceso a la memoria de diferentes *clusters* se realiza bajo el esquema de Acceso a Memoria No Uniforme (NUMA), la cual toma menos tiempo en acceder a la memoria local de un procesador que acceder a memoria remota de otro procesador.

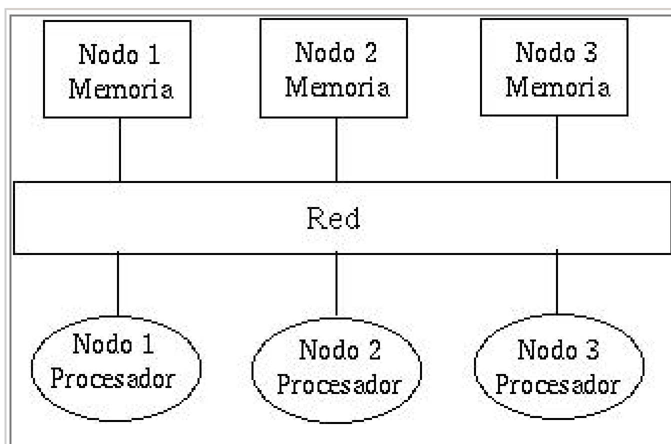


Figura 1.7 Sistemas de Memoria Compartida Distribuida.

1.3 CONDICIONES DE CARRERA

Para computadoras de memoria compartida de tamaño pequeño es posible acceder a cualquier celda de memoria al mismo tiempo, esta propiedad es común en los sistemas SMP (symmetric multiprocessor), sin embargo entre mayor es el número de procesadores esto se hace más difícil por lo que los fabricantes conectan y organizan los procesadores en módulos en los que todos accedan a la misma memoria pero están conectados por una red de alta velocidad, de tal forma que lógicamente usan memoria compartida pero en términos de rendimiento funcionan como memoria distribuida, ya que la computadora puede estar diseñada de tal forma que un número definido de procesadores compartan memoria de manera directa y tengan acceso a la memoria de los demás, pero como es de esperarse, la velocidad de acceso las memorias que no son propias no es la misma, por lo que el rendimiento disminuye.

Este tipo de máquinas de memoria compartida son llamadas NUMA (*non uniform memory accesss*) y lo interesante de este tipo de máquina es que los programas que son optimizados para usarse en memoria distribuida funcionan mejor que los usados para la memoria compartida.

En ambos modelos puede existir bloqueo, en el caso de la memoria compartida puede darse el caso de que los procesadores traten de leer la misma localidad de memoria al mismo tiempo, lo que ocasiona ciertos retrasos; en el caso de la memoria distribuida pasa algo similar pero cuando el nodo maestro recibe diversos mensajes de los nodos al mismo tiempo, de tal forma que no los puede atender a todos y se genera un retraso.

Otro aspecto importante es que los programas deben de mantener una correcta sincronización de los procesos de tal forma que si un proceso requiere datos de otro proceso, no debe de estar en espera un tiempo largo, o peor aún, no debe interrumpir otros procesos que deben enviar datos al proceso que le enviará la información que él requiere y por último debe de asegurarse que los datos que recibió sean realmente los que él necesita, puesto que tal vez el valor fue modificado por otro proceso antes de que él lo reciba y por lo tanto es un dato incorrecto. Lo anterior es conocido como condiciones de carrera o *race conditions*. *Race conditions* son una fuente de errores en programación paralela, ya que encontrarlos es difícil, por lo que se deben de tratar de evitar desde el diseño.

1.4 GRANULARIDAD

Cuando hablamos de arquitecturas paralelas generalmente se habla de granularidad lo cual se refiere a la relación del número y complejidad de los procesadores. Una máquina con granulado fino consiste en un número relativamente grande de procesadores de poca memoria y poco poder de cómputo. Una máquina de granulado grueso se refiere a unos cuantos procesadores con un gran poder de cómputo.

El granulado fino generalmente consiste en máquinas SIMD en la cual cada uno de los procesadores no son muy poderosos y cada uno tiene su memoria local. En este caso se tienen procesadores muy baratos y sencillos. En los casos en los que se usan procesadores más caros pero que no llegan a ser de alto rendimiento por lo que pueden estar en cualquier PC, se usan dentro de máquinas de granularidad media con arquitecturas MIMD.

Estas últimas máquinas son las más usadas en la actualidad para cómputo paralelo por su relación precio rendimiento.

Las máquinas de granularidad fina son más difíciles de usar debido a que su relativo poco poder de cómputo hace que los algoritmos deban de ser poco demandantes

en cada uno de los procesos, de tal forma que no todos los algoritmos se pueden adaptar.

Como ejemplo de granularidad fina tenemos la figura 1.8 en la que a pesar de tener doce procesadores, cada uno de ellos son procesadores de PC de bajo rendimiento.

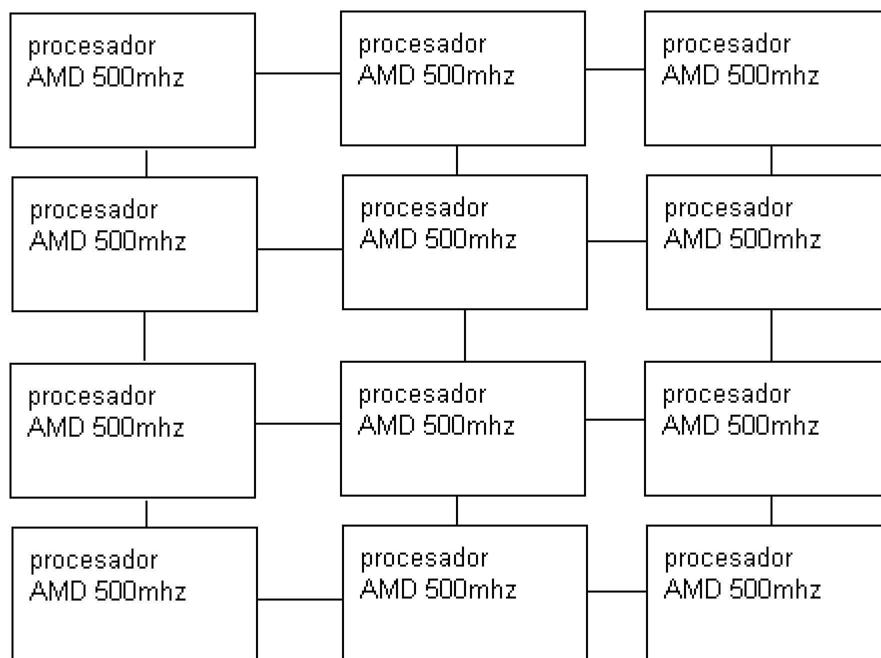


Figura 1.8 Graularidad fina

Como ejemplo de Granularidad gruesa se tiene la figura 1.9 en la cual a pesar de que el número de procesadores es menor, el rendimiento es mayor en cada uno de los nodos.

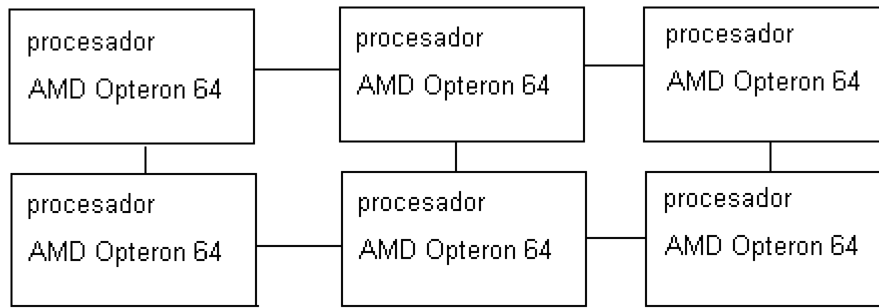


Figura 1.9 Granularidad gruesa

1.5 MODELOS DE CONEXIÓN DE REDES

Como se menciona a lo largo del capítulo las redes juegan un papel fundamental en las computadoras de varios procesadores, el tipo de redes que usa no es muy similar a las usadas en la industria por lo que hay que definir ciertos términos y estructuras.

1.5.1 Grado de un procesador

El **grado de un procesador** P es el número de procesadores que están directamente conectados a P por medio de un canal de comunicación bidireccional o liga.

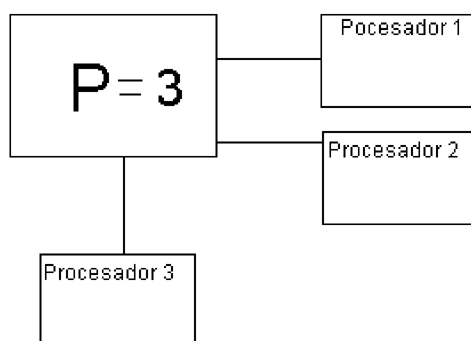


Figura 1.10 Grado de un procesador

1.5.2 Grado de una red.

El **grado de una red** es el máximo P de alguno de los procesadores que son parte de la red.

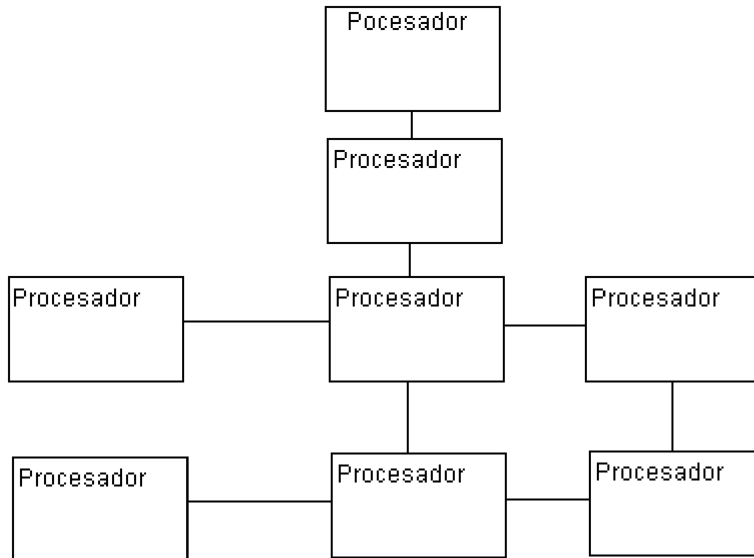


Figura 1.11 Ejemplo de red grado 4

1.5.3 Distancia entre procesadores.

La **distancia** entre 2 procesadores es el número de ligas de la ruta más corta que comunica ambos.

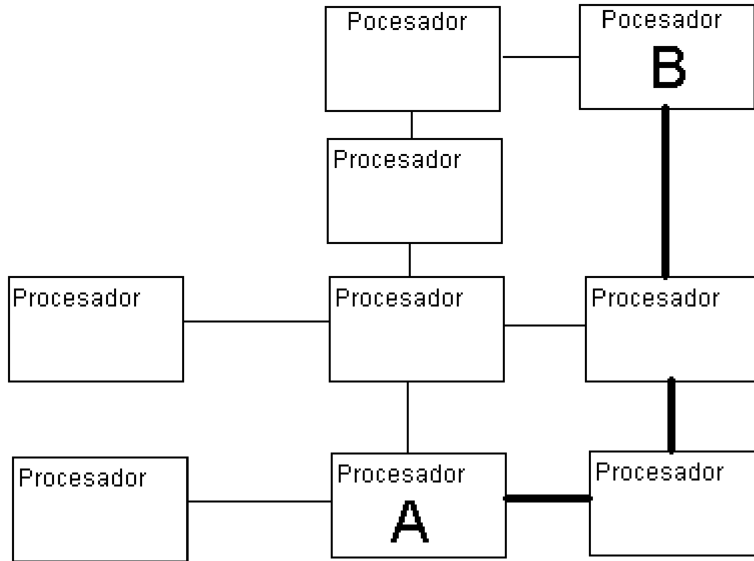


Figura 1.12 Distancia entre los procesadores A y B con valor tres.

1.5.4 Diámetro de comunicación de la red

El **diámetro de comunicación de la red** es la distancia máxima de una par de procesadores que forman parte de la red.

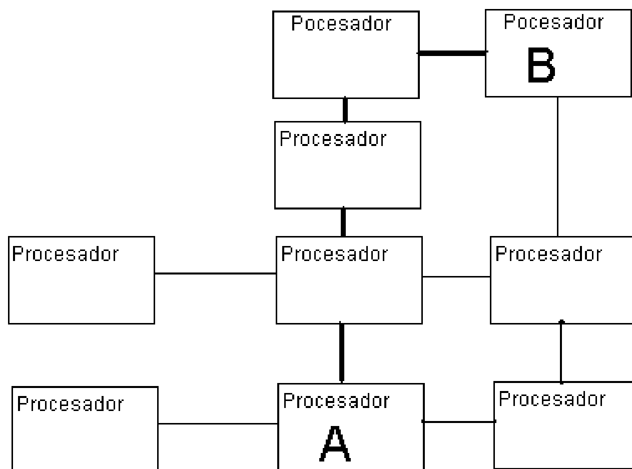


Figura 1.13 Diámetro de comunicación de la red entre los procesadores A y B.

1.5.6 División del ancho de banda.

La **división del ancho de banda** es el número mínimo de ligas que se usan para cortar una red en dos partes, cada una con el mismo número de procesadores.

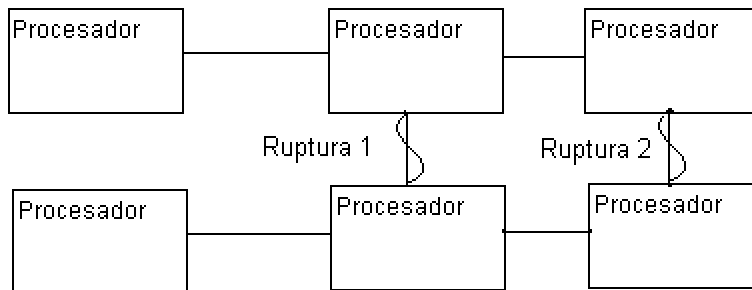


Figura 1.14 División del ancho de banda con dos rupturas.

1.5.7 Objetivos de interconexión.

Una vez definidos los conceptos anteriores se pueden definir las metas de la interconexión de redes como sigue:

- Reducir el grado de los procesadores para reducir costos.
- Minimizar el diámetro de comunicación de la red para reducir el tiempo que le toma a un mensaje llegar a su destino.
- Maximizar la división del ancho de banda para poder minimizar la contención cuando varios mensajes están siendo enviados de manera concurrente.

1.5.8 PRAM

Este modelo es ampliamente usado a pesar de no ser aplicado en la realidad ya que es una idealización de una red perfecta en la cual no hay pérdidas de tiempo por colisiones y problemas propios de la red con un diámetro de comunicación de la red igual a uno.

PRAM proviene de *Parallel Random Access Machine* el cual es una máquina de memoria compartida en la que todos sus procesadores son iguales y tienen acceso a cualquier celda de memoria con una sola unidad de tiempo.

El propósito de la PRAM es realizar el programa paralelo enfocándose en el algoritmo y dejando a un lado problemas de la red y de rendimiento de cada uno de los procesadores.

La PRAM funciona con un modelo SIMD en el que todos los procesos están sincronizados, o mejor dicho deben de estarlo porque de no ser así, se producirá un problema de acceso a memoria.

1.5.9 MODELOS DE RED

Las redes que se van a definir son las más comunes y las más usadas, a pesar de esto las variaciones a los modelos pueden presentarse de manera frecuente.

1.5.9.1 Ring

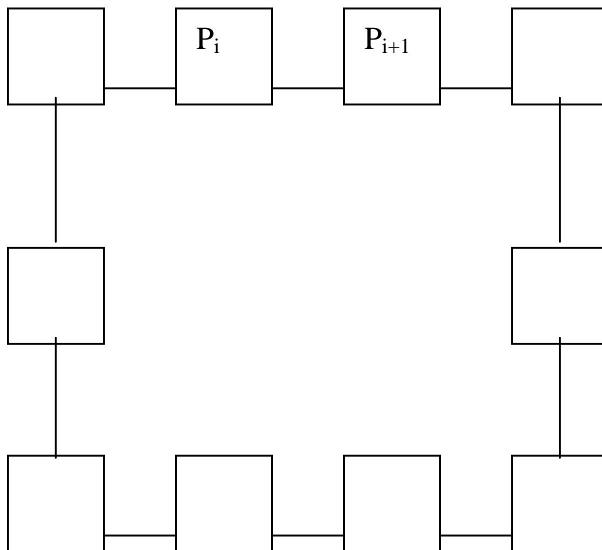


Figura 1.15 Ring.

Como se puede ver en el modelo (figura 1.15) todos los procesadores están conectados en forma de anillo por lo que se pueden ver varias características:

- El grado de la red es sólo dos.
- El diámetro de comunicación es de *n/2* donde *n* es el número total de procesadores el cual es bastante alto.
- La división del ancho de banda es de dos el cual es muy bajo.

1.5.9.2 Mesh

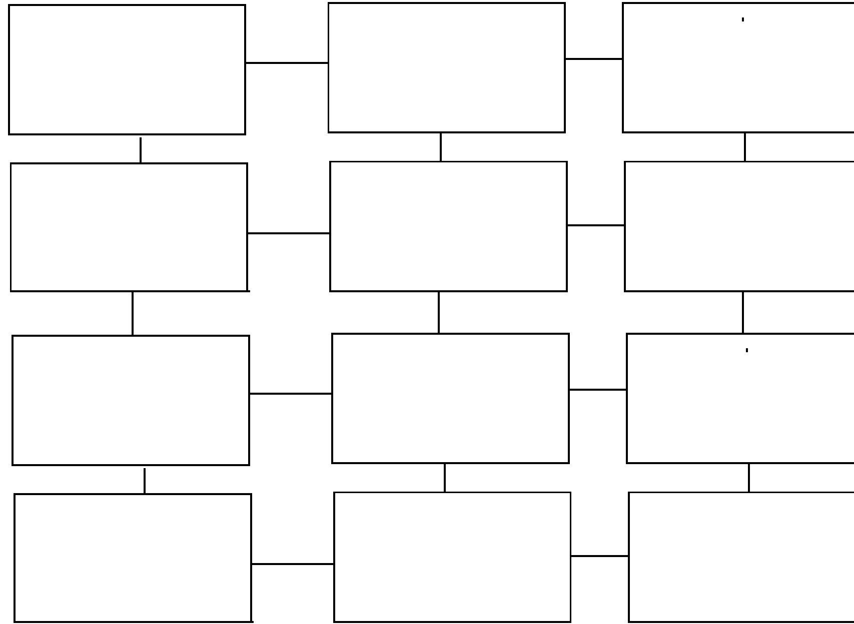


Figura 1.16 Mesh

Los procesadores en este modelo de red están dispuestos en un arreglo bidimensional de grado dos, de tal forma que cada uno de los procesadores tiene conectados cuatro vecinos, los procesadores de los extremos sólo tres y los de las esquinas dos.

Las características de este modelo son las siguientes:

- El grado de la red es de cuatro por lo que es el doble del modelo *Ring*.
- El diámetro de comunicación es $2(n^{1/2} - 1)$ por lo que para la figura 1.16 toma un valor de seis por lo que se reduce con respecto al *Ring*.
- La división del ancho de banda es $n^{1/2}$

Hay que notar que el modelo *Ring* es un caso especial de este modelo en el que el grado del arreglo es 1.

1.5.9.3 Hipercube

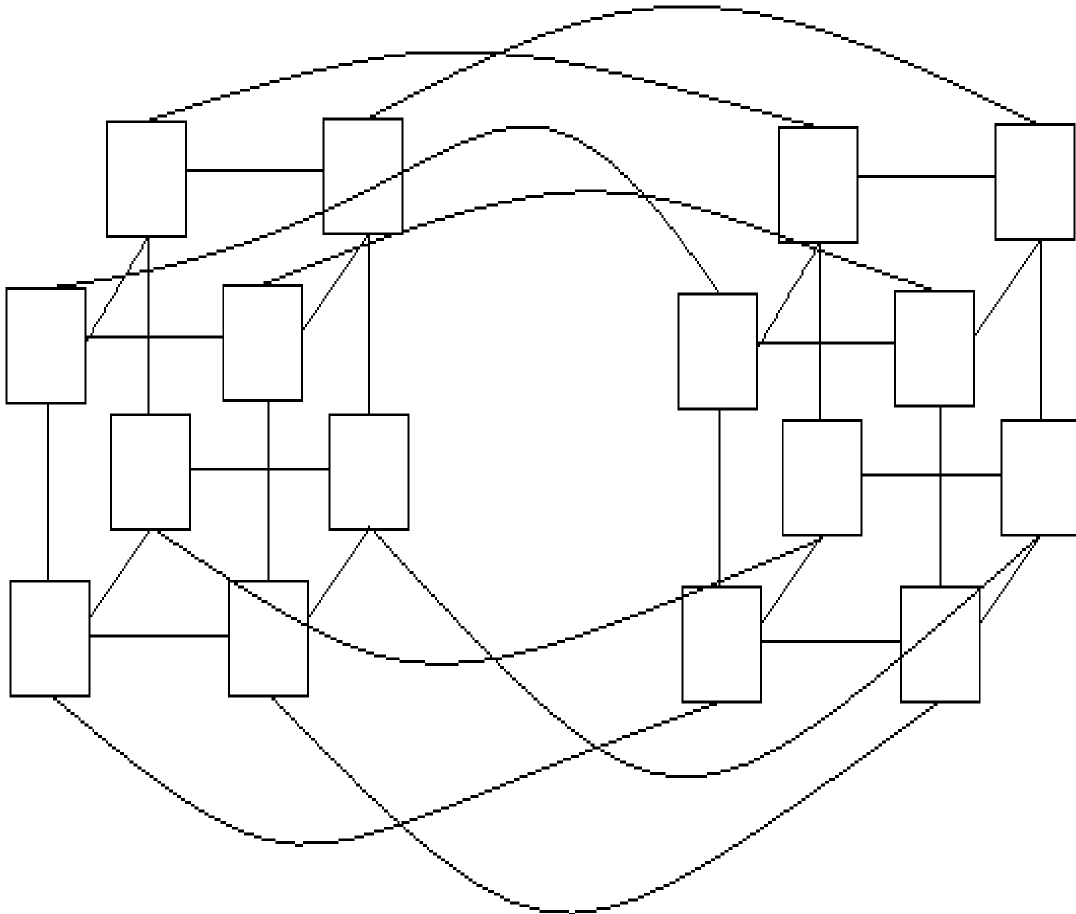


Figura 1.17 Hipercube

Como se puede ver en la figura 1.17 los procesadores se colocan en arreglos con forma de cubo, los cuales a su vez se pueden conectar con otros cubos uniendo los nodos de posición similar, de tal forma que la unión de cuatro cubos forma un cubo nuevo.

Las ventajas de esta arquitectura es que el diámetro de comunicación es solo dos y la división de ancho de banda es *n/2*, su desventaja es que el número de ligas de comunicación de cada procesador es de $log_2 n$.

1.5.9.4 Árbol

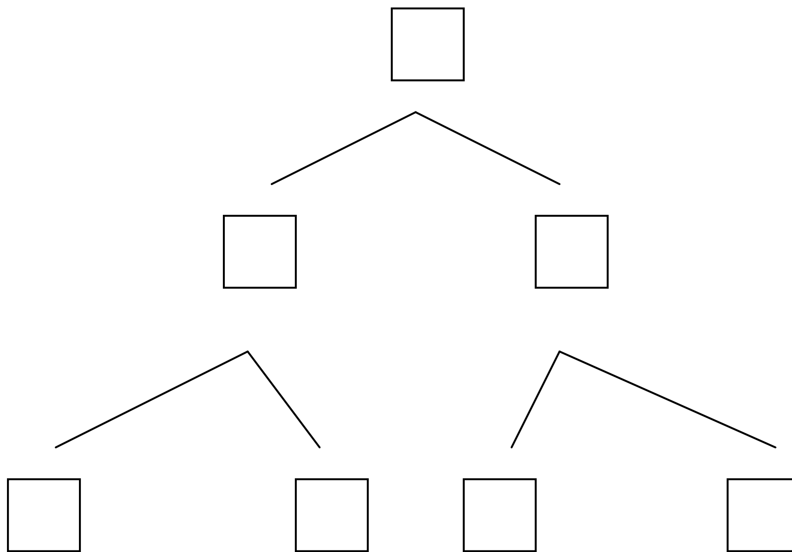


Figura 1.18 Árbol

Un árbol binario completo de altura *k* para todo *k>0* de valor entero tiene $n=2^{k+1} - 1$ procesadores, donde el nodo raíz está en el nivel 0, cada procesador tiene dos hijos con excepción de los nodos del nivel k, el nodo raíz no tiene padre y todos los nodos sólo tienen un padre.

El grado de la red es de 3 y el diámetro de comunicación es $2k=2[log_2 n]$, la gran desventaja de este modelo es cuando hay gran comunicación entre los nodos, ya que los mensajes pasan generalmente por el

nodo raíz, de tal forma que se puede generar un cuello de botella.

1.6 ALGORITMOS

Muchos de los algoritmos paralelos son meramente secuenciales con la misma estructura que los algoritmos diseñados para máquinas seriales.

Se pueden considerar tres fases, la fase de inicialización, la fase de cómputo y la fase de terminación en las cuales durante la segunda fase se cambia el programa de una forma no secuencial, de tal forma que dista mucho de su inicio secuencial optimizado.

El diseño de los algoritmos paralelos, por razones obvias es muy distinto al de máquinas secuenciales, por lo que los patrones que se siguen son distintos

Para manipular estructuras enteras en un paso, es práctico tener un conjunto de operaciones previamente hechas que permita optimizar dichas manipulaciones, estas operaciones globales pueden ser un verdadero problema, pero una vez optimizadas pueden resultar ser muy prácticas, por ejemplo los filtros que nos permiten ordenar valores, o las operaciones de *broadcast* las cuales permiten enviar información de un procesador al otro.

Detrás del manejo de subconjuntos de procesadores, se encentran muchas de las operaciones globales más comunes, como son mínimo valor de una lista, máximo, operaciones básicas como sumatoria y producto.

Como ejemplo de esto se encuentra *Fortran* el cual cuenta con muchas de estas operaciones, así como también lo tiene MPI y bibliotecas de C.

Otras de las operaciones importantes son las instrucciones de ordenamiento, las cuales permiten de una lista de valores obtener una lista de valores ordenada, la cual puede tomar ventaja de todos los procesadores disponibles para ordenar subconjuntos de la lista principal.

1.6.1 Divide y vencerás

Divide y vencerás es un poderos algoritmo que explota la repetida subdivisión de problemas y datos en problemas más pequeños, sin embargo es poco usado en el cómputo paralelo porque el dividir el problema en problemas de menor tamaño que se ejecuten de manera separada nos lleva a algoritmos secuenciales que se ejecutan en cada procesador, por lo que la interacción entre los procesadores es mínima y el problema puede ser resuelto con un algoritmo secuencial optimizado.

Un ejemplo de este algoritmo son los casos de paralelismo perfecto en los cuales un problema es iterativo y no requiere demasiados datos previos de otros procesos; como ejemplo de esto se puede pensar en un problema de predicción del tiempo en el que se analiza el comportamiento del océano de tal forma que una parte del océano es partida por un malla de sensores, cada uno de los sensores arroja datos que deben de ser analizados y procesados, por lo que si se ejecuta un proceso que haga esto para cada sensor, de manera secuencial, el tiempo de respuesta es directamente proporcional al número de sensores, por lo que este tipo de algoritmo puede ser eficaz en este caso; a pesar de que el proceso de análisis de cada uno de los sensores sea el mismo, si tomamos en cuenta que el análisis de la información de un sensor puede durar un par de minutos y que se tienen 500 sensores en el océano, el procesar toda la información tomaría 1000 minutos, por lo que si se quieren sacar muestras cada 10 minutos sería imposible hacerlo con una máquina secuencial, en cambio con una máquina de 250 procesadores se podría lograr en 5 minutos aproximadamente.

Es importante resaltar que este algoritmo en realidad no está parelilizando un proceso, es decir, no lo está dividendo en varias partes dentro de una máquina de varios procesadores, sino que se está ejecutando en procesos separados que pueden ser el mismo en varios procesadores.

Como se puede ver en la figura 1.19 existe un nodo maestro sin embargo cada uno de los procesos son manejados de manera independiente sin interacción constante con el maestro.

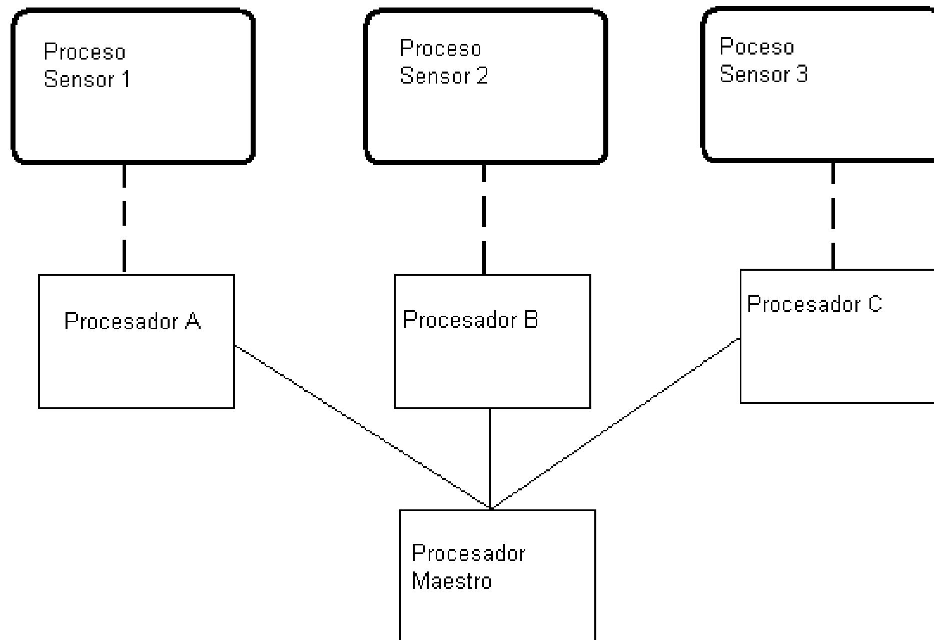


Figura 1.19 Divide y vencerás

1.6.2 MAESTRO Y ESCLAVO

Éste es uno de los algoritmos más usados sobre todo en máquinas que funcionan con paso de mensajes, consiste en tener uno de los nodos como maestro y los demás como esclavos, hay que mencionar que el nodo maestro es cualquier nodo, no se requiere un nodo especial, sin embargo, para computadoras que no son de multipropósito, el optimizar un maestro con un hardware más potente es de gran ayuda.

El maestro es responsable de la sincronización de los otros nodos, es el encargado de evitar condiciones de carrera y realizar pequeñas operaciones con los datos obtenidos de los nodos, el desarrollo del sistema en Java de la presente tesis es desarrollado bajo este esquema; aunque se puede permitir usar un algoritmo de divide y vencerás, en el fondo el sistema es administrado en su totalidad por un nodo maestro, es decir el sistema funciona como maestro esclavo para el control de los nodos sin que eso implique que los programas paralelos que se realicen deban de seguir esta lógica.

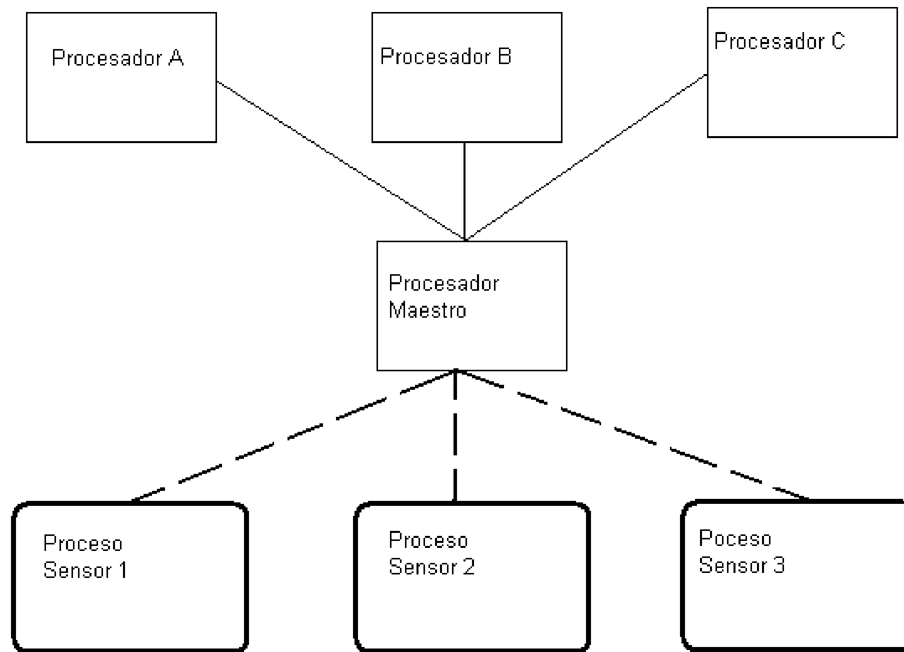


Figura 1.20 Maestro Esclavo

1.6.3 *PIPELINE* O TUBERÍA

Otra técnica de algoritmo paralelo es la tubería en la cual la salida de un proceso es la entrada de otro; esto es muy usado cuando la carga que recibe un procesador de un algoritmo no es la misma en todas sus partes, por lo que distribuir las cargas es una excelente opción, de tal forma que las tareas más pesadas son atendidas por varios nodos y las tareas más ligeras por uno solo o un número de nodos menor.

Un ejemplo de esto es cualquier algoritmo que tenga una parte sencilla y otra demasiado complicada en la

cual se requiere gran poder de cómputo de tal forma que si se tienen, por ejemplo 10 nodos, 3 grupos de tres pueden realizar las partes más difíciles y demandantes del algoritmo y una vez realizadas mandar los resultados al nodo restante que realiza la parte sencilla del algoritmo en un tiempo mucho menor, evitando de esta forma que en algún momento algunos ejecuten de manera individual las partes más sencillas y otros las partes más difíciles sin ayuda de los demás nodos.

Como se puede ver en la figura 1.21 los procesadores A y B atienden a los procesos más demandantes de procesamiento, mientras que el procesador C sólo atiende al proceso ligero que procesa datos obtenidos de los procesos demandantes.

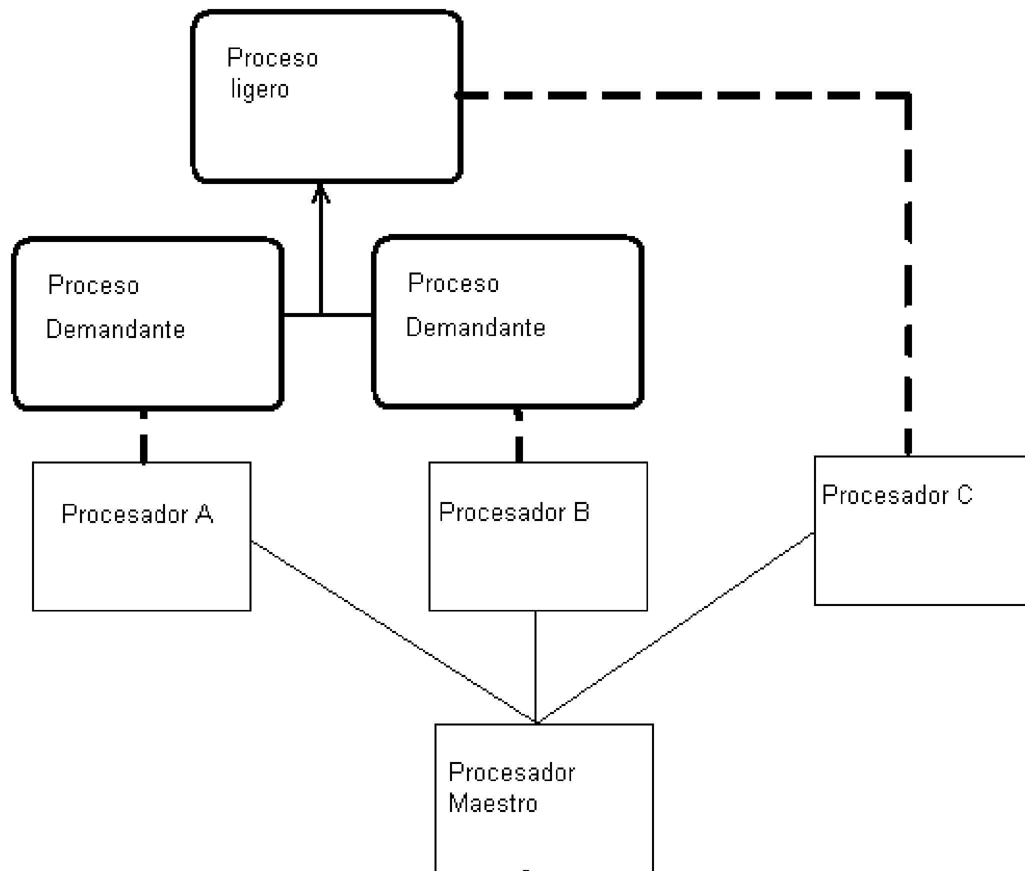


Figura 1.21 Tubería

CAPÍTULO II

PASO DE MENSAJES

2.1 INTRODUCCIÓN

Principalmente hay dos esquemas de comunicación: los sistemas de memoria compartida y los sistemas de mensajes.

En los sistemas de memoria compartida se necesita que los procesos comunicantes compartan algunas variables y se trata de que intercambien información mediante esas variables compartidas. En estos sistemas, la responsabilidad de facilitar la comunicación reside en los programadores de la aplicación, mientras que el sistema operativo sólo tiene que facilitar la memoria compartida.

En los sistemas de mensajes, la responsabilidad de facilitar la comunicación reside en el sistema operativo, permitiendo que los procesos intercambien mensajes sin necesidad de recurrir a variables compartidas. Estos sistemas son interesantes especialmente para sistemas distribuidos sin memoria común, donde el uso de semáforos y monitores (normalmente centralizados en una parte del sistema) es con frecuencia ineficaz y lento.

Obviamente estos dos esquemas no son mutuamente excluyentes y podrían utilizarse simultáneamente dentro del mismo sistema operativo.

Las arquitecturas en las cuales los procesadores están comunicados por una red de comunicaciones han llegado a ser muy comunes, un ejemplo de ello son los *Clusters*.

2.2 CLUSTERS

A un nivel básico un *cluster* (también llamado NOW-Network of Workstations o COW-Cluster of Workstations) es una colección de estaciones de trabajo o PC's que están interconectados mediante algún tipo de red. Generalmente estas redes son de alta velocidad. La principal característica de un *cluster* es que trabaja como una colección integrada de recursos y dispone de una única imagen del sistema que es compartida por todos sus nodos.

El uso de clusters para desarrollar, depurar y ejecutar aplicaciones paralelas está ganando en popularidad, convirtiéndose en una gran alternativa al uso de arquitecturas más especializadas debido al gran precio de éstas. Un factor importante que ha hecho que el uso de *clusters* sea práctico es la estandarización de numerosas herramientas y utilidades usadas en aplicaciones paralelas. Ejemplos de estos estándares son la biblioteca de paso de mensajes MPI (Message Passing Interface), y el lenguaje paralelo de datos High Performance Fortran (HPF), más adelante en la presente tesis se discuten estas herramientas. Esta estandarización permite que las aplicaciones sean desarrolladas y probadas (e incluso ejecutadas) en *clusters*, y a continuación, en una etapa posterior, puedan ser llevadas, con ligeras modificaciones, a plataformas paralelas especializadas.

2.2.1 CARACTERÍSTICAS DE CLUSTERS

Cada uno de los nodos es una máquina completa que podremos utilizar como una estación de trabajo normal, al mismo tiempo que lo empleamos para nuestra máquina paralela.
Debido a que puede estar formado por PC's normales de gama media o incluso baja (Pentium X,AMD), su precio es relativamente bajo (causado por el gran mercado que hay actualmente).
Si no vamos a emplear cada uno de los nodos como estación de trabajo, nos bastará con un teclado y una consola para todo el sistema, pudiéndonos evitar la compra incluso de la tarjeta de vídeo ya que el sistema operativo Linux (sobre el que se suele trabajar) puede ejecutarse sin todos estos elementos.
Como bien sabemos la construcción de una supercomputadora de *n* procesadores (tomando *n* como un valor superior a cuatro) es complejo, caro y en muchos casos irrealizable. Sin embargo, crear una red con 16, 100 o incluso más nodos es, en comparación, algo trivial.
Cualquier problema en una máquina de este tipo es relativamente sencillo de resolver, ya que si por ejemplo se estropea un procesador en un nodo, nos bastará con sustituirlo por otro para resolverlo, esto es importante tenerlo en cuenta, ya que es infinitamente más sencillo encontrar un nuevo procesador AMD que cualquiera de los componentes de un complejo Fujitsu por ejemplo.

Las estaciones de trabajo están ganando en potencial de procesamiento. Su potencial se ha incrementado drásticamente en los últimos años, y se dobla cada 18-24 meses. Y todo indica a que está tendencia continuará por varios años.

El ancho de banda de la comunicación entre estaciones de trabajo se está incrementando y el tiempo de latencia se decrementa según se implementan las nuevas redes y protocolos en LAN.

Los *clusters* de estaciones de trabajo son más fáciles de integrar en redes existentes que ordenadores paralelos específicos.

Usualmente los usuarios de estaciones de trabajo no aprovechan toda la potencia de cómputo que éstas les ofrecen.

Las herramientas de desarrollo para estaciones de trabajo están más maduras en comparación con las herramientas de desarrollo para ordenadores paralelos, esto se debe principalmente a la naturaleza no estándar de los distintos sistemas paralelos.

Los clusters de estaciones de trabajo son baratos y son una alternativa lista y disponible a las plataformas especializadas de computación paralela.

Los *clusters* son fácilmente escalables. La capacidad de los nodos se puede ampliar de forma sencilla, añadiendo memoria o procesadores.

2.2.2 HERRAMIENTAS PARA CLUSTERS

2.2.2.1 Fortran

Fortran, tal y como es hoy, es un lenguaje de alto nivel, con numerosas mejoras con respecto al ANSI del 1966, (objetos como los de C++, instrucciones para procesamiento en paralelo,...)

2.2.2.3 Mentat

Mentat es un lenguaje orientado a objetos para procesamiento en paralelo, funciona en máquinas *clusters* y se encuentra disponible para Linux. Su sintaxis es similar a la del popular C++ y por tanto a Java.

2.2.2.4 NESL

NESL es un lenguaje paralelo que integra varias ideas de la comunidad teórica (algoritmos paralelos), de la comunidad de lenguajes (lenguajes funcionales) y de la comunidad de sistemas (muchas de las técnicas de implementación).

2.2.3 AMBIENTES DE PROGRAMACIÓN PARA *CLUSTERS*

La principal ventaja de establecer un estándar en el paso de mensajes es la portabilidad y la facilidad de uso. En un sistema de memoria compartida en el cual las rutinas de más alto nivel y las abstracciones están construidas sobre una capa de bajo nivel de paso de mensajes, los beneficios de la estandarización son particularmente aparentes. Además permite que se ofrezca soporte hardware y se aumente la escalabilidad.

2.2.3.1 PVM

La PVM (Parallel Virtual Machine) no es más que un API GNU de programación C para máquinas paralelas, capaz de crear una máquina virtual paralela, es decir, emplea los recursos libres de cada uno de los nodos sin tener que preocuparnos del cómo, haciéndonos creer que estamos ante una única supermáquina.

Un punto a destacar de PVM es que si disponemos de una red UNIX no tenemos más que instalar el software que, además, es libre y ya dispondremos de una supercomputadora paralela con un coste nulo, y podremos ponernos a trabajar con ella sin dejar de emplear cada máquina de la red para las mismas funciones que antes.

Además, PVM está portado para otras máquinas además de Linux, entre las que se encuentran distintos tipos de UNIX e incluso existe una versión para NT aunque cabe decir que alcanza unas velocidades bastante inferiores a las que alcanza con Linux.

2.2.3.2 Inferno

Inferno es un nuevo **sistema operativo** y un **entorno de programación** para repartir contenido en un rico entorno de redes heterogéneas, clientes y servidores. El sistema Inferno incluye el *kernel* Inferno, el lenguaje de programación Limbo, y APIs de referencia que incluyen interfaces para redes, gráficos, protocolos de red, seguridad y autentificación y varios kits de herramientas.

2.2.3.4 MPI

MPI (Message Passing Interface). El objetivo básico de MPI es desarrollar un estándar de amplia utilización para escribir programas de paso de mensajes. Esta interfaz intenta establecer un práctico, portable, eficiente y flexible estándar para el paso de mensajes.

En el diseño del MPI se utilizó en gran medida el MPI Forum de forma que se tomaba en cuenta la opinión de los programadores. Ha sido influenciado en gran medida por el trabajo en el IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, y PARMACS. Otras contribuciones importantes provienen de Zipcode, Chimp, PVM, Chamaleon y PICL.

La principal ventaja de establecer un estándar en el paso de mensajes es la portabilidad y la facilidad de uso. En un sistema de memoria compartida en el cual las rutinas de más alto nivel y las abstracciones están construidas sobre una capa de bajo nivel de paso de mensajes, los beneficios de la estandarización son particularmente aparentes. Además permite que se ofrezca soporte hardware y se aumente la escalabilidad.

2.2.3.5 Mosix

Mosix es una herramienta desarrollada para sistemas tipo UNIX, cuya característica resaltante es el uso de algoritmos compartidos, los cuales están diseñados para responder al instante a las variaciones en los recursos disponibles, realizando el balanceo efectivo de la carga en el cluster mediante la migración

automática de procesos o programas de un nodo a otro en forma sencilla y transparente.
El uso de Mosix en un cluster de PC's hace que éste trabaje de manera tal que los nodos funcionan como partes de un solo computador. El principal objetivo de esta herramienta es distribuir la carga generada por aplicaciones secuenciales o paralelizadas.

A diferencia de paquetes como MPI o PVM, Mosix realiza la localización automática de los recursos globales disponibles y ejecuta la migración dinámica en línea de procesos o programas para asegurar el aprovechamiento al máximo de cada nodo.

2.2.3.6 Condor

Condor es un manejador de carga del sistema especializado para trabajos de computación intensiva. Los usuarios envían sus trabajos a Condor, el cual los coloca en una cola, elige cuándo y dónde ejecutar los trabajos, y finalmente informa al usuario que se han completado.
Puede usarse para manejar un cluster dedicado de nodos de ordenadores (como los cluster “Beowulf”).

2.3 FUNDAMENTOS TEÓRICOS DE PASO DE MENSAJES

Un mensaje es una colección de información que se puede intercambiar entre un proceso emisor y uno receptor. Los mensajes pueden contener datos, órdenes de ejecución o incluso trozos de código.

Un dispositivo de paso de mensajes tiene básicamente dos operaciones: **send** (mensaje) y **receive** (mensaje) (también denominadas enviar y recibir, transmitir y recibir, etc). La operación **receive** normalmente bloquea al proceso que la llama hasta que recibe el mensaje.
Los formatos de los mensajes suelen ser flexibles. Podemos dividirlos en cabecera y cuerpo. La cabecera tendría una serie de campos fijos: identificadores de emisor y receptor, tipo del mensaje, longitud, etc, y el cuerpo, que es opcional, tiene el mensaje en sí y su tamaño puede variar de unos sistemas operativos a otros e incluso dentro de un mismo sistema operativo.

Para que los procesos puedan comunicarse mediante este sistema, es necesario un canal *o* enlace de comunicación entre ellos. Este canal puede implementarse de múltiples formas.

Ya sea que los procesadores estén o no comunicados por una red, es conveniente que los procesos no compartan las variables, en el caso del paso de mensajes, el manejo de variables separadas, tomando en cuenta un proceso por cada procesador independiente, es casi automático.
Cuando se utiliza el paso de mensajes, los canales de comunicación son los únicos objetos que son compartidos, donde un canal es una abstracción de una red de comunicaciones física.
No se requieren mecanismos especiales de exclusión mutua, por lo que las variables son *caretaker*, es decir, cada variable es local y se puede tener acceso

a ellas sólo por un proceso. El no tener variables compartidas provoca que las técnicas de programación cambien y que los programas desarrollados para paso de mensajes no puedan ser utilizados o ejecutados en ambientes de memoria compartida.

Los programas que usan paso de mensajes, distribuyen los mensajes entre los procesadores, por este motivo, se les conoce como programas distribuidos.

Un programa de paso de mensajes se puede adaptar para ambientes de un solo procesador, o bien para ambientes de memoria compartida, esto es mediante el uso de *threads* o multiproceso en lugar de varios canales de comunicación.

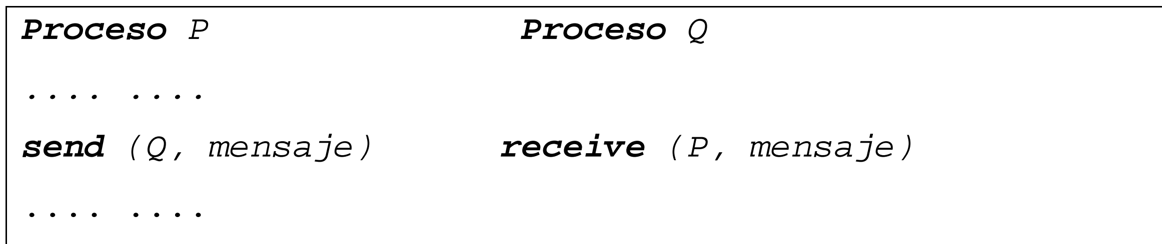
2.3.1 DESIGNACIÓN DIRECTA

Una decisión muy importante a la hora de implementar los mensajes es si la designación sería directa o indirecta. Por *designación directa*, entendemos que siempre que se realice una operación con mensajes, cada emisor debe designar al receptor específico y viceversa, cada receptor debe especificar el emisor del que desea recibir un mensaje. Las operaciones primitivas *send* y *receive* se definirían de la siguiente forma (figura 2.1):

send *(A, mensaje).*Envía un mensaje al proceso A
receive *(B, mensaje).*Recibe un mensaje del proceso B

Figura 2.1 Instrucciones *send* y *receive*

De esta manera, para enviar un mensaje del proceso P al proceso Q necesitaríamos lo siguiente (figura2.2):



Proceso *P* **Proceso** *Q*

....

send *(Q, mensaje)* **receive** *(P, mensaje)*

....

Figura 2.2 Enviar mensaje de P a Q

2.3.1.1 DIRECCIONAMIENTO SIMÉTRICO

Este tipo de comunicación es una comunicación simétrica en el sentido de que cada emisor debe conocer y nombrar a sus receptores y viceversa. Esta asignación uno a uno producida proporciona una comunicación segura de mensajes, ya que cada canal está asociado exactamente a dos procesos *(simétrico)*, pero puede ser un inconveniente para la implementación de rutinas de servicio, por ejemplo sería de poca utilidad una impresora que debe saber los nombres de todos los procesos que pueden acceder a ella, y lo que es peor, el orden en que lo van a hacer.

2.3.1.2 DIRECCIONAMIENTO ASIMÉTRICO

Una variante de este esquema es tener un direccionamiento *asimétrico*, donde únicamente el emisor nombra al receptor. Al receptor no se le pide que nombre al remitente. En este esquema, las primitivas se definen como sigue:

send *(P, mensaje)*. Envía un mensaje al proceso P

receive *(id, mensaje)*. Recibe un mensaje de cualquier proceso, "id" se carga con el identificador del proceso que comunica.

2.3.2 DESIGNACIÓN INDIRECTA

Un método alternativo es la *designación indirecta*, donde los mensajes se envían y se recogen desde depósitos específicamente dedicados a ese propósito. Estos recipientes se denominan normalmente buzones. Las operaciones quedarían de la siguiente forma:

send *(buzon1, mensaje)*. Envía un mensaje al buzón *buzon1*
receive *(buzon2, mensaje)*. Recoge un mensaje del buzón *buzon2*
Para que dos procesos puedan comunicarse deben utilizar el mismo buzón. Cuando se utiliza esta forma de comunicación, el sistema operativo debe proporcionar servicios adicionales como son crear y destruir un buzón. Esta comunicación indirecta es muy versátil ya que puede proporcionar enlaces entre procesos emisores y receptores uno a uno, uno a muchos, muchos a uno y muchos a muchos. La comunicación uno a uno supondría tener un buzón específico para los dos procesos, sería un canal privado entre los dos procesos. La comunicación muchos a uno es útil para procesos servidores donde tendremos muchos emisores (procesos de usuario) y un único receptor (proceso servidor). Si en este caso fuese importante conocer la identidad del emisor, ésta se puede dar dentro del mismo mensaje.
Cada sistema operativo gestionará los tipos de enlaces de distinta forma, normalmente permitiendo a procesos

ser "propietarios" de un buzón, pudiendo ampliar o transferir ese derecho de propiedad, eligiendo el conjunto de posibles emisores o receptores, etc. El sistema operativo también debe incluir una sección de recogida de "residuos" para los buzones que dejan de ser operativos.

2.3.3 COPIA DE MENSAJES

Por definición, el intercambio de mensajes entre dos procesos transfiere el mensaje desde el espacio de direcciones del emisor al espacio de direcciones del receptor. Esta transferencia puede ser copiando el mensaje dentro de la memoria del proceso receptor o pasándole un puntero al mensaje, o sea, el paso de mensajes puede ser por valor o por referencia.

Seleccionar uno u otro supone la elección de la seguridad o la eficacia. Los mensajes copiados son más seguros, ya que el emisor y el receptor quedan desacoplados y cada uno puede manejar los datos a su antojo. A su contra tienen el hecho de que consumen memoria y tiempo de CPU. Estos problemas hacen que algunos diseñadores opten por pasar simplemente un puntero al mensaje. La desventaja de este método es que hay que sincronizar a los dos procesos para utilizar correctamente la información del mensaje (no modifique un proceso parte del mensaje mientras el otro lo está usando).

2.3.4 ALMACENAMIENTO INTERMEDIO.

Aquí se nos presenta el problema de si los mensajes enviados pero no recibidos todavía, deben ser almacenados o no. Si el sistema no almacena los mensajes, tendremos una comunicación *síncrona* entre dos procesos, ya que si un emisor desea mandar un mensaje y el receptor no está preparado debe de esperar a que se prepare y de esta forma se sincroniza con el receptor.

Las ventajas del mecanismo síncrono emisor-receptor son la implementación sencilla y la seguridad por parte del emisor de que el mensaje ha sido recibido al acabar la operación *send*. La desventaja es la sincronización forzosa, que en algunos casos no sería deseable como en el caso del proceso servidor descrito antes.

Una alternativa a este mecanismo, permitiendo almacenamiento intermedio de los mensajes, es la comunicación *asíncrona* entre emisores y receptores. Si el receptor no está preparado para recibir el mensaje, el emisor lo envía al sistema operativo para su envío posterior al receptor. De esta forma, un proceso emisor puede seguir su ejecución sin preocuparse de las actividades de los receptores. Este modo de funcionamiento tiende a aumentar el grado de concurrencia del sistema. Este mecanismo, aunque muy útil, puede ser peligroso si se abusa de él, ya que un proceso podría mandar mensajes incontroladamente y completar rápidamente la capacidad de almacenamiento intermedio del sistema, bloqueando el resto de

posibles mensajes. Una forma de aliviar este problema es poner un límite en el número de mensajes para cada par emisor-receptor o para cada canal de comunicación.

2.3.5 LONGITUD

El problema que vamos a discutir aquí es si los mensajes deben tener una longitud fija o variable. En este caso, la elección es facilidad de implementación o flexibilidad. Los mensajes de tamaño fijo suelen tener poca sobrecarga al permitir que las memorias intermedias tengan también tamaño fijo. Los problemas vienen por los mensajes muy pequeños, que desperdician espacio y por los muy grandes, ya que se necesitan procesos suplementarios para dividirlos en trozos.

La alternativa de un tamaño variable para los mensajes alivia estos problemas creando dinámicamente espacio en memoria para encajar el tamaño de cada mensaje. El problema en este caso es esa gestión dinámica de memoria que puede consumir mucho tiempo de CPU, especialmente con el problema de la fragmentación.

Existe otra opción que trata de recoger las ventajas de ambos métodos. Se trata de tener una serie de tipos de mensajes, que son de longitud fija, pero distinta entre ellos. De esta manera, se podrá optar en todo momento por el tamaño de mensaje que más nos interese.

2.4 PASO DE MENSAJES ASÍNCRONO

Con el paso de mensajes asíncrono los canales de comunicación son colas de mensajes separadas. Un proceso agrega un mensaje al final de una de las colas

del canal ejecutando una sentencia de envío(*Send*) la cual no bloquea el proceso de envío *Sender*.
Un proceso recibe el mensaje de un canal ejecutando una sentencia *Receive*, la ejecución de *Receive* retrasa el *Receiver* hasta que el canal no esté vacío, entonces el mensaje al frente del canal es removido y puesto en variables locales por el *Receiver*.
Existen varios tipos de notación para representar paso de mensajes. La notación propuesta es simple, representativa y será definida durante el desarrollo de JMP.

2.4.1 FILTROS

Como se debe de haber notado una clave importante en el entendimiento de paso de mensajes es entender las suposiciones de comunicación, por lo que el desarrollo de un proceso requiere antes que nada especificar las suposiciones de comunicación basadas en los tipos de programas paralelos vistos en el capítulo anterior de la tesis.

Desde que la salida de un filtro es función de la entrada del mismo, una especificación apropiada es aquella que relaciona los mensajes de salida, como la entrada de otro canal de comunicaciones, esto es muy similar a un esquema de *pipeline* o tubería.
Para ilustrar cómo los filtros son programados tomemos el caso de ordenamiento de una lista de *n* números de forma ascendente.
La forma más sencilla de resolver este problema es escribir un proceso de filtrado único que reciba la

lista de números como entrada, los procese mediante un algoritmo de ordenamiento y escriba la salida hacia otro canal, la meta del proceso de ordenamiento es asegurarse de que todos los números están ordenados y si hubo un cambio de valores con respecto a la entrada.
Tomemos *sent[i]* como el i-ésimo elemento enviado a la salida de tal forma que la meta se podría definir como:

*SORT:(Vi:1<=i<n:sent[i]<sent[i+1])*Los valores enviados como salida son los mismos de la entrada de forma intercambiada.

El proceso de ordenamiento puede definirse como:

- Recibe los números del canal de entrada.
- Ordena los números.
- Envía los números por el canal de salida.

La función que recibe la lista de números es un bloque primitivo en el cual un problema a resolver es saber cuándo se han recibido todos los números; una solución es saber el número de elementos de la lista por adelantado, otra es enviar el primer valor de la lista como el total de elementos de la lista y una solución más genérica es usar un indicador de fin de cadena.
Si el sistema permite el ordenamiento de números(como un proceso pesado) como es el caso de la mayoría de los sistemas operativos entonces el problema seguramente se resolverá de manera sencilla y eficiente.

Sin embargo un punto de vista diferente puede proponer una red de pequeños procesos que se ejecuten en paralelo e interactúen para resolver el problema.
Hay varios tipos de ordenamiento de redes así como hay varios algoritmos de ordenamiento, de los cuales usaremos la fusión de redes (Merge Network).
La idea detrás de una fusión de redes es fusionar de manera repetida en paralelo dos listas ordenadas en una lista de mayor tamaño ordenada.
Cada proceso de fusión toma dos entradas ordenadas *in1*,*in2* y produce una salida ordenada *out*. Se asume que las cadenas van acompañadas de una señal de fin de lista *EOS* (*End of Stream*), esta señal es agregada por el proceso de fusión, de tal forma que si se tiene una lista de *n* valores, lo siguiente es cierto (Figura 2.3):

in1 y in2 están vacíos ^ sent[n+1]=EOS^SORT

Figura 2.3 Fusión

Una manera en la que se puede implementar lo anterior, es recibiendo las dos entradas como los canales a y b entonces las entradas se unen y se aplica un algoritmo de ordenamiento, sin embargo esto requiere ordenar todos los datos, por lo que se pueden comparar las listas de entrada elemento por elemento y enviando el menor como lista de salida, posteriormente se recorre la lista que tenía el elemento menor y se repite el proceso, hasta que una de las dos listas no tenga más elementos, en ese caso se manda los elementos de la otra lista al final de la salida, por último cuando

las dos listas de entrada no tienen elementos se envía un EOS.

Las redes de filtros pueden usarse para resolver diversos problemas de programación, por ejemplo redes de multiplicación de matrices y de vectores.

La aplicación de un método de ordenamiento depende en gran medida del poder de los procesadores y del tipo de red.

2.5 PASO DE MENSAJES SÍNCRONO

Una sentencia de paso de mensajes síncrono puede contener varios mensajes encolados, lo que puede contener diversas desventajas de las cuales podemos resaltar las siguientes:

- Si el proceso A envía un mensaje al proceso B y requiere saber que en efecto éste lo recibió entonces debe de esperar una respuesta de B.

- Si el proceso A envía un mensaje al proceso B y éste no responde, el proceso A no sabe si hubo un problema que interrumpió el proceso B o si bien el proceso sí se realizó y faltó solamente la respuesta que se perdió o bien no fue enviada.

- Los mensajes tienen que ser enviados dentro de una cola de mensajes, es decir deben de estar dentro de un *buffer* por lo que consumen recursos y por consiguiente se debe tener en cuenta que en

la realidad el espacio destinado para el *buffer* es finito.

El paso de mensajes síncrono evita estas consecuencias; para este tipo de paso de mensajes el envío (*send*) y la recepción(*receive*) son primitivas de bloqueo, esto se explica como el mutuo bloqueo entre procesos; si un proceso A trata de enviar un mensaje por un canal, no lo podrá hacer hasta que el proceso B reciba su mensaje de respuesta, por lo que el envío y recepción son sincronizados en cada punto de comunicación.
Como se puede observar se tienen dos puntos importantes con el paso de mensajes síncrono, por un lado se resuelven diversos problemas ocasionados por el uso de un *buffer*, pero por el otro, algunos algoritmos son más difíciles de implantar, por lo que uno de los propósitos de la presente tesis es usar paso de mensajes síncrono dejando al desarrollo del proyecto las mayores dificultades, de tal forma que el usuario final no tiene que trabajar en los problemas que el paso de mensajes síncrono contiene.

2.5.1 SENTENCIAS DE COMUNICACIÓN

Un proceso de comunicación con periféricos está compuesto de sentencias de entradas y salidas. Los procesos que comparten variables de comunicación entre ellos están compuestos de sentencias de asignación, el paso de mensaje síncrono toma ventaja de estos dos conceptos, las sentencias de entrada y salida proveen

la base de procesos de comunicación y el efecto de la comunicación es similar al efecto de una sentencia de asignación.

Supóngase que A pretende comunicarle el valor de z a B, esto se puede describir de la siguiente manera (figura 2.4):

```
A:: ...B!z ...
B:: ...A?m ...
```

Figura 2.4 A comunica Z

B!z es una sentencia de salida, es decir una sentencia (*output*), esto nombra el proceso de destino, que en este caso es B y especifica una expresión z la cual contiene el valor por ser enviado a B. A? es una sentencia *input* lo que identifica al proceso A como la fuente o el origen de donde se obtiene el valor en el proceso B que será guardado en *m*.

El símbolo ! es conocido como (*Bang*) y el símbolo ? es llamado (*Query*), de tal forma que si los tipos de *m* y *z* son iguales, tanto el proceso de extraer el valor de *z* y enviarlo y el de recibir el valor de z y asignarlo a *m* deben de estar sincronizados mientras que la comunicación toma lugar, una vez que esto se realiza, ambos pueden tomar lugar de manera independiente.

Los procesos se comunican cuando están ejecutando una sentencia de comunicación *match*, esto es cuando ambos procesos están coordinados o emparejados, es decir que las sentencias *input* y *output* están en *match* si todas las piezas son compatibles entre ellos.

La definición de lo anterior de una forma más formal es la siguiente:

2.5.1.1 SENTENCIA DE COMUNICACIÓN *MATCH*

Una sentencia *input* y una sentencia *output* están en *match* si cumplen cada uno de los siguientes cuatro puntos.

1. La sentencia *output* debe aparecer en el proceso nombrado por la sentencia *input*.
2. La sentencia *input* debe aparecer en el proceso nombrado por la sentencia *output*.
3. Los identificadores de puerto (un puerto es un canal de comunicación singular en el proceso destino) son los mismos y si se presentan los valores subscritos son los mismos.
4. Todas las asignaciones del tipo $a_i:=b_i$ serán sentencias de asignación válidas.
   Donde *a* representa las variables de la sentencia *input* y *b* representa las expresiones de la sentencia *output*.

2.5.2 REGLAS DE PRUEBA.

Existen tres pasos para la construcción de una prueba de un programa que use paso de mensaje síncrono:

- Construir una prueba secuencial para cada proceso
- Mostrar que esas pruebas no se interfieren
- Y mostrar que todas las aseveraciones de comunicaciones están satisfechas, en este caso a diferencia de cuando se hace con procesos de

prueba de paso de mensajes asíncrono, no se debe de llevar el control de variables que muestren el estado de los *buffers* de mensajes.

2.5.3 REDES DE FILTROS

Como fue visto en el paso de masajes asíncrono la existencia de *buffers* implícitos es importante, sin embargo en el caso de paso de mensaje síncrono no se requieren, fuera de esto, la forma en que se programan las redes de filtros es muy similar en ambos casos.
El no tener *buffer* puede ser utilizado de buena forma, porque el proceso que envía(*sender*) sabe cuándo el mensaje fue recibido, sin embargo la ausencia del *buffer* puede complicar algunos algoritmos.
Un ejemplo de lo anterior puede ser el de dos procesos que tratan de enviar un mensaje de manera mutua, y ambos están esperando la respuesta del otro.

2.5.4 ALGORITMO DE *HEARTBEAT*

El Algoritmo de Heartbeat nos permite ordenar una lista de números dados en paralelo, como se mostró anteriormente, el algoritmo de ordenamiento para paso de mensajes asíncrono da buenos resultados, sin embargo tiene la mayoría de los problemas que se mencionaron anteriormente para paso de mensajes asíncrono.

En esta parte se presenta una técnica en la cual cada uno de los procesadores extrae información de los vecinos de manera recurrente.
Para un ejemplo se tienen dos procesos P1 y P2 y a cada proceso se le asigna un valor igual a *n/2* en donde *n* es el total de números a ordenar, por lo que se tomará *n* par, con la finalidad de facilitar el ejemplo.

Primero cada uno de los procesos toma su lista de valores asignada y la ordena en el orden que se desea ordenar la lista final, es decir de manera ascendente o descendente; entonces los procesos extraen valores entre ellos hasta que P1 tiene todos los valores más pequeños y P2 tiene todos los valores más grandes.
Por separado en cada paso P1 le manda a P2 una copia del valor más grande y P2 le manda a P1 una copia del valor más pequeño, entonces cada uno de los procesos inserta el valor que recibió dentro de su lista de valores en el lugar propio de ese número.
El algoritmo termina cuando el valor más grande de P1 no es mayor al más pequeño de P2.
Un ejemplo es como se muestra en la figura 2.5:

```
L={5,3,1,6,2,4}
P1={5,3,1}
P2={6,2,4}
```

P1	P2
1,3,5	2,4,6
1,3 5 →	2 4,6 ←
1,2,3	4,5,6
3<4	4>3
123456	123456

Figura 2.5 Algoritmo de *heartbeat* para dos procesos

Como se puede ver el algoritmo funciona para dos listas y por tanto para dos procesos, por lo que se debe de adaptar para usarse en más de 2 procesadores; la manera es la siguiente:

Se tienen 4 procesos y una lista de cuatro números por ordenar, L={4,3,2,1}

Se definen *rounds*, como la competencia entre dos procesos los cuales van a intercambiar datos entre ellos usando el algoritmo antes descrito, para el ejemplo y la facilidad de comprensión del mismo se usa, un solo número por proceso.

Round		P2		P4
0	4 ⟺		2 ⟺	
1		4 ⟺		2
2	3 ⟺		4 ⟺	
3	1	⟺		4
4	1	2	3	4

Figura 2.6 Algoritmo de *heartbeat* para cuatro procesos

Esto se extiende fácilmente de tal forma que cada celda en lugar de tener un número, puede tener una lista de valores, de tal forma que la competencia entre ellos se realiza como el ejemplo de dos procesos.

La pregunta que resta responder es cuándo se han ordenado los números, es decir cuándo finaliza el algoritmo; una manera es mediante un nodo que se encargue de llevar un control de los cambios que cada una de las listas realizó internamente, cuando ninguna de las listas realiza ningún cambio, entonces el algoritmo termina.

La ventaja del método anterior es que si la lista está prácticamente ordenada, entonces el algoritmo finaliza casi de inmediato, el problema es que aumenta el número de mensajes, puesto que se debe de estar enviando el estatus de cambios, por lo que una manera más general, en el algoritmo de dos procesos sólo un *round* es requerido, por lo que el número de *round's* requeridos es igual al número de procesos usados menos uno.

2.5.5 MULTIPLICACIÓN DE MATRICES POR VECTORES

Para el caso en el que se pretenden multiplicar matrices por vectores se considera por simplicidad el caso en el que se tiene una matriz cuadrada y un vector de la misma dimensión *n*, el propósito es obtener el vector resultante en la figura 2.7:

$R_n = m_{nxn} \text{ X } v_n$

Figura 2.7 Vector resultante

Esto requiere realizar *n* productos internos, uno por cada elemento de *v* por lo que cada vector que resulta está definido en la figura 2.8.

```
R[i]=m[i,1]*v[1]+.....+m[i,n]*v[n]
```

Figura 2.8 *n* productos internos

El algoritmo secuencial de esto es muy simple, está conformado por dos ciclos en los cuales el primero define el i-ésimo elemento y el ciclo interno se encarga de realizar las sumatorias de los *n* valores.
Para solucionar este problema es necesario que se empleen *n* diferentes procesos los cuales ejecutan el proceso secuencial antes descrito, sin tener el doble ciclo ya que para cada uno de los valores del i-ésimo elemento se asigna un proceso nuevo. Esto se puede resumir como que cada R[i] se calcula de manera paralela, en un proceso independiente, el cual sólo envía como respuesta el valor R[i].
Para la multiplicación de matrices esto se extiende a multiplicar a la matriz por los *n* vectores de la segunda matriz.

2.6 CONDICIONES DE EXCEPCIÓN.

Los sistemas de paso de mensajes resultan útiles en entornos distribuidos. En estos entornos es más fácil que se produzca un error que en los sistemas centralizados, aunque tienen la ventaja de que si se produce un fallo, no necesariamente todo el sistema falla. En los sistemas centralizados, los mensajes se suelen implementar en memoria compartida, por tanto, si se produce un fallo, todo el sistema falla.

Cuando se produce un fallo, tanto en un sistema como en otro, hay que realizar algún tipo de recuperación de error (manejo de condiciones de excepción). Veamos a continuación alguna de estas condiciones:

2.6.1 TERMINACIÓN DE UN PROCESO IMPLICADO EN UN MENSAJE.

Tanto un emisor como un receptor pueden acabar antes de que se procese un mensaje.

Esta situación deja mensajes que nunca serán recibidos o procesos esperando mensajes que nunca serán enviados. En estos casos el sistema debe de tomar las medidas oportunas, tales como notificar la terminación de un proceso a sus posibles receptores o emisores.

Si tenemos un sistema asíncrono, el emisor no tendría problemas ya que colocaría su mensaje y seguiría su ejecución despreocupándose de si el mensaje se recibe o no. Sólo tendría problemas si necesitase acuse de recibo por parte del receptor. El caso del receptor es distinto, porque se queda esperando que aparezca un mensaje que no ha sido enviado. Para evitar este problema, algunos sistemas usan una variante de la operación *receive*, en forma no bloqueante, de tal forma que permite recibir un mensaje si existe, pero no bloquea al proceso si no existe.

Otra variante es proporcionar una facilidad para establecer un límite de tiempo para completar una operación de mensaje. Si transcurrido ese tiempo, no se completa la operación, los procesos continúan su ejecución sin bloquearse.

2.6.2 MENSAJES PERDIDOS.

Un mensaje cualquiera puede perderse dentro de la red de comunicaciones debido a múltiples causas. El método más usual para detectar si un mensaje se ha perdido es el empleo de un compás de espera como se vio en el apartado anterior (también denominados *time-outs*).
Una vez superado dicho tiempo, se supone que el mensaje se ha perdido, y si se da el caso, se procederá a la repetición del mensaje.

Básicamente hay tres métodos para tratar esta situación:

1. El sistema operativo es el responsable de detectar esa situación y de reenviar el mensaje.
2. El proceso emisor es el responsable de la detección de la situación y procede como considere oportuno.
3. El sistema operativo es el responsable de la detección y se encarga de notificarla al proceso emisor. El proceso emisor puede entonces proceder como desee.

No siempre es necesario detectar los mensajes perdidos, el usuario debe especificar esa situación.
Otro problema surge cuando se repite algún mensaje, pero el mensaje original no se había perdido, sino simplemente retrasado. En estos casos, debe de haber algún mecanismo para detectar cuando un mensaje es copia de otro o no.

CAPÍTULO III

DESARROLLO

3.1 PROCESO DE DESARROLLO

Este proceso consiste en un conjunto de actividades y resultados asociados que generan un producto de software. Estas actividades son llevadas a cabo por los ingenieros de software. Existen tres actividades fundamentales de procesos que son comunes para todos los procesos del software. Estas actividades son:

- Especificación del software
- Desarrollo del software
- Validación del software

Distintos procesos del software organizan estas actividades de diferentes formas y las describen con diferente nivel de detalle, el tiempo de cada actividad varía así como sus resultados.

3.1.1 MÉTODOS DE INGENIERÍA DE SOFTWARE

Un método de ingeniería de software es un enfoque estructurado para el desarrollo del software cuyo propósito es facilitar la producción del software de alta calidad a un precio costeable. Existen varios métodos como Yourdon, Jackson, de los años 70 que fueron siendo complementados por métodos orientados a

objetos los cuales se han ido integrando en un solo enfoque unificado basado en UML (Lenguaje de Modelado Unificado).

La presente tesis utiliza UML como herramienta principal de las 4 etapas de desarrollo Análisis, Diseño, Construcción y Pruebas con un enfoque orientado a objetos.

- El análisis genera un Documento de requerimientos y un Diagrama de Casos de Uso (Figura 3.1) así como la especificación de los mismos.
- El Diseño genera un ....
- La construcción genera el producto de Software que en este caso es el API de programación paralela con Java, así como una descripción de cada uno de los constructores y métodos disponibles.
- En las pruebas se construye un ejemplo básico y sencillo que utilice el API de programación paralela así como los resultados obtenidos.

3.2 ANÁLISIS

En esta parte se presenta una descripción del problema así como un documento de requerimientos basado en el estándar IEEE, por último se describen los casos de uso presentados en su correspondiente diagrama.

3.2.1 DOCUMENTO DE REQUERIMIENTOS

I.-Prefacio

Este documento está dirigido a investigadores y arquitectos de software en general que tengan la necesidad de enriquecer la biblioteca de desarrollo de programación paralela en Java desarrollada en la presente tesis.

Al ser ésta la primera versión no se tiene una historia previa por lo que no se justifican mejoras sobre versiones anteriores, sin embargo al integrarse de manera transparente al lenguaje Java se puede justificar como una extensión más al lenguaje, que permite usarlo en proyectos de investigación totalmente distintos a los que comúnmente se utiliza.

II.- Introducción

La programación paralela involucra herramientas complejas que no permiten que su uso sea sencillo para investigadores, por lo que se requiere una herramienta en un lenguaje común y corriente ampliamente usado y sobre todo muy bien estructurado como lo es Java.

El procesamiento paralelo involucra varios factores como la comunicación entre varios procesadores, los cuales pueden estar o no en la misma máquina, la sincronización de los diferentes procesos, instrucciones sencillas y fáciles de usar que permitan al programador enfocarse en los algoritmos o procesos que le resuelvan sus problemas y proyectos.

Lo anterior debe ser resuelto de forma tal que el usuario no deba tener una noción profunda del lenguaje Java, lo cual hace que la herramienta deba realizar algunos procesos de manera autónoma.
La integración con herramientas y sistemas previamente desarrollados en Java es de manera transparente por lo que es compatible con cualquier ambiente de desarrollo y cualquier sistema operativo que soporte Java.

III.- Glosario

JPI - Java parallel Iterface.
Browser – navegador de Internet.
Unix – Sistema operativo.
Puerto - Subdirección lógica del sistema operativo correspondiente a una IP.
URL – Universal request lenguaje.
RMI – Remote method iterface.
Class loader - cargador de clases.
Virtual machine – Máquina Virtual de Java de Sun.
Thread – Proceso ligero.
API - (Conjunto de bibliotecas comunes) Application Public Interface.

IV .- Requerimientos Funcionales

El API debe proporcionar una clase que se ejecute como nodo maestro.

El API debe proporcionar una clase que se ejecute como nodo cliente.

El API debe proveer una instrucción que permita conectarse a cada uno de los nodos.

La instrucción de conexión debe responder verdadero o falso dependiendo si fue exitosa o no.

En caso de que la conexión no sea exitosa debe de responder la IP y puerto que no permitió la conexión.

El API debe proveer una instrucción de especificación de dirección IP.

El API debe proveer una instrucción de especificación de dirección Puerto.

EL API debe proveer una opción para cargar funciones las cuales se ejecutarán en los nodos cliente.

El API debe proveer una instrucción que defina el número de clientes.

El API debe proveer una instrucción que permita obtener el estado de cada uno de los nodos.

El API debe proveer una instrucción que permita ejecutar un método en un nodo seleccionado.

El API debe permitir volver a ejecutar programas sin necesidad de volver a ejecutar los programas cliente.

V.-Requerimientos no Funcionales.

El API debe ser portable a cualquier sistema operativo que soporte Java.

El API debe funcionar con cualquier versión de JSE superior a la *JSE* 1.3.

El API debe funcionar en cualquier *RAD* que soporte Java.

Las aplicaciones generadas con el API deben de tener alto *performance* (desempeño).

El espacio físico en disco no debe ser incrementado en un porcentaje mayor al 5% de lo que utiliza la *Java virtual machine.*

El API debe funcionar con *Java virtual machines* optimizadas para un sistema operativo.

3.2.2 CASOS DE USO

Los casos de uso que surgen a partir del documento de requerimientos son los siguientes:

- Escribir funciones
- Iniciar nodos cliente
- Conectar clientes
- Ejecutar instrucciones
- Obtener resultados
- Iniciar nodo maestro

3.2.2.1 DESCRIPCIÓN CASO DE USO ESCRIBIR FUNCIONES

Flujo de eventos principal: El usuario escribe cada una de las funciones que va a utilizar en una clase llamada *jpi.java*, debe compilar el archivo con *javac* y verificar el lugar en el que se generó *jpi.class*

Flujo de eventos excepcional: Cada una de las funciones debe seguir el estándar de escritura de *beans*, sin ser esto último necesario para su funcionamiento.
Flujo de eventos excepcional: La localización del archivo puede estar en cualquier ruta de la máquina que contiene el nodo maestro, pero de preferencia debe ser la misma en la que se encuentra el código del programa principal del nodo maestro.

3.2.2.2 DESCRIPCIÓN CASO DE USO INICIAR NODOS CLIENTE

Flujo de eventos principal: El usuario toma cada uno de los nodos y ejecuta la clase cargador con la instrucción *c:\java cargador* en caso de que no exista ningún problema el programa devuelve la salida *esperado entrada...*

Flujo de eventos excepcional: En caso de surgir algún problema con la ejecución del programa cliente se genera una excepción de Java que se presenta en la pantalla.

3.2.2.3 DESCRIPCIÓN CASO DE USO CONECTAR CLIENTES.

Flujo de eventos principal: El usuario incluye la siguiente línea de código al principio del programa maestro (Figura 3.1).

```
Import master.*;
```

Figura 3.1 Principio del programa

Después extiende la clase de su programa de la siguiente manera (Figura 3.2).

```
Public class miEjemplo extends master
```

Figura 3.2 Extiende clase master

El usuario establece la ruta donde se encuentra el archivo *jpi.class* mediante la instrucción que se muestra a continuación en la figura 3.3:

```
setPath(C:\);
```

Figura 3.3 ruta de archivo jpi.class

El usuario establece cada uno de los nodos que piensa utilizar con la instrucción setChannel que toma como parámetros la dirección IP y el puerto, de la siguiente manera (Figura 3.4):

```
setChannel("172.20.4.40","8049");
setChannel("172.20.4.41","8049");
```

Figura 3.4 Establece canales

El usuario establece el número total de nodos con la instrucción *numch* como se muestra a continuación (figura 3.5):

```
numch=2;
```

Figura 3.5 Número de canales.

Por último ejecuta la instrucción *connectChannels()* para conectar a todos nodos.

Flujo de eventos excepcional: La instrucción de la clase maestro puede ser omitida si el archivo de la clase que se está escribiendo y la de la clase *master* están en el mismo archivo.

Flujo de eventos excepcional: Pueden existir varios nodos en una sola máquina utilizando un puerto diferente para cada uno de ellos (figura 3.6).

```
setChannel("127.0.0.1","8049");
setChannel("127.0.0.1","8050");
```

Figura 3.6 Varios nodos en el mismo equipo

3.2.2.4 DESCRIPCIÓN CASO DE USO EJECUTAR INSTRUCCIONES

Flujo de eventos principal: El usuario ejecuta la instrucción mediante el uso de un objeto *chmanager*, para eso debe hacer referencia al nodo que quiere usar como cliente comenzando por cero hasta los *n* nodos que se han declarado, en el mismo orden que fueron declarados con *setChannel* como se muestra:

```
t[0].enable("getSuma(1,2,3)");
```

Figura 3.7 Ejecutar Instrucción

Donde t[0] hace referencia al nodo cero y la instrucción *enable* le indica que ejecute el comando que está entre comillas, el cual es parte de las funciones del archivo jpi.class.
Flujo de eventos excepcional: El usuario no requiere declarar el objeto t[] puesto que se hereda de la clase *master*.
Flujo de eventos excepcional: El usuario puede validar si el nodo está disponible antes de ejecutar la instrucción mediante el uso de la instrucción let() la cual devuelve un valor *boleano.*

```
boolean b=t[0].let();
```

Figura 3.8 Nodo disponible

3.2.2.5 DESCRIPCIÓN CASO DE USO OBTENER RESULTADOS

Flujo de eventos principal: El usuario llama al método *getRes*() el cual devuelve *null* o una cadena con el resultado; posteriormente hace la conversión al tipo deseado usando los atributos de la clase *String*.

```
t[0].getRes()
```

Figura 3.9 Obtiene resultado

3.2.2.6 DESCRIPCIÓN CASO DE USO INICIAR NODO MAESTRO

Flujo de eventos principal: El usuario guarda el archivo miejemplo.java y lo compila con javac, posteriormente obtiene miejemplo.class y ejecuta la clase miejemplo con la instrucción de la figura 3.10

```
c:\java miejemplo
```

Figura 3.10 Inicia nodo maestro

En caso de que no exista ningún problema el programa devuelve la salida *esperado entrada...*

Flujo de eventos excepcional: En caso de surgir algún problema con la ejecución del programa cliente se genera una excepción de Java que se presenta en la pantalla.

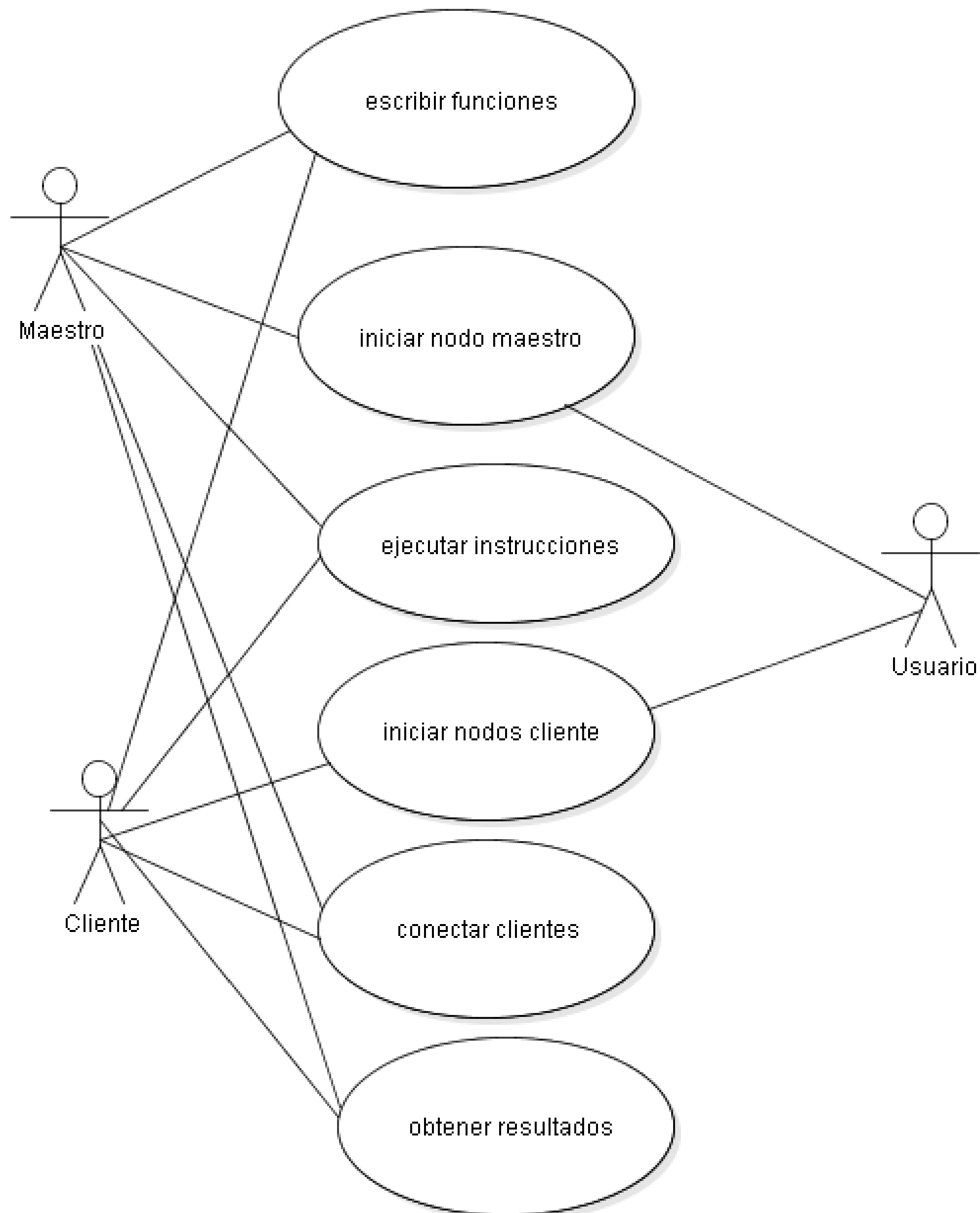


Figura 3.11 Diagrama de casos de uso.

3.3 DISEÑO

En esta etapa se genera la estructura que servirá de base para la construcción del API, para esto se hace uso de UML y se presentan los siguientes diagramas:

Diagrama de Clases en la Figura 3.12.
Diagrama de actividades en la figura 3.13.
Diagrama de secuencia figura 3.14

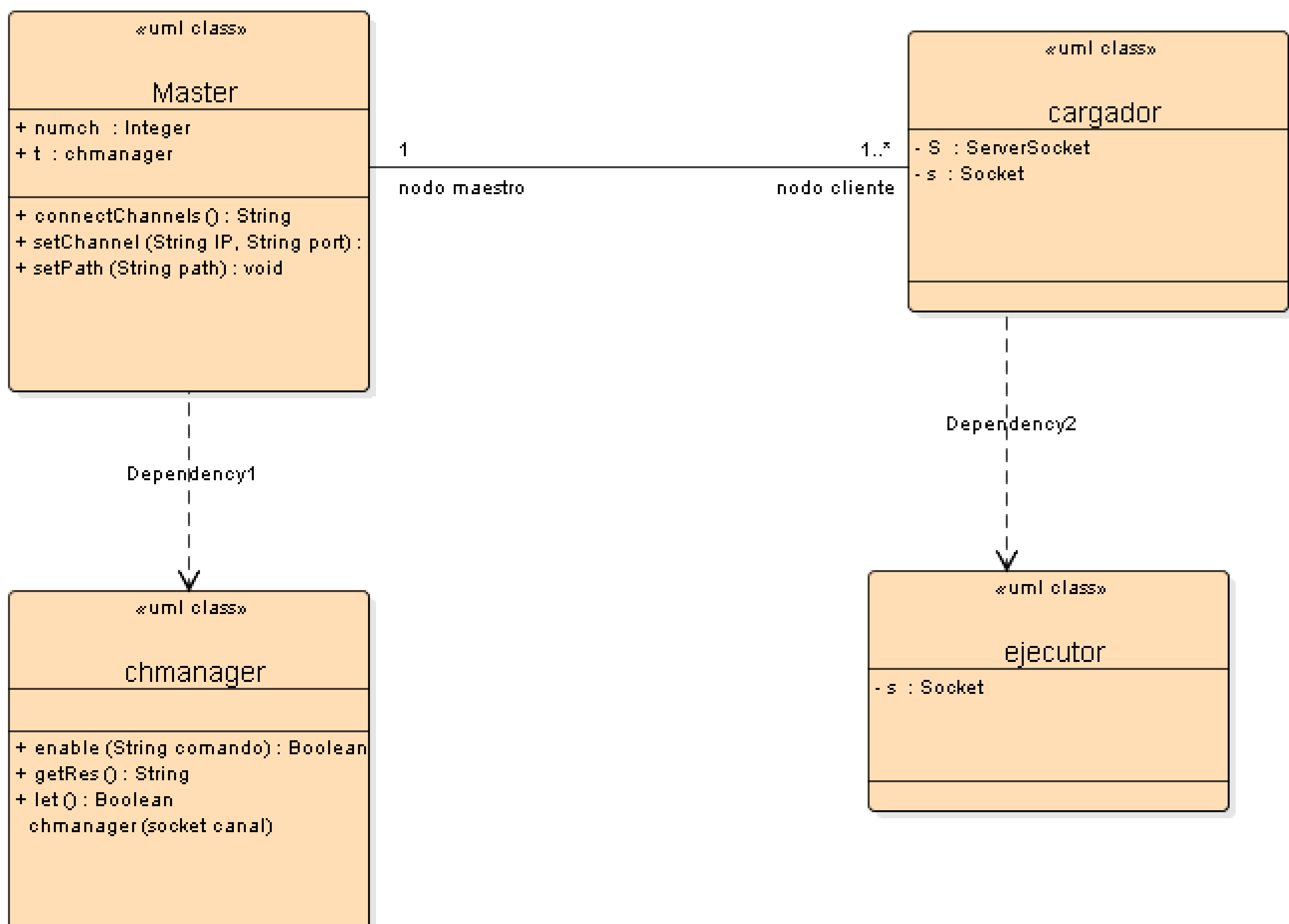


Figura 3.12 Diagrama de clases

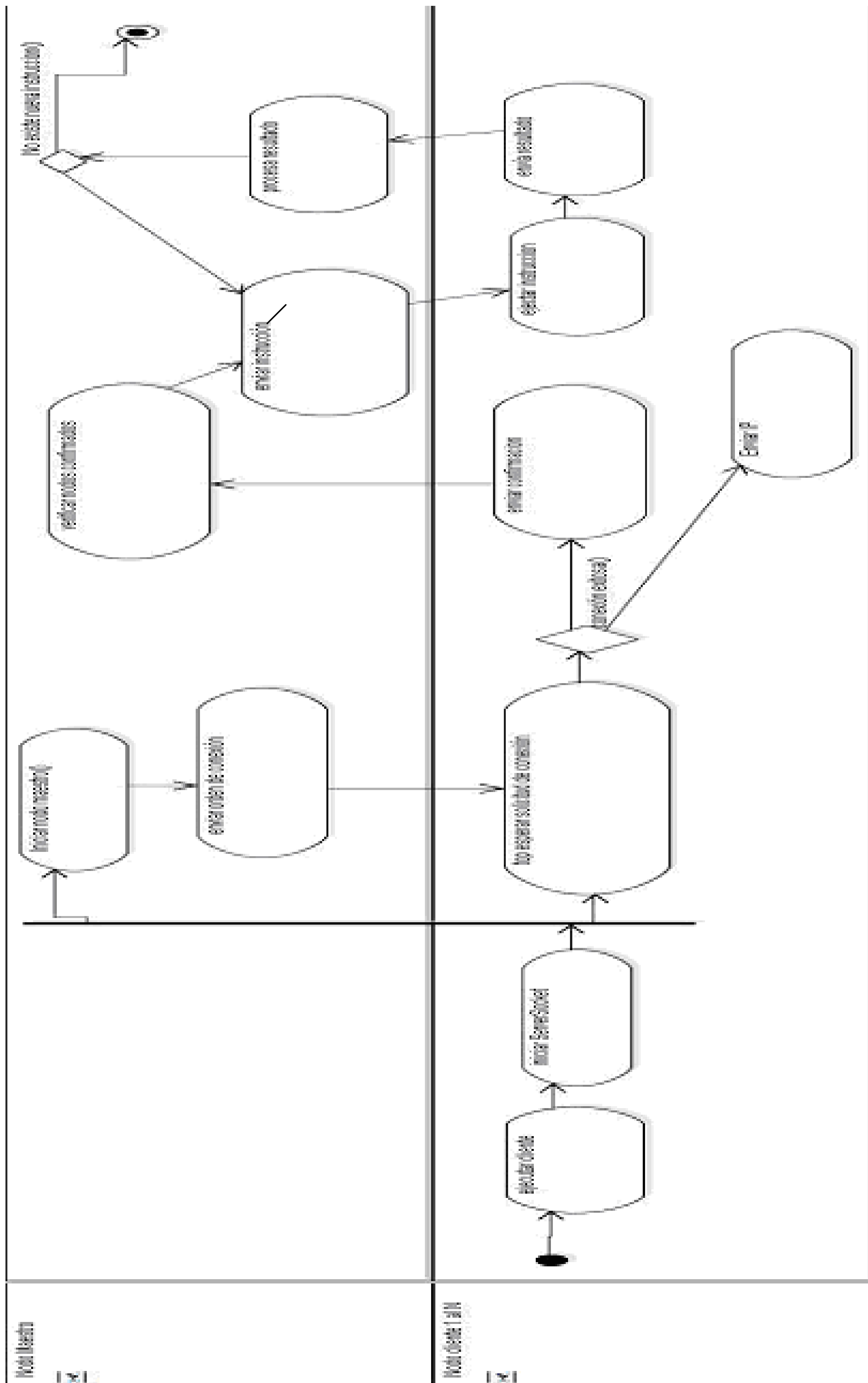


Figura 3.13 Diagrama de actividades

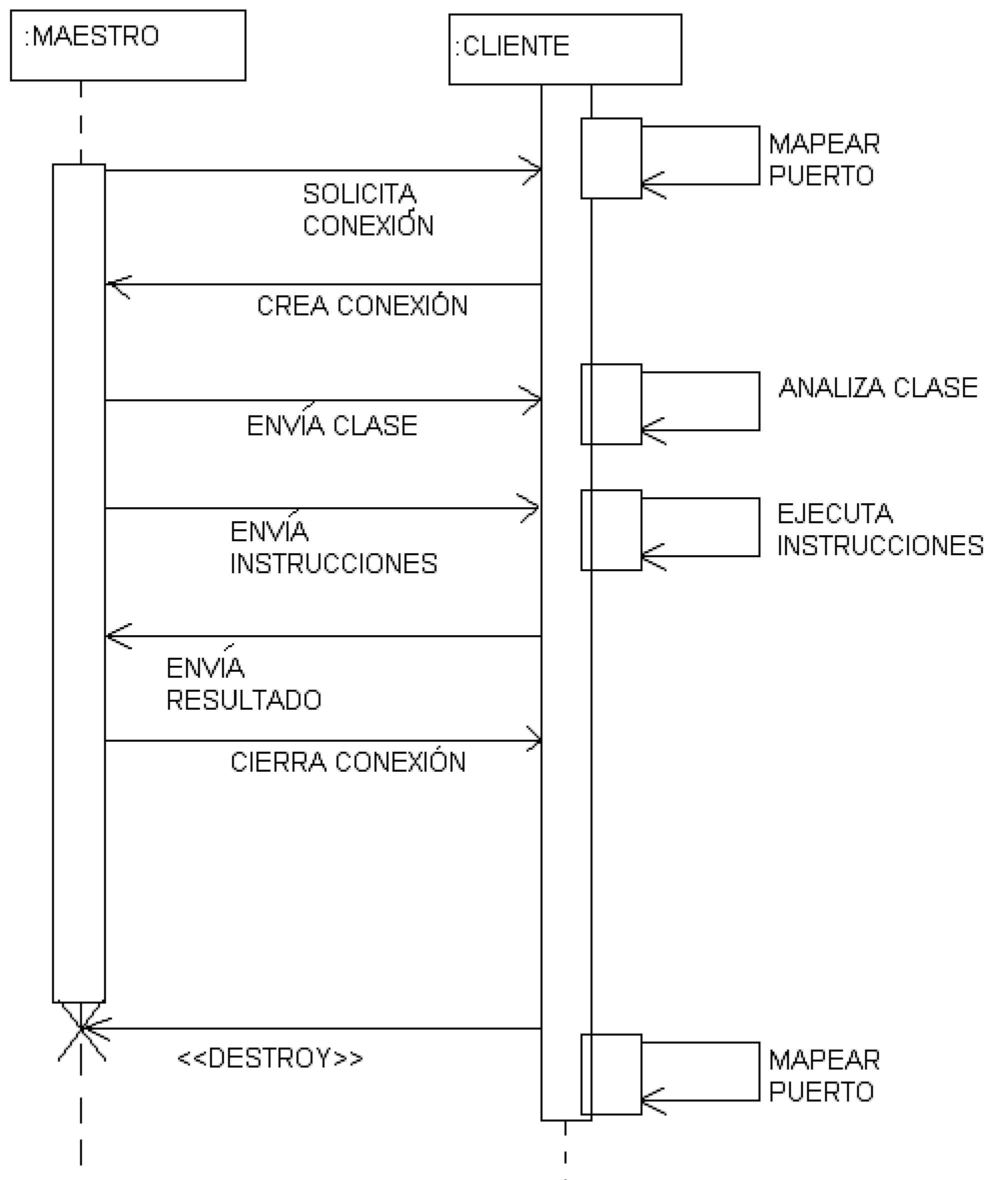


Figura 3.14 Diagrama de secuencia

Por último se presenta una estructura de clases generada para HTML.

	Package			
Class	Tree	Deprecated	Index	Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD

Class chmanager

```
java.lang.Object
  └ java.lang.Thread
      └ chmanager
```

All Implemented Interfaces:

java.lang.Runnable

---

public class **chmanager**

extends java.lang.Thread

Field Summary

Fields inherited from class java.lang.Thread
MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary
chmanager(java.net.Socket s)

Method Summary	
Boolean	**enable**(java.lang.String comando)
java.lang.String	**getRes**()
Boolean	**let**()
Void	**run**()

Methods inherited from class java.lang.Thread
activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getContextClassLoader, getName, getPriority, getThreadGroup, holdsLock, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setName, setPriority, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

chmanager

```
public chmanager(java.net.Socket s)
```

Method Detail

let

```
public boolean let()
```

---

getRes

```
public java.lang.String getRes()
```

---

enable

```
public boolean enable(java.lang.String comando)
```

---

run

```
public void run()
```

Package
Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FR

SUMMARY: NESTED | FIELD | CONSTR | METHOD DE

Class master

```
java.lang.Object
  └master
```

---

public class **master**

extends java.lang.Object

Field Summary	
static int	**numch**
static chmanager[]	**t**

Constructor Summary	
master()	

Method Summary	
static java.lang.String	**connectChannels**()
static void	**setChannel**(java.lang.String ip, java.lang.String port)
static void	**setPath**(java.lang.String p)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

numch

`public static int numch`

---

t

`public static chmanager[] t`

Constructor Detail

master

`public master()`

Method Detail

setPath

`public static void setPath(java.lang.String p)`

---

setChannel

```
public static void setChannel(java.lang.String ip,
                              java.lang.String port)
```

---

connectChannels

```
public static java.lang.String connectChannels()
```

Class cargador1

```
java.lang.Object
  └ cargador1
```

---

public class **cargador1**

extends java.lang.Object

Constructor Summary
cargador1()

Method Summary	
static void	main(java.lang.String[] args)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

cargador1

```
public cargador1()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

3.4 CONSTRUCCIÓN

En esta parte se describe la construcción tanto del cliente así como algunos de los problemas que surgen en la construcción de los mismos.

3.4.1 CONSTRUCCIÓN DEL CLIENTE

El cliente antes que nada debe ser simple en su instalación por lo que lo único que requiere es una máquina virtual de Java en cada uno de los nodos, con la única restricción que no sea un compilador *hit* habilitado desde el navegador de Internet ya que las restricciones de seguridad no permiten carga dinámica de clases en algunos de los casos.

La transmisión de los datos es mediante *sockets*, por lo que el cliente siempre está escuchando sobre un puerto que de no indicarse debe ser el 8084, el motivo es que para algunos sistemas UNIX los puertos de números pequeños son reservados y el involucrar un puerto de los antes mencionados puede requerir que el administrador instale la aplicación en los clientes, lo cual implica una dependencia del administrador UNIX en todo momento.

Una vez que el cliente se ejecuta, comienza a mapear el puerto 8084; cuando recibe una solicitud del nodo maestro, el cliente antes de realizar

cualquier otra tarea crea un *socket* para establecer la comunicación con el maestro, en este punto hay que señalar que el tipo de *socket* que se crea es distinto al que se usa para mapear el puerto, ya que es un tipo *Socket*, mientras que el *socket* que escucha en el puerto es un *ServerSocket*.

Al tener creado el *socket* que comunica el cliente con el maestro, se usa para recibir todas las peticiones del nodo maestro, es decir todas las instrucciones de ejecución de métodos que se encuentran en la clase que se carga de forma dinámica y posteriormente enviar todas las respectivas respuestas al nodo maestro; como se puede ver, el tipo de mensajes que se aplica es síncrono, debido a que el nodo no recibe más de una petición a la vez, aunque debido al multithread que utiliza tiene la capacidad de hacerlo, no está diseñado para eso, por lo que el modelo de programación que debe utilizar es el paso de mensajes síncrono, que se menciona anteriormente en la presente tesis.
Antes de que el nodo se encuentre listo para recibir las peticiones del nodo maestro, debe de cargar la clase que contiene todos los métodos que el nodo maestro va a usar, es decir, el nodo cliente al recibir su primer mensaje asume que se trata de la clase que contiene todo los métodos;

para realizar esto se propusieron dos métodos, el primero usaba la carga dinámica de la clase mediante un URL el cual permite cargar la clase directamente, de una forma similar a como lo hace RMI; en apariencia éste era el mejor método de carga de clases, pero al llevarlo a la práctica no fue tan eficiente, por dos razones principales:

- Se presentan problemas de seguridad.
- Es más lento que el segundo método propuesto.

El segundo método es más ineficiente, consiste en recibir el archivo jpi.class en formato binario mediante el *socket* de conexión; una vez que se tiene el archivo descargado se guarda en un archivo del mismo nombre, para poder invocar al *class loader* que de forma dinámica lo cargará como se menciona en partes previas de la presente tesis, los archivos jpi.class son de tamaño pequeño por lo que la transferencia es muy rápida, además ésta se realiza en el momento en que el nodo maestro reconoce cada uno de los nodos, es decir antes de empezar a ejecutar los programas.

La gran ventaja del método antes mencionado es que una vez descargado el archivo jpi.class la carga de la clase es más rápida y sobre todo más robusta, además de que al ser leída de la máquina cliente, la *virtual machine* no desconfía tanto del contenido, por lo que sólo hay que preocuparse de que se tengan los permisos de escritura de archivos en la máquina cliente.

Una vez que se tiene la clase cargada correctamente en el cliente se procede a analizarla, ya que en este punto el cliente no sabe absolutamente nada de la clase que recibió, es decir, el problema se complica porque el programa ya está en ejecución y no podemos pararlo y volverlo a ejecutar para que anexe la nueva clase con todos sus métodos.

El realizar lo anterior involucra hacerlo en cada uno de los métodos por lo que si tenemos un cluster de más de 20 procesadores por citar un ejemplo, las cosas se complican y todo se vuelve muy ineficiente, por lo que es indispensable que de forma dinámica el programa cliente sea capaz de cargar la clase y de analizarla de tal forma que sepa con qué métodos cuenta, el tipo de métodos que se tiene y por último el tipo de cada uno de los parámetros de cada método, una vez que se tiene todo esto, el programa debe ser capaz de invocar un método de la clase que cargó dinámicamente; es decir mediante el *socket* el cliente lo único que recibe es una cadena que contiene algo como *getSuma*(2,3), por lo que el cliente debe de ser capaz, después de analizar la clase, de saber que contiene un método *getSuma* con dos parámetros de tipo entero y que va a regresar un tipo *int*, todo esto para poder invocar el método *getSum*, es decir para invocar un método que no está en su código original.

Todo lo anterior se realiza mediante reflexión de Java, de tal forma que una vez que se tiene toda la información de la clase y una forma de invocar sus métodos, lo único que resta es realizar una búsqueda de las órdenes que recibe, posteriormente se requiere que los valores se conviertan en objetos del tipo antes detectado por lo que la interfaz internamente cuenta con un método que recibe dos cadenas, una con el valor y otra con el tipo y devuelve un objeto referente al tipo, por ejemplo para un entero devuelve un tipo *Integer*, que será puesto dentro del arreglo de argumentos que se le pasan al invocador.

Después de realizar todo lo anterior sólo resta esperar que el invocador reciba la respuesta del método invocado, esto puede tardar bastante, depende totalmente de lo contenido en el método.

3.4.1.1 PROBLEMÁTICA PRESENTADA

El funcionamiento antes presentado puede complicarse cuando se requiere ejecutar una segunda clase, es decir, debido a cambios en el programa principal del nodo maestro o bien por cambios en los métodos de la clase jpi.class que se carga en los clientes de manera dinámica, se decide enviar una nueva petición al cliente, sin

embargo el cliente previamente ya había hecho ciertas cargas, lo que provoca una serie de errores internos que complican la manera en la que el cliente funciona internamente.
Otro gran problema es que cuando el nodo maestro se cierra los nodos clientes pierden la conexión y se genera una Java *exception*, por lo que dejan de ejecutarse, esto obliga a volver a levantar cada uno de los nodos.

La solución consiste en crear un ciclo que se encargue de mapear el puerto pero una vez que se recibe una petición en el nodo cliente se crea un *thread* en el cliente y se le pasa como parámetro el *socket* recién creado, el proceso ligero realiza todo su trabajo y muere cuando el nodo maestro termina o cierra sus conexiones, pero el programa principal del nodo cliente sigue esperando una llamada en el puerto, por lo que cuando el nodo maestro se vuelve a ejecutar se repite el proceso sin ningún problema.

3.4.2 CONSTRUCCIÓN DEL MAESTRO

El maestro es el nodo encargado de ejecutar el programa principal y por tanto de controlar a los clientes, para realizar lo anterior es necesario que el maestro herede todas las propiedades de la clase *master* que pertenece a jpi, esto es para facilitar el desarrollo y proteger el código de la clase *master*.

En resumen, los desarrolladores o los investigadores que la utilizan sólo se preocupan por su código, no por problemas de comunicación ni por la administración de procesos.

La clase *master* cuenta con un método de conexión y sincronización de canales, por lo que es capaz de comprobar que todos los canales que se le indicaron existen, están conectados y listos para recibir el archivo jpi.class; una vez que el método comprueba que todo está listo le transfiere el archivo jpi.class, con lo que todos los nodos están listos para recibir órdenes del programa principal.
El método anterior a su vez está controlando un *manager* de canales de conexión que es un proceso ligero que se crea cada que se tiene una conexión completa con el nodo.
El *manager* de canal tiene una tarea fundamental, enviar órdenes mediante el canal de comunicación previamente asignado y esperar que el nodo al que se le envió la orden responda, hay que resaltar que todo esto no traba o paraliza el flujo del programa principal porque se trata de *threads*, así que se utiliza la multitarea del sistema operativo en el que se corre el maestro, ya sea Windows o un tipo de Unix; el esperar la respuesta del nodo cliente, sin utilizar ningún tipo de encolamiento es la base de la programación de paso de mensajes síncrono, por lo que a pesar de que en apariencia la aplicación permite trabajar de manera asíncrona, es decir mandando más de

una orden a un nodo sin que éste termine de responder por la tarea anterior, su diseño básico está pensado en paso de mensajes síncrono.

El *manager* de canales también proporciona información valiosa con respecto a los nodos, como el estado actual del nodo, disponibilidad y resultados enviados por el nodo.

El *master* además se encarga de buscar el archivo jpi.class generado a partir de las clase jpi.Java, que como se mencionó anteriormente contiene todos los métodos que se van a ejecutar en los clientes.

3.4.2.1 LOCALIZACIÓN DEL ARCHIVO jpi.class

Como no todos los ambientes tienen las mismas variables de ambiente y las mismas configuraciones, se tiene que establecer la ruta en la que nuestro compilador Java genera el archivo jpi.class, es recomendable que la clase jpi esté dentro del archivo que extiende la clase *master*, como se muestra a continuación (Figura 3.15).

```
Public class miEjemplo extends master
{
....
}
class jpi
{
....
}
```

Figura 3.15 jpi dentro de master

El *master* una vez que termina su tarea debe destruir todos los procesos ligeros que creó y por último cerrar todas las conexiones que tiene con los nodos sin que esto quiera decir que los nodos están deshabilitados, por el contrario, los nodos están listos para recibir una nueva petición de sincronización.

CAPÍTULO IV

PRUEBAS

4.1 APLICACIÓN DE EJEMPLO

Escribir programas que usen la capacidad de multiprocesamiento de lo expuesto en la presente tesis, es sencillo, el objetivo es que las personas que lo realicen no deban de saber nada de reflexión, *sockets* o algo similar, por lo que se presenta un sencillo ejemplo en el que se asignan tareas a 3 nodos cada uno de los cuales realizará una sencilla suma y regresará el resultado, para poder ilustrar la capacidad de esperar un resultado y saber cuándo alguno de los nodos ha terminado de realizar su tarea; se agrega al programa de la suma un *sleep*, con el objetivo de simular lo que sería un tiempo de procesamiento largo, las sumas que se asignan a cada nodo son distintas y una vez que todos los nodos terminan de realizarlas se procede a sumar los resultados, con esto se pretenden ilustrar dos puntos básicos.

- La capacidad de procesamiento independiente en cada uno de los nodos.
- La capacidad de manipular los datos de los nodos de manera conjunta y a su vez realizar nuevos cálculos con los datos obtenidos de los mismos.

Como se ha mencionado previamente se debe de tener en cuenta que el paso de mensajes utilizado es síncrono, por lo que sólo se puede asignar una tarea a la vez a uno nodo.

4.2 DESCRIPCIÓN DEL PROGRAMA

La primera parte del programa es necesaria para que podamos heredar todas las características de la clase *master*, la cual es la clase principal; de ella vamos a heredar todas las características que requerimos (Figura 4.1).

```
public class prueba extends master
```

Figura 4.1 Extiende a master

Por supuesto la clase debe ser pública puesto que el programa es un aplicación y o un componente.

Dentro de la aplicación especificamos la ruta en la cual se encuentra la clase que contiene todos los métodos que deseamos utilizar en los nodos clientes, por lo que para este ejemplo se utiliza una máquina con windows 2000; la ruta es C:\ (Figura 4.2).

```
public static void main(String[] args)
  {
         setPath("C:\\");
```

Figura 4.2 Indica ruta

Hay que notar que no se requiere hacer referencia a la clase padre puesto que heredamos todas sus características.

Después de decirle al programa maestro dónde está el archivo que debe de tomar para ser procesado por los nodos, hay que indicarle cuáles son los nodos como se muestra (Figura 4.3).

```
setChannel("172.20.4.40","8049");
setChannel("172.20.4.39","8049");
```

Figura 4.3 Establece canales

La forma en que está definido *setChannel* consta de dos partes, o mejor dicho es un método con dos parámetros, una cadena que indica la IP del nodo y otra cadena que indica el puerto.

Esto como se ha mencionado previamente da una gran flexibilidad, puesto que permite hacer cosas como la siguiente (Figura 4.4):

```
setChannel("127.0.0.1","8049");
```

Figura 4.4 Establece nodo

La ventaja de lo anterior es que los nodos se encuentran dentro de la misma máquina, lo único que se debe hacer es cambiar el puerto de cada uno de ellos, ése es el motivo por el que se especifica el puerto, además de ser flexible en cuanto a no requerir un puerto específico.

La pregunta obvia es para qué tener todos los nodos dentro de una misma máquina, si lo que se busca es llevar a cabo multiprocesamiento, la respuesta consta de dos partes, la primera es que una máquina puede tener varios procesadores sin necesidad de tener un cluster, hay casos como el de *compaq_tru* 64 *cluster* en la que el sistema operativo maneja el *file system* como uno solo y el usuario no siente que se trate de un *cluster* sino de

una sola máquina, por lo que si se ejecuta el programa cliente dentro de la misma máquina, el sistema operativo crea un nuevo proceso, pero la diferencia de las máquinas de un solo procesador le asigna un proceso a cada uno de los procesadores, por lo que las ventajas se hacen mayores, ya que se tiene una arquitectura de una máquina con varios procesadores, corriendo programas paralelos que funcionan con las ventajas de paso de mensajes; la segunda parte tiene que ver con el hecho de no tener que usar máquina de varios procesadores para llevar a cabo depuración y pruebas de los programas, al poder ejecutar todos los nodos en una misma PC o estación de trabajo, se puede simular el funcionamiento de una máquina más grande; es claro que el rendimiento es pésimo puesto que sólo se tiene un procesador que reparte su tiempo entre todos los procesos, pero es una gran ventaja poder ejecutar los programas tal cual van a funcionar ya sea en el *cluster* o en la máquina paralela; de hecho se podría utilizar un *RAD* como *JDeveloper*, Java *Builder*, *NETBEANS*, *WEBSPHERE* para depurar el programa maestro y más aún si la máquina cuenta con la suficiente memoria y el *RAD* lo permite, se puede depurar cada uno de los nodos y el maestro al mismo tiempo, con lo que se toman las ventajas de los *RADS* actuales, las cuales sobrepasan las de herramientas de depuración de otros lenguajes de programación paralela mencionados en la presente tesis.

Una vez que establecimos los canales hay que saber cuántos nodos se tienen; es difícil que se tengan varias máquinas paralelas, por lo que esto es una parte constante, ya que el programador siempre usa la misma máquina paralela (Figura 4.5).

```
//establece número de canales
      numch=2;
```

Figura 4.5

Cuando toda la información que se relaciona con los nodos está lista, se deben conectar todos los canales con su respectivo nodo; para saber si la conexión fue la correcta, pintamos en pantalla la respuesta del método que es un *string* (Figura 4.6).

```
    System.out.println("La respuesta es:"+
connectChannels());
```

Figura 4.6

La función *connectChannels* no requiere parámetros y lo que hace es establecer la conexión de los *sockets* de cada uno de los nodos clientes además de enviar los *bytecodes* de la clase que contiene cada uno de los métodos que van a ser usados en la máquina cliente.

La respuesta puede ser un *OK* o en caso de error la *IP* de la máquina que no pudo conectarse y la razón por la cual no se pudo realizar la conexión o la carga dinámica de la clase.

La clase de la suma contiene un método *getRes* que devuelve el resultado de la suma de sus tres parámetros entero. Para poder invocar ese método en uno de los nodos se hace lo siguiente (Figura 4.7):

```
  //cadena con la orden para un determinado cliente
      String command=new String("getRes(20,30,1000)");

   boolean s;
      if(t[0].let())
      {
       s= t[0].enable(command);

while(t[0].getRes()==null||t[0].getRes().equals("null"))
       {
          System.out.println(sss+"El proceso  debe estar
corriendo"+command+"regreso el valor"+t[0].getRes());
       }
       System.out.println(sss+"El proceso w  debe estar
corriendo"+command+"regreso el valor"+t[0].getRes());

      }
```

Figura 4.7

Como era de suponerse cada vez que enviamos un proceso a ser ejecutado por uno de los nodos no podemos esperar siempre a que termine de ejecutarse y nos regrese la respuesta síncrona, para poder enviar otro mensaje, por lo que todo en el programa maestro funciona con *Threads*;

la ventaja es que no se requiere conocer los *threads*, ya que la clase *master* lo hace por sí sola.

En el ejemplo se hace lo opuesto a lo antes descrito, es decir, se obliga a que cada uno de los nodos termine su labor para continuar, pero como se puede observar no se requiere esperar a que esto suceda, simplemente se omite el *while* que espera el resultado y se lanzan todos de manera continua, posteriormente se puede verificar el estado de cualquiera de los nodos.

t[0].let() es un método booleano que es verdadero cuando el *master* logró hacer lo que debía en cuanto a *threads* se refiere, la clase *master* cuenta con un administrador interno el cual administra todos los procesos y por tanto dicha clase cuenta con un arreglo de *managers* llamado *t* por lo que se crean tantos *managers* como nodos se tienen, si el método *let* es verdadero quiere decir que el nodo está arriba y listo para recibir una orden, es por eso que hay que verificar primero que *t[0]* esté disponible, éste generalmente se refiere al primer nodo que dimos de alta, pero en este punto se decidió que el control de los nodos sea por parte del *manager* interno y no por parte del programador.

t[1].getRes() obtiene el resultado del método que se invocó por lo que es útil para saber el estado del nodo.

t[0].enable(command) es la instrucción que dice a uno de los nodos que ejecute ese método.

4.3 LISTADO COMPLETO

El listado completo del programa se muestra a continuación (Figura 4.8):

```
public class prueba extends master
{
  public prueba()
  {
  }
  public static void main(String[] args)
  {
        setPath("C:\\");

      //establece los canales de comunicación
      //setChannel("127.0.0.1","8048");
      setChannel("127.0.0.1","8049");
      setChannel("172.20.4.40","8049");
      //establece número de canales
      numch=2;
      //conecta con los canales de comunicación a cada
uno de los nodos y carga las clases en la memoria de
cada uno de los clientes
     System.out.println("la respuesta de conexión es:"+
connectChannels());
      //cadena con la orden para un determinado cliente
      String command=new String("getRes(20,30,1000)");
```

```
      //asigna un proceso ligero para que admnistre la
conexión con el nodo y ejecute la orden de command
    //  chmanager t=new chmanager(s[0],command);
       boolean s;
      if(t[0].let())
      {
       s= t[0].enable(command);

while(t[0].getRes()==null||t[0].getRes().equals("null"))
       {
        //  System.out.println(sss+"El proceso   debe
estar           corriendo"+command+"regreso          el
valor"+t[0].getRes());
       }
       System.out.println(sss+"El proceso w  debe estar
corriendo"+command+"regreso el valor"+t[0].getRes());


      }
      if(t[1].let())
      {
       sss= t[1].enable(command);

while(t[1].getRes()==null||t[1].getRes().equals("null"))
       {
        }
       System.out.println(sss+"El proceso nodo200  debe
estar           corriendo"+command+"regreso          el
valor"+t[0].getRes());
```

```
      }



     if(t[0].let())
     {
              sss= t[0].enable("getSum(20,30)");

while(t[0].getRes()==null||t[0].getRes().equals("null"))
               {
                   // System.out.println(sss+"El proceso
debe      estar      corriendo"+command+"regreso      el
valor"+t[0].getRes());
               }

        System.out.println(sss+"El proceso  último debe
corriendo        getSum(20,30)        regreso        el
valor"+t[0].getRes());

     }
     System.exit(0);
  }//end main
}//end class
```

Figura 4.8 Listado completo

CONCLUSIONES

- Los programas que son optimizados para usarse en memoria distribuida funcionan mejor que los usados para la memoria compartida en máquinas que presentan distribución modular.
- Las condiciones de carrera(*RACE CONDITIONS*) son una de las principales causas de errores, tanto en entornos de memoria distribuida como de memoria compartida.
- Las máquinas *MIMD* son las más usadas en la actualidad para cómputo paralelo por su relación precio rendimiento.

- EL modelo *PRAM* es ampliamente usado para hacer una idealización de una red perfecta en la cual no hay pérdidas.

- Los algoritmos que se usan en cómputo paralelo generalmente difieren de los usados por programas secuenciales, pero tienen un rendimiento mayor.

- Para el caso de paso de mensajes el algoritmo Maestro Esclavo es el más óptimo.

- Un programa de paso de mensajes se puede adaptar

para ambientes de un solo procesador, o bien para ambientes de memoria compartida.

- La carga dinámica de la clase que contiene los métodos a ejecutarse en los nodos mediante un *URL* el cual permite cargar la clase directamente, no es eficiente y está muy restringido por problemas de seguridad, por lo que la carga dinámica es mejor.

- El API JPI con Java permite a las personas con conocimientos no avanzados de Java, escribir programas que se ejecuten en varios procesadores con esquemas de memoria compartida o distribuida.

BIBLIOGRAFÍA.

Lonnie P. Hammack

Parallel Data Mining with the Message Passing Interface Standard on Clusters of Personal Computers (Spiral-bound)

U. De Carlini, U. Villano
Transputers and Parallel Architectures: Message-Passing Distributed Systems (Ellis Horwood Series in Computers and Their Applications) (Hardcover)

H. K Reghbati
An efficient message passing mechanism for multicomputer processing (Research reports / University of Saskatchewan, Dept. of Computational Science) (Unknown Binding)

G. W Stewart
Communication and matrix computations on large message passing systems (Computer science technical report series) (Unknown Binding)

CAY HORTSMAN GARY CORNELL
JAVA 2 Fundamentos Volumen I
SUN MICROSYSTEMS PRESS

CAY HORTSMAN GARY CORNELL
CORE JAVA 2 ADVANCED FEATURES
SUN MICROSYSTEMS PRESS

LEWIS BERG
ProgramaciÓn multithread en Java
SUN MICROSYSTEMS PRESS