Una Plataforma para el Desarrollo de Aplicaciones en Paralelo Usando Java

Erick Pinacho Rodríguez
Asesor: Darnes Vilariño Ayala

20 de agosto de 2007

Resumen

La búsqueda de soluciones a problemas reales generalmente requiere realizar una gran cantidad de cálculos. Esto ocasiona que la obtención de resultados sea demasiado tardada. La única solución para este problema se ofrece por el *Paralelismo*, que permite realizar varias operaciones a la vez, favoreciendo la reducción del tiempo que se requiere para ejecutar una tarea.

Desde hace varios años el hardware no es más una limitante. Se ha vuelto más importante el desarrollo de herramientas de software que permitan sacar ventaja de las posibilidades de hardware.

En cuanto a hardware, existen diversas arquitecturas paralelas hoy en día, y la librería de funciones más usada para el desarrollo de aplicaciones paralelas es MPI. Esta librería posee limitantes acerca del manejo de memoria, y es responsabilidad del desarrollador construír los mecanismos necesarios si desea emplear medios de sincronización complejos, tal como una memoria compartida.

El lenguaje Java cada día gana mayor aceptación en la comunidad científica e industrial, extendiendo no sólo su uso, sino también las herramientas diseñadas para este lenguaje.

Java ofrece mecanismos de concurrencia, manejo de funciones de bajo nivel y tecnologías para desarrollar aplicaciones distribuídas. Estas y otras características que posee Java, lo hacen un excelente candidato para desarrollar software paralelo.

En este trabajo se presenta una Plataforma para la ejecución en paralelo de aplicaciones desarrolladas en Java. Esta Plataforma fue desarrollada usando el mismo lenguaje, y está orientada para su uso en Clusters o redes LAN.

Índice general

Capítulo 1

Introducción

Muchos problemas reales, ya sea de aplicaciones comerciales, industriales, de simulación y de investigación, entre otros, requieren realizar una gran cantidad de cálculos sobre una gran cantidad de datos. La cantidad de información que se debe manipular ocasiona que la obtención de resultados no sea inmediata, y en algunos casos es demasiado tardada para la aplicación. En muchos de los casos, la única solución para lidiar con esas grandes cantidades de datos y los cálculos que se hacen sobre ellos, es ofrecida por el *Paralelismo*. Al poder realizar varias operaciones al mismo tiempo, es posible completar una tarea en un menor tiempo, comparada con el tiempo que le tomaría realizar todas las operaciones de esta tarea en forma secuencial.

Los cambios en el diseño y construcción de hardware han permitido la aparición de arquitecturas más potentes y veloces. En los años que se construyeron las primeras computadoras, el software exigía arquitecturas más potentes. Hoy en día las arquitecturas actuales poseen características que deben ser explotadas por el software. Muchas de estas arquitecturas poseen cierto grado de paralelismo en las operaciones que realizan a nivel de hardware. Hoy, el hardware ya no es más una limitante, y en cambio, se requiere de herramientas de software que permitan aprovechar las posibilidades que ofrece el hardware.

Existe una gran gama de herramientas de software para desarrollar aplicaciones en paralelo. Estas herramientas están diseñadas para aprovechar los recursos que ofrecen las arquitecturas actuales, tal como varios nodos de procesamiento, memoría distribuída y capacidad de almacenamiento, entre otros. Algunas de estas herramientas se encuentran en fase de estandarización, y otras ya son estándar *de facto*. Dentro de estas herramientas, la librería de funciones más usada para el desarrollo de aplicaciones paralelas es MPI (*Message Passing Interface*), que funciona con base en el paso de mensaje entre nodos de ejecución.

Sin embargo, MPI posee ciertas limitantes acerca del manejo de memoria. Muchas de las aplicaciones paralelas pueden ser modeladas de forma más natural usando un modelo de *memoria compartida*, en el que todo nodo de procesamiento tiene acceso a una única memoria que es vista por los demás nodos. Esta clase de comunicación de datos no puede ser modelado directamente usando MPI, y es responsabilidad del

programador desarrollar los mecanismos que simulen estas funciones.

Otra de las limitantes que posee MPI, es que está diseñado para usarse con el lenguaje de programación C y Fortran. El uso de estos lenguajes para el desarrollo de aplicaciones en ambientes con diversas arquitecturas implica recompilar e incluso modificar la aplicación en cuestión para cada una de las arquitecturas, es decir, las aplicaciones no son portables.

Por otra parte, el lenguaje Java está ganando aceptación en diversos ámbitos, desde comerciales hasta investigación. El lenguaje Java ofrece ciertas características tales como portabilidad, mecanismos de concurrencia, tecnologías para el desarrollo de aplicaciones distribuídas y mecanismos para la ejecución de código nativo a cada arquitectura, entre otras. Esto hace de Java un candidato para el desarrollo de aplicaciones paralelas.

En este trabajo se presenta el diseño y construcción de una Plataforma para el desarrollo y ejecución en paralelo de aplicaciones desarrolladas en Java. Esta Plataforma fue desarrollada usando el mismo lenguaje, y está orientada para su uso en Clusters o redes LAN.

En el capítulo 2 se presentan algunas nociones sobre paralelismo, y la forma en que las tendencias de las aplicaciones y el diseño de computadoras han permitido la evolución del paralelismo. En este capítulo se menciona también el modelo general de la computadora paralela, empleada para hacer el modelado de aplicaciones en paralelo.

En el capítulo 3 se presentan algunas características que posee el lenguaje Java. Estas caracterśticas son importantes para el desarrollo de aplicaciones distribuídas. En este capítulo se presentan las nociones de Modularidad, Concurrencia, Escalabilidad y Paralelismo en Java. Tras definir estas nociones, es más fácil ubicar el desarrollo de la Plataforma. Al final de este capítulo se hace mención de una implementación de MPI para Java, se analiza y se presenta la comparación de esta implementación contra implementaciones para otros lenguajes.

En el capítulo 4 se presenta el análisis que se realizó para el desarrollo de la Plataforma, usando algunos diagramas de UML. Este análisis parte de los Requerimientos de Uso, y emplea los diagramas de Casos de Uso, Actividades, Clases y Secuencia. Con base en el análisis, se presenta la Arquitectura de la Plataforma, que se empleó para el desarrollo de la misma.

En el capítulo 5 se presenta la implementación de la Plataforma, que explica cómo se desarrollaron las funcionalidades que se detectaron en el análisis realizado en el capítulo 3. Se explica como se construyó el Canal de Comunicaciones de la Plataforma, la manera en que se estructura lógicamente la Plataforma y cómo es que proporciona los servicios de la memoria compartida.

En el capítulo 6, se identifican los métodos que puede emplear el desarrollador para acceder a las funcionalidades de la Plataforma, y concluye presentando dos ejemplos.

Finalmente, en el capítulo 7, se presentan las conclusiones del trabajo y las mejoras que se pueden realizar a la Plataforma.

Capítulo 2

Paralelismo

En esta capítulo se presentan algunas nociones sobre el paralelismo y el cómputo paralelo. Mediante la explicación de las tendencias en las aplicaciones que se han desarrollado en diversos ambientes, las tendencias en el diseño de computadoras y redes, se intenta mostrar cómo el paralelismo es una consecuencia natural. También se presentan las primeras nociones sobre los modelos de máquinas paralelas, que son ampliamente usados como base para modelar aplicaciones.

2.1. Cómputo Paralelo

En el cómputo paralelo los sistemas se extienden con más procesadores para obtener una reducción en el tiempo de ejecución. Otras veces, el problema a resolver puede ser modelado naturalmente en forma paralela. La eficiencia y la efectividad del paralelismo depende por mucho de los problemas a resolver con algoritmos selectos y de las arquitecturas de hardware dedicadas.

Una computadora paralela es un conjunto de procesadores que pueden trabajar cooperativamente para resolver un problema computacional. Esta definición es lo suficientemente amplia para incluir supercomputadoras que tienen cientos o miles de procesadores, redes de estaciones de trabajo, estaciones de trabajo con multiprocesadores y sistemas embebidos. Las computadoras paralelas son interesantes porque ofrecen el potencial para concentrar recursos computacionales, sean procesadores, memoria o ancho de banda de E/S, usados en problemas computacionales importantes.

El paralelismo a veces ha sido visto como un área rara y exótica de la computación, interesante pero de poca relevancia para el programador promedio. Un estudio de las tendencias en las aplicaciones, arquitecturas de computadoras y redes, muestra que este punto de vista no es sostenible. A pesar de que las aplicaciones científicas siguen siendo las principales motivantes detrás del desarrollo de tecnologías de cómputo paralelo, el paralelismo se está volviendo ubicuo, y la programación en paralelo se está convirtiendo en la columna de la programación empresarial[WAL01].

2.1.1. Tendencias en las Aplicaciones

Conforme las computadoras se hacen más rápidas, se puede comenzar a suponer que eventualmente se convertirán en lo 'suficientemente rápidas'y el ánimo por incrementar el poder de cómputo será satisfecho. Sin embargo, la historia sugiere que conforme una tecnología en particular satisface aplicaciones conocidas, surgirán nuevas aplicaciones a partir de esas tecnologías, y en su momento, demandarán el desarrollo de nuevas tecnologías.

Como ilustración de este fenómeno, un reporte preparado por el gobierno británico al final de la decada de los 40, concluía que los requerimientos computacionales de la Gran Bretaña se podían satisfacer con dos o tal vez tres computadoras. En esos días, las computadoras eran usadas principalmente para calcular tablas de balística. Los autores del reporte no consideraron otras aplicaciones en ciencia e ingeniería, dejando sólo las aplicaciones comerciales que podrían dominar la computación. De forma similar, los prospectos iniciales del proyecto de computadoras Cray predijeron un mercado para 10 supercomputadoras; varios cientos de ellas se vendieron [IAN01].

Tradicionalmente, los desarrollos de punta en cómputo se han motivado por simulaciones numéricas de sistemas complejos tal como clima, dispositivos mecánicos, circuitos electrónicos, procesos de manufactura y reacciones químicas. Sin embargo, lo que ha conducido al desarrollo de computadoras más rápidas hoy en día, son las aplicaciones comerciales, que requieren que una computadora maneje grandes volúmenes de informarción en formas sofisticadas. Estas aplicaciones incluyen videoconferencias, ambientes de trabajo colaborativo, diagnóstico médico asistido por computadora, bases de datos en paralelo usadas para la toma de decisiones, gráficos avanzados y realidad virtual para la industria del entretenimiento [IAN01]. Por ejemplo, la integración del cómputo paralelo, el alto desempeño de las redes y las tenologías multimedia están conduciendo al desarrollo de servidores de video, computadoras diseñadas para servir cientos o miles de peticiones simultáneas para video en tiempo real. Cada video puede involucrar transferencia de datos a altas velocidades de varios megabytes por segundo y grandes cantidades de procesamiento para codificación y decodificación. En gráficos, los conjuntos de datos tridimensionales se están aproximando a un volumen de 10^9 elementos (unos 1024 elementos por lado). Realizando 200 operaciones por elemento, una pantalla que se refresca 30 veces cada segundo, requiere una computadora capaz de realizar 6.4×10^{12} operaciones por segundo.

Aunque algunas aplicaciones comerciales podrían definir la arquitectura de muchas computadoras paralelas en un futuro próximo, las aplicaciones científicas tradicionales continuarán teniendo usuarios importantes de la tecnología de cómputo paralelo. Además, conforme los efectos no lineales impongan límites en la comprensión ofrecida por las investigaciones puramente teóricas y conforme la experimentación se vuelva más costosa e impráctica, los estudios computacionales de sistemas complejos se volverán más importantes. Los costos de computación tipicamente se incrementan a la cuarta potencia o más de la 'resolución'que determina la precisión requerida, así que estos estudios tienen una insaciable demanda por mayor poder de cómputo. Estos estudios son frecuentemente caracterizados por sus grandes requerimientos de memoria y de

E/S. Por ejemplo, una simulación del clima de la tierra de 10 años usando un modelo del estado del arte puede involucrar 10^{16} operaciones en punto flotante, unos 10 días a una velocidad de ejecución de 10^{10} operaciones en punto flotante por segundo (10 gigaflops). Esta misma simulación puede generar fácilmente un ciento de gigabytes o más de información. Además, los centíficos pueden imaginar ciertos refinamientos a estos modelos que pueden incrementar estos requerimientos de cómputo unas 10,000 veces.

En resumen, la necesidad por computadoras más rápidas se encuentra gobernada por las demandas de las aplicaciones de información intensiva en usos comerciales y científicos/de ingeniería. De manera creciente, los requerimientos de estos campos se están mezclando, conforme las aplicaciones científicas/de ingeniería manejan mayores cantidades de información, y las aplicaciones comerciales realizan cómputos más sofisticados.

2.1.2. Tendencias en el Diseño de Computadoras

El desempeño de las computadoras ha crecido exponencialmente desde 1945 al presente, en un promedio de un factor de 10 cada 5 años [IAN01]. Mientras las primeras computadoras realizaban unas cuantas decenas de operaciones en punto flotante por segundo, las computadoras paralelas de los mediados de los 90's logran decenas de miles de millones de operaciones por segundo. Tendencias similares se pueden observar en computadoras de menor desempeño, tal como calculadoras, computadoras personales y estaciones de trabajo. Toda esta evidencia sugiere que el crecimiento no se detendrá. Sin embargo, las arquitecturas tenderán a tener un cambio radical, de secuencial a paralelo.

El desempeño de una computadora depende directamente del tiempo que requiere para realizar una operación básica, y el número de operaciones básicas que puede realizar concurrentemente. El tiempo para realizar una operación básica está limitada por los ciclos de reloj del procesador, es decir, el tiempo que requiere para realizar la más primitiva operación. Sin embargo, los tiempos de ciclo de reloj están decrementando lentamente y parecen estar llegando a límites físicos, cercanos a la velocidad de la luz. No se puede depender de los procesadores aún más veloces para proveer un incremento en el desempeño computacional.

Teoría de la Complejidad Computacional y VLSI

La teoría de la Computabilidad se enfoca en estudiar los problemas de decisión para los cuales existe solución(correctamente, estudia para qué problemas de decisión existen algoritmos que los solucionen). La Teoría de la Complejidad Computacional se enfoca en ver qué algoritmos eficientes existen para solucionar problemas de decisión. Eso conlleva a establecer qué *recursos* se deben emplear *eficientemente*. Dos recursos de gran importancia son el *tiempo* y el *espacio*, es decir, el número de pasos que un algoritmo emplea (en el peor de los casos) en regresar resultados y la cantidad de memoria requerida para solucionar el problema [PAU01].

La tecnología VLSI(Very Large Scale Integration) ha cambiado el tamaño y velocidad de las estructuras de cómputo. Una microcomputadora VLSI ocupa menos de un centímetro cuadrado de silicio, pero realiza lo que varios metros cúbicos de componentes de computadoras de hace 20 años realizaban. Las densidades en circuitos que se pueden lograr con la tecnología VLSI están alcanzando niveles sorprendentes, aunque aún dejan espacio para mejoras en un futuro. Integrados con cientos de miles o hasta millones de transistores son una realidad.

El poder computacional de un integrado en ocasiones se mide por el número de transistores que contiene. Esta es una aproximación errónea, ya que la organización de la circuitería del integrado tiene un fuerte efecto en su tamaño y velocidad. En general, los diseños regulares de integrados hacen un uso más eficiente del área de silicio. Tales diseños usan un área menor para el cableado entre los transistores, dejando mayor espacio para los transistores mismos. Esto explica porque la tecnología de hoy en día puede poner millones de transistores en un integrado de memoria, pero sólo unos cuantos cientos de miles en un integrado de "lógica aleatoria". Todo esto explica también, porque el tamaño de un circuito es más naturalmente medido por el área que ocupa más que por los transistores que contiene[THO01].

Las metodologías establecidas para estudiar la complejidad computacional pueden aplicarse a los nuevos problemas expuestos por los circuitos VLSI. El área A y el tiempo T que le toma a un integrado VLSI en calcular una función de N entradas se encuentran relacionados. Una relación general, por ejemplo, para una función que calcula la transformada de Fourier de N puntos satisface la relación $AT^2 = c^2 log^2 N$ para cierto valor de c. Existen límites inferiores y superiores para las relaciones que se pueden encontrar, según la función que realice el integrado. Cuán alejados o cercanos se encuentren tales límites(superior e inferior) entre sí se puede demostrar por la existencia de circuitos cercanos al ópitmo(según la función y el algoritmo empleado para el diseño del circuito). Pero existe cierta característica que se puede denotar: los circuitos basados en patrones de interconexiones de intercambio aleatorio suelen ser rápidos pero muy grandes; los circuitos basados en patrones de interconexiones en red suelen ser lentos, pero muy pequeños[THO01].

VLSI y Concurrencia Interna

A pesar de un óptimo diseño, existen ciertas limitaciones físicas que impiden que un integrado tenga velocidades de respuesta aún más rápidas. Para lidiar con estas limitaciones, el diseñador debe intentar utilizar la concurrencia interna en un integrado, por ejemplo, operando simultáneamente en los 64 bits de dos números que han de ser multiplicados. Sin embargo, un resultado fundamental en la teoría de Complejidad VLSI afirma que esta estrategia es cara. Los resultados demuestran que para ciertos cómputos transitivos (en el cual un resultado puede depender de alguna salida que sea usada en cualquier entrada), el área A y el tiempo T requerido para realizar este cómputo, se relacionan tal que se requiere que la relación AT^2 no exceda alguna otra función dependiente del tamaño del problema[IAN01]. Este resultado se puede explicar informalmente asumiendo que cierto cómputo debe mover una cierta cantidad de

información de un lado del integrado al otro. La cantidad de información que se puede mover en una unidad de tiempo está limitada por la sección transversal del integrado $\sqrt{A}$. Esto da una tasa de transeferencia de $\sqrt{A}T$, de la cual la razón AT^2 se obtiene. Para decrementar el tiempo requerido para mover la información en cierto factor, la sección transversal debe incrementarse por el mismo factor, y entonces el área total se debe incrementar por el cuadrado de ese factor.

El resultado AT^2 significa que no sólo es difícil construir componentes individuales que operen a más altas velocidades, sino que puede resultar indeseable hacerlo. Resulta más barato usar un mayor número de componentes más lentos. Por ejemplo, si se tiene un área de silicio para usar en una computadora, se puede bien construir n^2 componentes, cada uno de tamaño A y que puedan realizar una operación en tiempo T, o construir un único componente que realice la misma operación en un tiempo T/n. El sistema multicomponente es potencialmente n veces más veloz.

Los diseñadores de computadoras pueden usar una variedad de técnicas para superar estas limitaciones, incluyendo "pipelining" (diferentes etapas de varias instrucciones pueden ejecutarse concurrentemente) y múltiples unidades de función (incluir varios multiplicadores, sumadores, etc., controlados por un único flujo de instrucciones). Ultimamente, los diseñadores están incorporando múltiples "computadoras", cada una con su propio procesador, memoria y lógica asociada de interconexión. Esta aproximación se puede lograr por los avances en la tecnología VLSI que continúa reduciendo el número de componentes requeridos para implementar una computadora. Conforme el costo de una computadora sea (aproximadamente) proporcional al número de componentes que contiene, una integración mayor incrementaría el número de procesadores que pueden incluirse en una computadora, manteniéndola en un costo particular.

2.1.3. Tendencias en Redes

Otra tendencia que ha cambiado la presencia del cómputo es el enorme incremento en las capacidades de las redes que conectan a las computadoras. No hace mucho, las redes de altas velocidades corrían a una tasa de 1.5 Mbits por segundo; para finales de los 90's los anchos de banda excedían ya los 1000 Mbits por segundo. Importantes mejoras en confiabilidad de las redes también se han logrado. Estas tendencias hacen posible el desarrollo de aplicaciones que usan recursos físicamente distribuídos, como si ellos fueran parte de la misma computadora. Una aplicación típica de esta clase puede utilizar procesadores en múltiples computadoras remotas, acceder a una selección de bases de datos remotas, realizar gráficos en una o más computadoras y proveer resultados en tiempo real.

El cómputo sobre redes de computadoras ("cómputo distribuído") no es sólo un subcampo del cómputo paralelo. El cómputo distribuído está profundamente ligado con problemas tales como confiabilidad, seguridad y heterogeneidad que generalmente se consideran tangenciales en el cómputo paralelo.

Un sistema distribuído es en el que a la falla de una computadora que forme parte del 'sistema', y que el usuario no sabía siquiera que existiera, puede hacer que su propia computadora (o más propiamente su fracción de aplicación) no opere

correctamente[IAN01]. Aunque la tarea básica de desarrollar programas que puedan correr en varias computadoras es a la vez un problema del cómputo paralelo. En este aspecto, el cómputo paralelo y distribuído están convergiendo.

2.1.4. Paralelismo, Una Consecuencia Natural

Estas tendencias en cuanto a aplicaciones, arquitectura de computadoras y redes, sugiere un futuro en el cual el paralelismo prevalece no sólo en supercomputadoras, sino también en estaciones de trabajo, computadoras personales y redes. En tal futuro, se requerirá que los programas exploten los múltiples procesadores de cada computadora, además de los procesadores que tenga disponible a lo largo de la red. Muchos algoritmos actuales están especialmente diseñados para ejecutarse en un sólo procesador. Esta situación implica una necesidad para que nuevos algoritmos y estructuras de programas puedan realizar más de una operación a la vez. La concurrencia es ahora un requerimiento fundamental para algoritmos y programas.

De forma adicional, aparentemente los procesadores continuarán desarrollándose cada vez más rápidos. De ahí, los sistemas de software contarán con el hecho de que las plataformas en las cuales se ejecutarán serán cada vez más potentes; esto es importante, ya que en este ambiente la escalabilidad se vuelve un aspecto fundamental; es decir un programa que es capaz de usar solamente un número fijo de procesadores o de ejecutarse en una sola computadora no se puede considerar una "inversión en software".

2.1.5. Primera Clasificación de Arquitecturas Paralelas

Un proceso es cualquier flujo de control a través de un conjunto de instrucciones almacenadas en una computadora, ejecutándose en un dispositivo de hardware dedicado. Hablando de la arquitectura de hardware, una computadora paralela es una colección de dos o más procesadores que tienen cierta conexión uno con otro mediante una interfaz de red o mediante memoria.

Existen dos términos que discriminan de forma importante los tipos de máquinas paralelas:

- SIMD: Single Instruction Multiple Data
- MIMD: Multiple Instruction Multiple Data

Si los procesadores operan de forma síncrona, es decir, se bloquean y sincronizan a cada instrucción ejecutada, se dice que la computadora paralela es un sistema SIMD. Los procesadores SIMD ejecutan de forma simultánea exactamente las mismas instrucciones, pero sobre conjuntos diferentes de datos. En un sistema SIMD, todos los elementos hacen las mismas cosas al mismo tiempo, a menos que los procesadores estén inactivos.

Si los procesadores operan de manera independiente, pero con ocasionales pausas para sincronizar sus procesos, se dice que la computadora paralela es un sistema MIMD. Los sistemas MIMD contienen múltiples unidades de secuencia, lo que significa que estos

sistemas pueden operar de forma asíncrona e independiente. Cada procesador corre bajo el control de su propia unidad de secuencia, lo que significa que instrucciones diferentes pueden ejecutarse simultáneamente, una en cada procesador [IAN01].

La distinción más importante entre los sistemas anteriores es el grado de sincronización entre los procesadores: las arquitecturas SIMD se encuentran estrechamente sincronizadas. Los procesadores en las computadoras SIMD casi siempre se encuentran interconectados por alguna clase de interfaz de red, que les permite pasarse mensajes entre si. En tales sistemas la memoria se encuentra asociada con cada uno de los procesadores, en vez de con un grupo de procesadores, de ahí que no exista memoria central. La información de la aplicación debe ser copiada y enviada a donde se procesará. Por ello, las máquinas SIMD también son conocidas como computadoras de memoria distribuída.

La memoria distribuída en las máquinas SIMD se encuentra ligada por una red de interconexión en forma de hipercubo. La tendencia ha sido crear interconexiones más y más complejas entre las memorias y los procesadores. Si la interconexión crea un camino a través de todos los procesadores, se dice que es una red 1-D; si la interconexión puede ser dibujada sobre papel sin tocar las aristas de otras conexiones, se dice que es una red 2-D; las redes 3-D tienen una configuración tridimensional, y así sucesivamente.

Las máquinas MIMD pueden clasificarse en:

- Memoria compartida.
- Memoria distribuída.

En una máquina MIMD de memoria compartida, todos los procesadores se encuentran interconectados con la misma memoria central, así que ellos comparten el acceso a la información en vez de duplicarla. Un sistema de memoria compartida es tal que las partes paralelas de una aplicación se sincronizan estableciendo y limpiando cerraduras sobre la información almacenada en un espacio de memoria centralizado y compartido, o a través de la sincronización de procesos, o a través de *barridas* a nivel de software. Una cerradura previene el acceso a la información a menos que ciertas condiciones se cumplan, tal como 'sólo un proceso tiene acceso'. Una *barrida* forza a los procesos a esperar hasta que todos los procesos han llegado al mismo punto en el programa paralelo.

En una máquina MIMD de memoria distribuída, los procesadores y el almacenaje de información se replican. Es aparentemente inmediato que cada uno de los nodos de procesamiento que se han replicado (o más concretamente, los procesadores) sean semejantes al primer nodo. Cada uno es capaz de almacenar y ejecutar su propio programa, con su propio conjunto de datos, completamente independiente de, y asíncronamente con, el resto de los nodos de procesamiento. El término multicomputadora a veces es usado para este tipo de arquitectura, y el nombre es correcto. Una máquina MIMD con memoria distribuída usa el paso de mensajes para sincronizar las partes paralelas de una aplicación.

En ocasiones, si los nodos que conforman esta máquina MIMD de memoria distribuída, no son sino nodos de propósito general interconectados por alguna interfaz de

red de uso general, tal como una red de estaciones de trabajo, se le conoce como SPMD (Same Program, Multiple Data)[IAN01].

Recientemente, la programación multihilos sobre arquitecturas SMP(Symmetrical MultiProcessor) y la programación con base en el Paso de Mensajes en sistemas de memoria distribuída (o incluso en sistemas tipo cluster) se vuelven más y más populares[IAN01].

2.2. Programación en Paralelo

Para poder crear soluciones que empleen paralelismo, se deben conocer los modelos generales que representan las arquitecturas sobre las cuales se ejecutarán tales aplicaciones. Cada arquitectura puede poseer características propias de las cuales se puede sacar provecho para poder modelar específicamente una aplicación, o bien para estimar el comportamiento de la aplicación. En esta sección se presenta el Modelo General de la Computadora Paralela, junto con otros modelos conocidos. Al final, se presenta una introducción al Modelo de Programación en Paralelo.

2.2.1. El Modelo General de la Computadora Paralela

El modelo general de la computadora paralela es simple y realista. Es simple porque ayuda a facilitar su entendimiento y programación; es realista porque asegura que los programas desarrollados para este modelo se ejecutarán con eficiencia razonable en computadoras reales.

Para poder explicar claramente el modelo de la *Multicomputadora*, se parte del modelo de Von Neumann. Este modelo se refiere a las arquitecturas de computadoras que utilizan el mismo dispositivo de almacenamiento tanto para las instrucciones como para los datos. El modelo de la "Multicomputadora" es un modelo de una máquina paralela. Este modelo (visto en la Figura 2.1), está conformado por un conjunto de nodos que hacen las veces del modelo simple de una computadora de Von Neumann, pero se encuentran interconectados por una red. Cada computadora ejecuta su propio programa. Este programa hace lecturas y escrituras sobre su memoria local y puede enviar y recibir mensajes a través de la red. Los mensajes se usan para comunicar los nodos, o de una manera equivalente, para leer las memorias remotas. En una red idealizada de interconexión el costo de enviar un mensaje entre dos nodos es independiente de la localización del nodo y del tráfico de la red, pero depende directamente del tamaño del mensaje.

Una característica que define al modelo de la multicomputadora, es que los accesos a la memoria local implican menor costo que el acceso a la memoria remota. Es decir, los costos de lectura y escritura en la memoria local son menores que el envio y recepción de mensajes. De ahí, es preferible que el acceso a la información local sea más frecuente que el acceso a la memoria remota. A esto se le llama "localidad", y es junto con la concurrencia (acceso a recursos por diferentes procesos al mismo tiempo) y la escalabilidad (la capacidad de un sistema para incrementar su desempeño global tras el incremento de recursos)un requerimiento para el software paralelo.

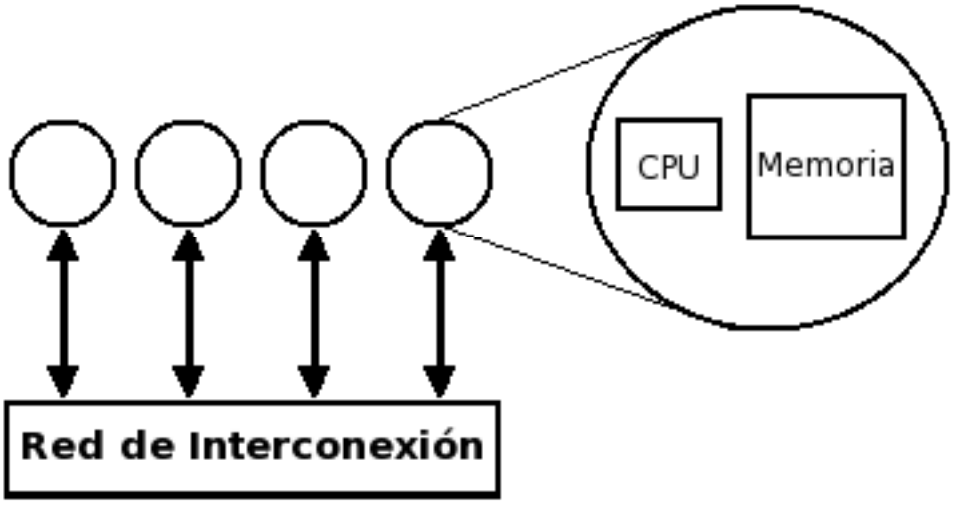


Figura 2.1: El modelo de la multicomputadora consiste en un grupo de nodos de computadoras, cada una siendo una máquina de Von Neumman. Cada nodo puede leer su memoria local y hace uso del paso de mensajes para leer la memoria de otros nodos.

2.2.2. Otros Modelos de Máquinas Paralelas

Existen otros modelos de máquinas paralelas que guardan ciertas diferencias con el modelo de la multicomputadora, como se expone en la Figura 2.2.

La multicomputadora es similar a la computadora MIMD de memoria distribuída. La computadora MIMD puede ejecutar un conjunto individual de instrucciones sobre su propio conjunto de datos, ya que la memoria se encuentra distribuída entre los procesadores. La principal diferencia entre la multicomputadora y la computadora MIMD de memoria distribuiída es que en esta última, el costo de enviar un mensaje entre dos nodos es dependiente de la ubicación del nodo y del tráfico en la red.

Otra clase de computadora paralela es la *Multiprocesador*, o computadora MIMD de memoria compartida. En las computadoras multiprocesadores, todos los procesadores comparten el acceso a una memoria común, típicamente por un bus o por una jerarquía de buses. En el modelo ideal de la máquina paralela de acceso aleatorio (o PRAM, por sus siglas en inglés, Parallel Random Access Machine), que se usa en estudios teóricos de algoritmos paralelos, cualquier procesador puede acceder a cualquier elemento de la memoria y le tomará el mismo tiempo que a cualquier otro procesador. En la práctica, esta arquitectura usualmente introduce alguna forma de jerarquía en la memoria. Es posible reducir la frecuencia con la cual la memoria compartida es accedida por un nodo en particular, almacenando copias de datos frecuentemente usados en la memoria caché local asociada a cada procesador. Acceder a esta memoria caché es mucho más veloz que acceder a la memoria compartida. De aquí que la *localidad* sea importante.

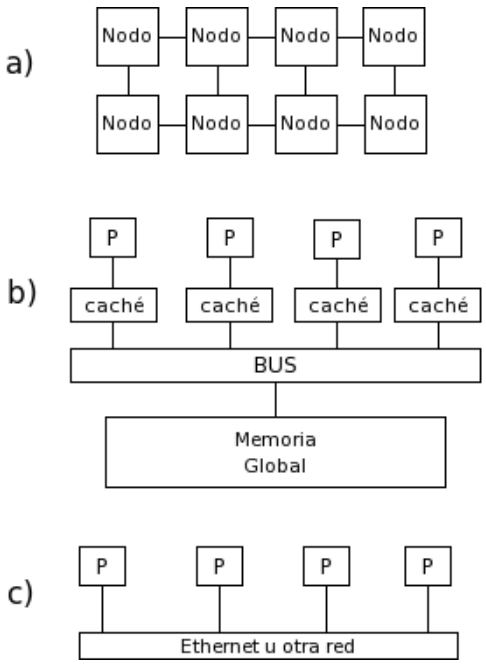


Figura 2.2: (a)Una computadora MIMD conectada en red con memoria distribuída; (b)Una computadora multiprocesador con memoria compartida; (c)una red de área local interconectando nodos. La letra P denota un nodo independiente.

Las diferencias entre multicomputadoras y multiprocesadores pueden ser meramente cuestión de acuerdo. Programas desarrollados en multicomputadoras se pueden ejecutar eficientemente en multiprocesadores, ya que la memoria compartida permite una implementación eficiente a través del paso de mensajes.

Una clase más especializada de la computadora paralela es la computadora SIMD. En las máquinas SIMD todos los procesadores ejecutan la misma instrucción en diferentes porciones de datos. Este enfoque puede reducir la complejidad de hardware y software, pero sólo es apropiado para problemas especializados, que se caracterizan por un alto grado de regulidaridad, por ejemplo, tratamiento de imágenes o simulaciones numéricas. Los algoritmos para una multicomputadora en general no pueden ejecutarse eficientemente en computadoras SIMD.

Otra clase de sistemas de computadoras se pueden usar a veces como computadoras paralelas, y son las computadoras interconectadas por una LAN o WAN. Aunque esta clase de enfoques introduce problemas adicionales acerca de confiabilidad y seguridad. Esta clase de sistemas se puede ver como multicomputadora, pero con el correspondiente alto costo de acceso remoto. Redes con base en Ethernet y ATM son comunes en esta clase de sistemas.

2.2.3. Un Modelo de Programación en Paralelo

La máquina del modelo de von Neumman asume que un procesador puede ejecutar secuencias de instrucciones. Una instrucción puede especificar, además de operaciones aritméticas, la dirección de un dato para ser leído o escrito en la memoria, y la dirección de la siguiente instrucción a ser ejecutada. Mientras es posible programar una computadora en términos de este modelo básico usando lenguaje de máquina, este método es complejo para muchos propósitos, a grados prohibitivos, ya que se debe llevar la pista de millones de localidades de memoria y organizar la ejecución de miles de instrucciones de máquina. De ahí, que sea necesario aplicar técnicas modulares de diseño, tal que programas complejos puedan escribirse a partir de componentes simples, y esos componentes sean estructurados en términos de abstracciones, tales como estructuras de datos, ciclos, procedimientos y objetos. La abstracción y la modularidad son tan importantes como en la programación secuencial. La modularidad es, junto con la concurrencia, la escalabilidad y la localidad un requerimiento para el desarrollo de software en paralelo [IAN01].

Es necesario considerar qué abstracciones son necesarias y útiles en el modelo de programación paralela. *Es claro que se necesitan mecanismos que permitan el uso de la concurrencia y la localidad, y que faciliten el desarrollo de programas escalables y modulares.* También se requiere que esas abstracciones sean simples y que concuerden con el modelo de la multicomputadora. Existen dos abstracciones que son de utilidad para estos fines, la tarea y el canal, las cuales se justifican a continuación:

- Un cómputo paralelo consiste de una o más tareas. Las tareas se ejecutan concurrentemente. El número de tareas puede variar durante la ejecución del programa.
- Una tarea encapsula un programa secuencial y su memoria local. Además un conjunto de entradas y salidas definen la interfaz de su ambiente.
- Una tarea puede realizar cuatro operaciones básicas, además de leer y escribir en su memoria local: enviar mensajes, recibir mensajes, crear nuevas tareas y terminar.
- El envío de información es asíncrono: se completa inmediatamente. La recepción es síncrona: provoca que la tarea se bloquee hasta que un mensaje se reciba.
- Pares de entradas/salidas se pueden conectar por colas de mensajes, llamadas canales. Los canales se pueden crear y eliminar, y las referencias a los canales se pueden incluir en los mensajes, así que la conectividad puede variar dinámicamente.
- Las tareas pueden ser mapeadas a procesadores físicos en varias formas; el mapeo empleado no afecta la semántica de un programa. En particular, múltiples tareas se pueden mapear a un único procesador. De hecho, es posible mapear una tarea única a múltiples procesadores.

Pensando en tareas, se puede obtener un mecanismo para hablar de *localidad*: la información contenida en una la memoria local de una tarea se considera "cerrada"; el resto de la información se considera remota. La abstracción del canal provee mecanismos para indicar que una tarea requiere información de otra tarea, para proceder (esto se llama dependencia de información)[IAN01].

Capítulo 3

Ubicación del problema

En este capítulo se definen los conceptos de Modularidad, Concurrencia, Escalabilidad y Localidad, además de presentar la forma en que el lenguaje Java cumple con cada uno de ellos. Esto servirá para poder ubicar más claramente el desarrollo de la Plataforma.

3.1. Modularidad en Java

Java es un lenguaje por naturaleza orientado a objetos puro. Los paquetes de Java son un mecanismo mediante el cual se organizan las clases de Java en *Espacios de Nombre* (Namespaces). Esta es la forma en que Java ofrece la modularidad. Los paquetes en Java son identificados de manera única mediante nombres completamente calificados. Las clases y las interfaces son los únicos tipos de datos que se exportan en los paquetes.

El `CLASSPATH` de Java existe por una razón muy práctica, ya que ayuda a localizar módulos. En Java, cuando se exporta un paquete, por ejemplo `java.net`, puede surgir la duda de dónde se localiza físicamente el paquete. Técnicamente, el `CLASSPATH` de Java juega un rol importante para encontrar clases y paquetes. Todas las clases que se pueden localizar directamente por el `CLASSPATH` se dice que pertenecen a un paquete llamado `default`. Es por esta razón que se dice que el `CLASSPATH` de Java sirve para localizar paquetes.

El `CLASSPATH` de Java puede interpretarse como una combinación de espacios de nombres. Cada entrada del `CLASSPATH` define un espacio de nombres con múltiples módulos definidos, y todo el `CLASSPATH` combina todos los nombres de espacio, con todos los módulos incluídos. Si dos módulos tienen el mismo nombre, ambos se mezclan en un módulo único, y las características de ambos se mezclan.

Es erróneo creer que el `CLASSPATH` es usado solamente para la búsqueda estática de clases, y que es usado solamente para la compilación estática. Esto no es cierto, ya que las clases se cargan tardíamente. En tiempo de ejecución, Java necesita una manera de determinar la ubicación de las clases. Java mantiene, de hecho, dos `CLASSPATH`, uno usado para la compilación estática y el otro para la búsqueda en tiempo de ejecución de la máquina virtual. Los dos `CLASSPATH` no tienen que ser necesariamente los mismos.

Los módulos en Java no están sujetos a una jerarquía [DAV01]. Esto puede sonar extraño. Se debe tomar en cuenta la vista lógica del arreglo de los paquetes, no la física. Se puede apreciar que la implementación física de los paquetes de Java, es decir, la que recae sobre el sistema de archivos, sigue una estructura jerárquica. Ya que los directorios pueden contener directorios, parece natural asumir que los paquetes también pueden contener paquetes. Pero esto no es cierto.

3.2. Concurrencia y Java

En esta sección se explica qué es la concurrencia. Una vez definida la concurrencia, se explica la forma en que Java soporta la concurrencia, haciendo mención del paquete `java.util.concurrent`, que es usado ampliamente en el desarrollo de la Plataforma.

3.2.1. Concurrencia

La concurrencia es una propiedad de los sistemas, la cual consiste en cómputos que se realizan a un mismo tiempo, en el cual se permite el compartimiento de recursos comunes entre esos cómputos. Edsger Dijkstra usó la siguiente definición: "la concurrencia ocurre cuando dos o más flujos de ejecución se ejecutan simultáneamente"[FIL01].

El cómputo concurrente es la ejecución múltiple de tareas interactivas. Estas tareas pueden estar implementadas como programas separados, o un conjunto de procesos o hilos creados por un mismo programa. Estas tareas pueden estarse ejecutando en un sólo procesador, en varios procesadores o distribuídos en una red. El cómputo concurrente se centra en la interacción de dichas tareas. Una secuencia correcta en las interacciones y/o comunicaciones entre las diferentes tareas, y la coordinación del acceso a los recursos que comparten son puntos claves en el diseño de sistemas concurrentes.

El uso concurrente de recursos compartidos es una fuente de muchas dificultades. Condiciones de carrera que involucran recursos compartidos pueden terminar en comportamientos impredecibles del sistema. La introducción de Exclusión mutua (mutex) puede prevenir condiciones de carrera, pero también puede provocar problemas tales como *deadlocks*(esperas infinitas) y *starvation*(consumo de recursos sin liberarlos). El diseño de sistemas concurrentes tiene que ver con encontrar técnicas para coordinar la ejecución, intercambiar información, reserva de memoria y la calendarización de ejecución para minimizar el tiempo de respuesta y maximizar el desempeño.

La teoría de la concurrencia ha sido un campo activo de investigación en las ciencias de la computación desde el trabajo presentado por Carl Adam sobre las redes de Petri, en 1960 [FIL01]. Desde entonces, una amplia variedad de modelos teóricos, lógicas y herramientas para entender los sistemas concurrentres se han desarrollado.

Una variedad de métodos formales para modelar y entender los sistemas concurrentes se han desarrollado, incluyendo [FIL01]:

- La máquina paralela de acceso aleatorio (PRAM)
- El modelo de Actor

- Las redes de Petri
- Cálculo de Procesos, que incluye
  - Cálculo de ambiente
  - Cálculo de sistemas comunicados
  - Procesos con comunicación secuencial
  - Cálculo π.

Algunos de estos modelos de concurrencia están enfocados a soportar el estudio y la especificación, mientras que otros se pueden usar en todo el ciclo de desarrollo, incluyendo el diseño, implementación, pruebas y simulación de sistemas concurrentes.

Varios tipos de lógica temporal se pueden usar para ayudar al estudio de sistemas concurrentes, como por ejemplo la Lógica Lineal Temporal y la Lógica de Árboles, ayudan a formar premisas sobre las secuencias de los estados en que un sistema concurrente se puede encontrar.

La programación concurrente trata de los lenguajes y los algoritmos a usarse para implementar sistemas concurrentes. La programación concurrente es usualmente considerada más general que la programación paralela, ya que involucra patrones de comunicación y de interacción arbitrarios y dinámicos, cuando los sistemas paralelos generalmente tienen un patrón de comunicaciones predefinido y bien estructurado. El objetivo fundamental de la programación concurrente incluye la correctitud, el desempeño y la robustez. Los sistemas concurrentes generalmente se diseñan para operar indefinidamente y no terminar inesperadamente. Algunos sistemas concurrentes implementan una forma de concurrencia transparente, en la cual las entidades concurrentes pueden competir por un recurso.

Ya que los sistemas concurrentes usan recursos compartidos, en general requieren la inclusión de alguna clase de árbitro en alguna parte de la implementación (que a veces está desarrollado en hardware), para controlar el acceso a los recursos. El uso de árbitros introduce la posibilidad de decisiones no determinísticas, que puede tener consecuencias para la correctitud y desempeño de un sistema concurrente.

3.2.2. La Concurrencia en Java

La plataforma de Java, desde su versión 1.5, incluye un nuevo paquete de utilerías de concurrencia. Estas utilerías están conformadas por clases que están diseñadas para ser usadas como bloques de construcción para clases concurrentes o aplicaciones. Las utilerás de Concurrencia simplifican el desarrollo de clases concurrentes proveyendo implementaciones de bloques de construcción comunmente usados en diseños concurrentes. Las utilerás de Concurrencia incluyen un *pool* de hilos flexible y de alto desempeño; un marco de trabajo para ejecución asíncrona de tareas; una colección de clases optimizadas para acceso concurrente; utilerías de sincronización tales como semáforos, variables atómicas, cerraduras y variables de condición.

Usar las utilerías de Concurrencia, en vez de desarrollar los componentes impone una serie de ventajas. Las implementaciones en las utilerías fueron desarrolladas y revisadas por expertos en concurrencia y desempeño, lo que asegura que las implementaciones son rápidas y escalables.

Las utilerías de Concurrencia incluyen:

- Marco de trabajo para calendarización de tareas. El marco del trabajo del `Executor` es usado para la estandarización en la invocación, calendarización, ejecución y el control asíncrono de tareas, de acuerdo a un conjunto de políticas de ejecución. Las implementaciones de estas políticas permiten que las tareas se puedan ejecutar dentro del mismo hilo invocador, en un hilo único en el fondo, en un hilo nuevo, o en un pool de hilos. Las implementaciones de este marco de trabajo ofrecen políticas configurables, tal como colas de tamaño definido y políticas de saturación, que pueden mejorar la estabilidad de aplicaciones previendo el consumo excesivo de recursos.

- Colecciones concurrentes. Nuevas clases de Colecciones se han agregado, incluyendo las interfaces `Queue` y `BlockingQueue`, además de implementaciones de alto desempeño de las clases `Map`, `List` y `Queue`.

- Variables atómicas. Clases para manejar variables de forma atómica, proveyendo aritmética atómica de alto desempeño, lo que las hace ideales para implementar algoritmos concurrentes de alto desempeño, tal como contadores y generadores de secuencias de números.

- Sincronizadores. Ofrece clases de propósito general para sincronización, incluyendo semáforos, mutex, barriers, latches; que ofrecen una cómoda coordinación entre hilos.

- Cerraduras. A pesar de que las cerraduras están construídas como parte del lenguaje de Java mediante el uso de la palabra reservada `synchronized`, existen algunas limitaciones para construir cerraduras de monitor. Estas utilerías ofrecen un paquete, `java.util.concurrent.locks`, que provee una implementación de cerraduras con la misma semántica de memoria que se usa en la sincronización, pero además soporta especificar un tiempo antes de generar una excepción, en la espera por obtener la cerradura. Soporta también múltiples variables de condición por cerradura, y soporta interrumpir hilos que están a la espera de obtener una cerradura.

- Tiempo en resolución de nanosegundos. Permite medir el tiempo con resolución al orden de nanosegundos, ya que varios métodos de otras clases emplean esta resolución para marcar fallo por tiempo de espera.

3.3. Escalabilidad en Java

La escalabilidad es la propiedad deseable de un sistema de mostrar mayor desempeño si sufre de un aumento de recursos. En esta sección se menciona un método básico para estimar la escalabilidad que es lo suficientemente general para ser aplicable a cualquier sistema. También se mencionan los factores que afectan a la escalabilidad de la Máquina Virtual de Java. Al final de la sección se enfatiza el hecho de que un buen diseño orientado a objetos permitirá una escalabilidad natural de las aplicaciones.

3.3.1. Ley de Amdahl

La ley de Amdahl se usa para encontrar la mejora máxima esperada en un sistema cuando sólo una parte de ese sistema se optimiza. Es usada frecuentemente en cómputo paralelo para predecir la mejora máxima teórica en desempeño, usando múltiples procesadores. En adelante, cuando se hable de mejora en velocidad, se refiere a la reducción en el tiempo que le toma a una porción de código en ejecutarse.

La forma general de la Ley de Amdahl es:

$$\frac{1}{\sum_{k=0}^{n}(\frac{P_k}{S_k})} \tag{3.1}$$

donde

- P_k es un porcentaje de las instrucciones que pueden ser mejoradas (o deterioradas).
- S_k es el factor de velocidad (donde 1 no mejora ni deteriora el desempeño).
- k representa una etiqueta para cada porcentaje diferente y factor de velocidad.
- n es el número de las diferentes mejoras/deterioros obtenidos con los cambios en el sistema.

En su forma más simple, la ley de Amdahl es un algoritmo que indica que la mejora en velocidad no es necesariamente directamente proporcional al número de procesadores. Eventualmente se llega a un punto en que no se puede paralelizar más el algoritmo[AMD01].

Más técnicamente, la ley se centra en la mejora en velocidad que se puede lograr cuando una proporción P de cierto cómputo se puede mejorar (en cuanto a velocidad) por un factor S. Por ejemplo, si una mejora puede acelerar un 30 % del total del cómputo, P será 0.3; si la mejora, logra que la porción afectada sea el doble de veloz, S tendrá un valor de 2. La ley de Amdahl establece que la mejora total en velocidad aplicando la mejora será

$$\frac{1}{(1-P)+\frac{P}{S}} \tag{3.2}$$

Para ver cómo se derivó esta fórmula, se asume que el tiempo de ejecución anterior del cómputo es 1, para alguna unidad de tiempo. El tiempo de ejecución del nuevo cómputo será la fracción de tiempo que toma la parte del cómputo que no se mejoró (el cual es $1 - P$), más la fracción de tiempo que toma la parte del cómputo que se mejoró. La fracción del tiempo de la parte del cómputo que se mejoró es igual al tiempo que le tomaba ejecutarse (antes de la mejora) entre el factor de velocidad, que es igual a $\frac{P}{S}$. La velocidad final se calcula dividiendo el tiempo anterior que le tomaba al cálculo completo entre el nuevo tiempo.

En el caso especial de la paralelización, la ley de Amdahl establece que si F es la fracción de un cómputo que es secuencial (es decir, no se beneficia de la paralelización), y $1 - F$ es la fracción que se puede paralelizar, entonces la máxima velocidad que se puede alcanzar usando N procesadores es

$$\frac{1}{F + \frac{1-F}{N}} \tag{3.3}$$

En el límite, conforme N tienda a infinito, la máxima velocidad tiende a ser $\frac{1}{F}$. En la práctica, la razón precio/desempeño caerá rápidamente conforme N se incremente si se logra que $\frac{1-F}{N}$ no sea tan pequeña en comparación con F, es decir, se paralelice lo más posible el cómputo.

3.3.2. Escalabilidad en el Software

Tanto en telecomunicaciones como en software, la escalabilidad es una propiedad deseable de un sistema, una red o un proceso, el cual indica su habilidad para manejar el incremento en la carga de trabajo exitósamente, o que esté dispuesto a agrandarse [BON01]. Por ejemplo, se puede referir a la capacidad de un sistema para incrementar su desempeño total bajo cierto incremento de recursos, típicamente de hardware.

La escalabilidad es en general, un término difícil de definir, ya que en cualquier caso particular es necesario definir los requerimientos específicos para la escalabilidad en las dimensiones que sean importantes para el caso. El sistema al que se le agrega hardware y logra una mejora proporcional a la cantidad de hardware agragegada, se dice que es escalable [BON01].

Se dice que ocurre una escalación vertical cuando se agregan recursos a un nodo en específico, como agregar memoria o un disco de mayor capacidad, pero a un sólo nodo.

Se dice que ocurre una escalación horizontal cuando se agregan más nodos al sistema, tal como agregar un nodo extra a un cluster.

Hablando de software, la escalabilidad describe la forma en que una aplicación se comporta conforme su carga de trabajo y sus recursos de cómputo disponibles se incrementan. Un programa escalable puede manejar una carga de trabajo mayor si cuenta con más procesadores, memoria o ancho de banda de E/S. Bloquear un recurso compartido por acceso exclusivo es un cuello de botella de la escalabilidad[GOE01], evita que otros hilos tengan acceso a ese recurso, aún si hay procesadores inactivos para atender a esos hilos. Para lograr la escalabilidad, se debe eliminar o reducir la dependencia en cerraduras sobre los recursos.

3.3.3. Escalabilidad en la Máquina Virtual de Java

La Máquina virtual de Java toma los recursos que le ofrece el hardware host, parte de esos recursos los emplea en el funcionamiento de la misma Máquina Virtual y el resto los pone a disposición de las aplicaciones que ejecuta. Disponer de una mayor cantidad de recursos en el hardware host, es decir, una escalación vertical, ofrece una mayor cantidad de recursos para las aplicaciones que ejecuta la Máquina Virtual y a ella misma. Sin embargo, algunos elementos de la Máquina Virtual de Java se pueden mejorar para aprovechar aún más esos recursos. Uno de esos elementos, que puede acelerar los tiempos de ejecución, es el Garbage Collector.

3.3.4. Escalabilidad y el Garbage Collector

El lenguaje de programación Java es inherentemente orientado a objetos [NAG01], e incluye una característica automática, llamada "Garbage Collection". Esta característica es el proceso de reclamar memoria que fue usada por objetos que ahora ya no poseen ninguna referencia (es decir, ya no son *referidos*). Los objetos no referidos son los que la aplicación ya no puede usar por que todas las referencias a ellos se encuentran fuera de ámbito.

Lenguajes como C y C++ no poseen un *Garbage Collector*. Los desarrolladores deben reservar y liberar la memoria de forma manual, usando las funciones `malloc()` y `free()` para C, y los operadores `new` y `delete` para C++. En las aplicaciones Java, la memoria es reservada por el operador `new`, pero los desarrolladores no necesitan liberar esta memoria explícitamente. En vez de ello, el Garbage Collector determina que objetos aún tienen referencias válidas (objetos vivos) y cuales ya no tienen referencias (objetos muertos), y automáticamente libera la memoria reservada para los objetos muertos.

El Garbage Collector de Java se ejecuta en un hilo separado. Conforme la aplicación reserva memoria para más y más objetos, la memoria del sistema comienza a agotarse; al llegar a cierto umbral, el proceso del Garbage Collector inicia. *El Garbage Collector detiene todos los hilos*, marca cada objeto como vivo o muerto y libera la memoria ocupada por los objetos muertos.

Hay muchos tipos diferentes de Garbage Collectors. Cada uno se apoya en un algoritmo diferente, es decir, exhibe un comportamiento diferente; estos varían de colectores simples de conteo de referencias, hasta colectores generacionales muy avanzados. Ambos, el algoritmo y la implementación pueden afectar el comportamiento del Garbage Collector. Estas son algunas implementaciones de colectores, y no son mutuamente exclusivas:

- Stop-the-world. Detiene todos los hilos de las aplicaciones mientras trabaja.
- Concurrente. Permite que los hilos de las aplicaciones continuen su ejecución.
- Paralelo. Múltiples hilos trabajan en la tarea de recolección.

Desde la versión 1.4 de la plataforma de Java, se incluyeron nuevos colectores, tal como el *Colector Paralelo de Generaciones Jóvenes* y el *Colector Concurrente de*

Generaciones Viejas. De hecho la plataforma viene con dos colectores que son paralelos, y uno de ellos trabaja en conjunción con el Colector Concurrente de Generaciones Viejas y es usado para aplicaciones en tiempo real. Los colectores en paralelo hacen uso de múltiples hilos para paralelizar y escalar sobre arquitecturas SMP, acelerando la recolección.

Los retardos en la recolección pueden disminuirse al orden de milisegundos con la ayuda de los colectores paralelos y concurrentes, mejorando la eficiencia de la aplicación hasta en un 90 % [NAG01].

3.3.5. Escalabilidad y el diseño orientado a objetos

Existe una visión general del cómputo distribuído orientado a objetos, en el cual desde el punto de vista del programador, no hay diferencia esencial entre objetos que comparten el mismo espacio de memoria y los objetos que se encuentran en máquinas separadas. Esta clase de enfoque se ha encontrado en diversos sistemas, tal como CORBA, del *Object Management Group*.

En tales sistemas, un objeto, sea local o remoto, se define en términos de un conjunto de interfaces declaradas en algún lenguaje de definición de interfaces. La implementación del objeto es independiente de la interfaz y oculta a otros objetos. Mientras los mecanismos para invocar un método de ese objeto pueden diferir dependiendo de la ubicación del mismo, esos mecanismos se ocultan al programador, quien escribe exactamente el mismo código para cualquier tipo de llamada, y el sistema se encarga del resto.

Esta visión se puede ver como una extensión del objetivo de las Llamadas a Procedimientos Remotos (RPC), al paradigma orientado a objetos. Los sistemas RPC intentan hacer parecer a la llamada de una función como si fuera local (al menos al programador). Extendiendo esto al paradigma de programación orientada a objetos, evita el *marshalling* y el *unmarshalling* (empaquetado y desempaquetado, respectivamente) de los parámetros y de los resultados, además de la localización y conexión con el objeto destino. Aislando la implementación del objeto a los clientes de ese objeto, el uso de objetos para el cómputo distribuído parece natural.

Implícito en esta visión, es que todo el sistema debe estar conformado por objetos en su totalidad, es decir, todas las invocaciones o llamadas a servicios del sistema eventualmente se convertirán en llamadas a objetos que pueden residir en otras máquinas[WAL01].

En la práctica, la invocación al método de un objeto local y la invocación a un objeto en otra computadora no son lo mismo. La visión es que los desarrolladores escriban sus aplicaciones de tal forma, que los objetos en una aplicación se unan usando la misma lógica que los objetos de diferentes aplicaciones que se comunican. Lo que se requiere en realidad, es un conjunto de técnicas que vayan desde las implementaciones en el mismo espacio de direcciones, tal como OLE de Microsoft, a típicas llamadas RPC[WAL01].

Una parte central de la visión es que si una aplicación se construye usando solamente objetos y haciendo uso correcto del paradigma orientado a objetos, los puntos de error

emergerán naturalmente, además que el cometer errores en el diseño, implicará una fácil corrección.

Una justificación conceptual para esta visión es que independientemente de si la llamada es local o remota, no tendrá impacto en la correctitud del programa. Si un objeto soporta una interfaz en particular, y el soporte para esa interfaz es semánticamente correcta, entonces no hay diferencia en la correctitud del programa, si la operación se realiza en la misma máquina, o en otra máquina diferente o en hardware dedicado fuera de línea. Es más, ver la localización como parte de la implementación de un objeto y por consecuencia como parte de su estado que oculta del mundo, parece una extensión natural del paradigma orientado a objetos.

Tal sistema disfrutaría de muchas ventajas. Permitiría que la tarea de mantenimiento del software se transforme fundamentalmente. La granularidad de los cambios, y por tanto de las actualizaciones, se puede cambiar del nivel de sistema completo a nivel de componente. Mientras las interfaces entre objetos permanezcan constantes, las implementaciones de esos objetos pueden modificarse a voluntad. Los servicios remotos se pueden mover a un espacio local de direcciones, y los objetos que comparten un área de direcciones se pueden separar a máquinas diferentes. Un objeto se puede 'reparar' y puede ser instalado sin preocuparse de que los cambios alterarán el funcionamiento del resto de los objetos del sistema.

3.4. Localidad en Java

La creciente disparidad en cuanto a velocidades entre el procesador y la memoria, está motivando una fuerte necesidad de optimizar la memoria de programas. Numerosos investigadores han dedicado esfuerzos a estudiar la localidad de la información, su importancia y su efecto en el desempeño, y han investigado la interacción entre el programa y varias arquitecturas de procesadores y de memorias. A pesar de que el trabajo en optimizar las aplicaciones que usan arreglos han tenido mejoras, las aplicaciones que hacen uso intensivo de apuntadores y manejo de memoria dinámica, todavía representan un reto serio. A pesar de que mucho trabajo se ha realizado en esta última área, los investigadores indican que aún hay mucho por hacer para llegar a niveles deseables de desempeño [YEF01].

La creciente popularidad de lenguajes como Java en una amplia variedad de plataformas, que van desde sistemas embebidos hasta servidores, presenta un particular reto desde dos puntos de vista:

- Los programas de Java tienden a hacer uso intensivo de la reserva de memoria, lo que presiona en el subsistema de memoria.
- La naturaleza dinámica de Java, debido a características como la carga dinámica de clases, hace impráctico aplicar optimizaciones, haciendo uso de análisis intensivo del programa y las clases que usa.

Existen algunas técnicas que han mostrado ser eficientes, que recaen sobre la visión macroscópica del sistema, en vez de la particularidad de las aplicaciones[YEF01]. Una

de estas técnicas se basa en la identificación de las clases que más frecuentemente se instancian, que se conocen como *tipos prolíficos*, de un programa dado, y tratan de colocar todas las instancias de esos tipos prolíficos juntas, al momento de la reserva de memoria. Otras técnicas, trabajan al momento que se activa el Garbage Collector, haciendo más efectivo el trabajo de éste.

3.5. Paralelismo en Java

El lenguaje Java está ampliamente considerado como inadecuado para tareas de cómputo intensivo. La razón obvia de esto recae en el pobre desempeño de los programas de Java, que corren más lento comparado contra sus contrapartes de C y Fortran [PAS01]; pero a pesar de que Java no es tan eficiente como Fortran optimizado o C, la velocidad de Java es mejor de lo que su reputación sugiere[VLA01]. Existen varias razones por las cuales Java es el lenguaje ideal para algunas de las aplicaciones a desarrollar.

Java es un lenguaje fácil de aprender, seguro y escalable para resolver problemas complejos. Su popularidad y amplia aceptación han llamado la atención de la comunidad científica e ingenieril, y ha permitido el desarrollo de librerías y herramientas adaptadas al cómputo paralelo de alto desempeño.

Lo más importante, es que la independencia de Java a la arquitectura o al sistema operativo, permiten la distribución de los programas en plataformas heterogéneas. Esto tiene el potencial de llevar las aplicaciones paralelas o distribuídas más allá de clusters especializados de máquinas homogéneas, usadas tradicionalmente en cómputo de alto desempeño[VLA01].

Finalmente, compiladores JIT (Just In Time) que traducen bytecode de Java a código nativo de la plataforma, han tenido avances significativos para mejorar el desempeño, y algunos de estos compiladores han logrado alcanzar un $\frac{2}{3}$ de la velocidad de código en C. El proyecto *Ninja* de IBM ha demostrado que cuando se compila código específicamente para arquitecturas paralelas, se puede alcanzar entre un 80 % y un 100 % del desempeño logrado por código en Fortran optimizado. Combinado con técnicas cuasi estáticas, el código de Java puede ser tan veloz, como el de C o Fortran[VLA01].

Por otra parte, existe un paralelismo implícito en la arquitectura de la Máquina Virtual de Java. Esto se da en la implementación de la Máquina Virtual, la cual hace uso intensivo y natural de los hilos. Si la plataforma de hardware posee las capacidades (como tener instalado más de un procesador, o capacidades de administración multihilos), al igual que el sistema operativo (usando núcleos optimizados para ejecutar tareas en paralelo), la Máquina Virtual podrá hacer uso de estos recursos, logrando cierto paralelismo dentro de la misma computadora.

3.6. Java y MPI

En esta sección se da una breve introducción a MPI. Después se mencionan los esfuerzos que se están realizando para crear una implementación de MPI para Java y

las implicaciones que tal implementación conlleva. Finalmente, se muestran las comparaciones en cuanto a desempeño de una de esas implementaciones de MPI para Java contra otras desarrolladas para otros lenguajes y arquitecturas.

3.6.1. MPI

MPI (*Message Passing Interface*) es un conjunto de funciones que conforman una API que permite a los programadores escribir programas paralelos de alto desempeño, que pasan mensajes entre procesos para realizar una tarea en paralelo. Más formalmente, MPI es una especificación técnica para una librería estándar[GRA01]:

- Define la sintaxis y semántica de un modelo extendido de paso de mensajes.
- No es un lenguaje o la especificación de un compilador.
- No es una implementación específica.
- No ofrece especificaciones para implementaciones:
  - Se ofrecen recomendaciones, pero el implementador tiene la libertad de decisión.
  - Diferentes implementaciones pueden realizar las mismas tareas, pero de formas totalmente diferentes.

MPI es la especificación técnica para una librería que soporta el desarrollo de aplicaciones paralelas en un ambiente de memoria distribuída[GRA01] y las implementaciones que se desarrollen con esta especificación deben ofrecer:

- Rutinas para el paso de mensajes cooperativos en varias modalidades:
  - Comunicación punto a punto.
  - Comunicación colectiva.
- Rutinas para sincronización.
- Tipos de datos derivados para patrones de acceso no contiguos.
- Habilidad de crear subconjuntos de procesos.
- Habilidad para crear topologías de procesos.

MPI es el resultado de décadas de investigacin en cómputo paralelo. En abril de 1992 se organiza un grupo que es nombrado como el *MPI Forum*, conformado por un grupo abierto de representantes de la industria del desarrollo de software, de hardware, más representantes de intereses académicos. En su primera reunión, logran crear la primera versión preliminar. Para mayo de 1994, logran liberar la primera versión completa del estándar de MPI.

MPI es ideal para máquinas paralelas de gran desempenño, tal como la IBM SP, SGI Origin, etc., pero tambin funciona en ambientes más pequeños como un grupo de estaciones de trabajo. Ya que los clusters de estaciones de trabajo son una realidad en muchas instituciones, es común usarlos como un recurso paralelo, ejecutando aplicaciones desarrolladas con MPI. El estándar MPI fue diseñado para soportar la portabilidad e independencia de la plataforma. Como resultado, los usuarios disfrutan de la capacidades de desarrollo multiplataforma, al igual que la comunicación heterogénea transparente[LAM01].

3.6.2. Java y MPI

Con el evidente éxito de Java como lenguaje de programación y su inevitable conexión con el cómputo paralelo y distribuído, la ausencia de una librería bien diseñada y específica para Java para el paso de mensajes, conllevaría a aplicaciones divergentes, no portables. El "Message-Passing Working Group" del foro Java Grande se formó en otoño de 1998 como respuesta a la aparición de varias APIs para el paso de mensajes. Algunas de esas implementaciones se encuentran disponibles desde 1997 trabajando sobre plataformas Linux, Solaris, Windows NT, Irix, AIX, HP-UX y MacOS, al igual que en plataformas paralelas tales como IBM SP-2 y SP-3, Sun E4000, SGI Origin-2000, Fujitsu AP3000, Hitachi SR2201 y otras[VLA01]. Una meta inmediata para el grupo fue la discusión y acuerdo de un API común para librerías MPI, para el paso de mensajes en Java. El API que se definió como parte del trabajo del grupo se nombró *MPJ* (*Message Passing for Java*). Una de las implementaciones más completas de esta API se llama *mpiJava*[BAK01].

El estándar de MPI es explícitamente orientado a objetos. Las librerías de C y Fortran hacen uso de *objetos opacos* que pueden ser manipulados por manejadores de objetos que se obtienen con los constructores de tales objetos, y pasando esos manejadores a las funciones indicadas. La librería de C++, según se especifíca en el estandar de MPI-2, agrupa esas clases en una jerarquía y define muchas de las funciones de la librería como funciones miembro de las clases. La especificación de MPJ sigue este modelo, tomando la estructura de las clases directamente de C++. El propósito de esta acción es proveer una estandarización inmediata y natural para programas que hagan el uso de paso de mensajes en Java, al mismo tiempo que provee una base para la conversión de código entre C, C++, Fortran y Java.

La funcionalidad de mpiJava descansa sobre envoltorios (funciones que hacen llamado a otras funciones) a las implementaciones nativas de MPI para la plataforma objetivo. Mientras esta parece ser una solución razonable para muchos casos, tiene algunas desventajas:

- Una instalación de dos fases. Primero se debe encontrar instalada una versión de MPI en la computadora antes de poder instalar mpiJava.
- Durante el desarrollo de mpiJava, se encontraron conflictos entre el comportamiento de la Máquina Virtual de Java y la librera de MPI en ejecución .

- La estrategia de crear envoltorios va en contra, en cierto sentido, con la ideología de Java de *escribir una vez, ejecutar donde sea*, ya que depende totalmente de la implementación de MPI propia de la plataforma.

De forma ideal, los dos primeros problemas podrían ser resueltos por los proveedores de las librerías nativas de MPI. Ellos podrían ofrecer una interfaz de Java junto con sus implementaciones de MPI de C y Fortran. Esto tendría la ventaja de que ellos serían los más indicados para resolver las disparidades que existen entre sus implementaciones y el comportamiento de la Máquina Virtual de Java. Al final de cuentas, las implementaciones más veloces del mercado de MPJ, serían las que ellos mismos podrían ofrecer.

Estas son algunas características de mpiJava [BRY01]:

- Los objetos opacos se presentan en Java como objetos Java.
- Generalmente los destructores de los objetos se absorben en los destructores de los objetos Java, los cuales son llamados automáticamente por el Garbage Collector (a través del método *finalize*). Algunas funciones requieren que ciertos eventos sucedan antes de poder destruir los objetos que las contienen. Un ejemplo de estas funciones es *MPI_COMM_FREE*, que es una operación colectiva que requiere que los procesadores involucrados respondan antes de poder eliminar el objeto que invocó la llamada. Para esos casos, se ofrece un método `free()` que se debe invocar explícitamente.
- Todas las clases de mpiJava se encuentran en el paquete `mpi`.
- En C y Fortran, los arreglos multidimensionales y unidimensionales tienen una correspondencia directa con la memoria que ocupan. Java trata los arreglos de manera diferente, ya que en Java no existen los arreglos multidimensionales. En mpiJava, el paso de arreglos como parámetros siempre considera que el arreglo es 1-dimensional.
- En C y Fortran, cualquier desplazamiento dentro de un arreglo se puede manipular como otro arreglo. Esto no sucede en Java, por ello, los métodos incluyen un parámetro para indicar el desplazamiento que se quiere emplear como inicio del arreglo.
- En las implementaciones de C y Fortran, las funciones regresan valores de errores. En mpiJava se emplea el mecanismo de las excepciones para reportar errores.
- En las funciones de mpiJava generalmente se omite el paso del parámetro que indique el tamaño de los arreglos, ya que eso se puede obtener directamente del arreglo con su propiedad `length`.

3.6.3. Comparación Java contra MPI

Uno de los problemas a los cuales se enfrenta Java en su aceptación en la comunidad científica es su desempeño. Existen muchos proyectos cuya finalidad es analizar el comportamiento de Java para aplicaciones de alto desempeño [VLA02].

El propósito del desarrollo de una librería MPI para Java fue para proveer a los programadores con la funcionalidad tradicional de MPI a través de una interfaz, a las librerías nativas de MPI. El mecaniso usado para soportar esta interfaz es JNI (Java Native Interface), que especifíca la transición entre código de Java corriendo en la JVM y código nativo dependiente, en C o en Fortran. Esta interfaz ofrece muchas ventajas inmediatas, como el rápido desarrollo de aplicaciones Java usando el desempeño establecido de MPI. Sin embargo la transición entre código de Java y código nativo introduce ciertos tiempos extras.

Comparación del desempeño de los bindings de MPI en diferentes arquitecturas

En esta sección se presenta la comparación del desempeño entre los *bindings* de Java y C/Fortran en tres diferentes plataformas paralelas: un multiprocesador de memoria distribuída (Sun E4000), un cluster Linux y una computadora de memoria distribuída (IBM SP-2). Estos resultados fueron obtenidos y presentados por Vladimir Getov y Michael Philippsen en [VLA01].

Los test de benchmarking que se realizaron fueron el *Numerical Aerodynamic Simulation parallel Embarrassingly Parallel (EP)* y el *Numerical Aerodynamic Simulation parallel Integer Sort (IS)*. La rutina IS evalúa operaciones con enteros y comunicaciones bidireccionales, donde los enteros a ordenar se intercambian entre los nodos. El test EP evalúa el desempeño de operaciones en punto flotante pero requiere de pocas comunicaciones.

Los test se realizaron con las mismas implementaciones de librerías MPI, pero primero se realizaron con código en C y Fortran, y luego se empleó Java y la interfaz de JNI (mpiJava) a las mismas librerías y usando código en Java.

Los test *NAS parallel* establecen diferentes tamaños de problemas llamados "clases" para asegurar resultados comparativos entre diferentes plataformas y ambientes. En estos resultados, para el test *EP* se aplicaron test de clase B(2^{30} puntos de datos), mientras para el test *IS* se aplicaron test de clase A(2^{23} puntos de datos).

Los resultados de la evaluación del test *EP* se presentan en la Figura 3.1. Las estadísticas en el tiempo de ejecución no muestran diferencias significativas más allá del evidente desempeño relativo. El código en Fortran sobre la plataforma IBM SP-2 fue el de mejor desempeño. El código sobre Linux fue el de menor desempeño. En todos los casos, el código demuestra una buena escalabilidad dentro del rango que permitieron las configuraciones de hardware, pero el programa corre aproximadamente 2.5 veces más lento en Java, que en comparación con su contraparte de Fortran.

Un compilador Java de código nativo se puede emplear en lugar de la JVM para resolver el problema anterior. Afortunadamente, se está realizando un gran trabajo

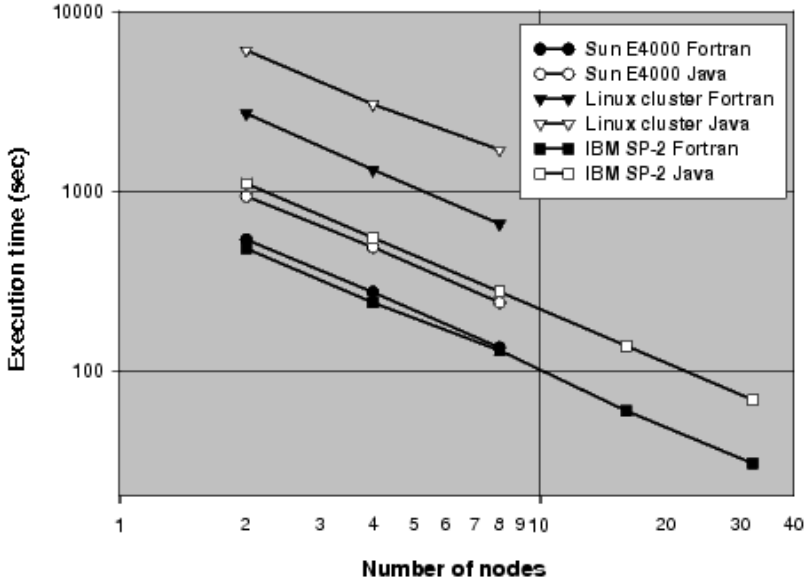


Figura 3.1: Se muestran los tiempos de ejecución para el test de benchmarking para el test *NAS parallel EP*.

en esta área, creando compiladores de Java optimizados, tal como el compilador de IBM de alto desempeño para Java, HPJC (*High Performance Java Compiler*), el que genera código nativo para la arquitectura RS6000. Tales compiladores funcionan como un tradicional compilador de C o de Fortran, donde la compilación estática sucede sólo una vez, antes de la ejecución. Con esto, los tiempos extras se pueden reducir al generar código nativo.

Para el test *IS*, se empleó el compilador HPJC, que toma como entrada el bytecode, aunque también acepta el código fuente. Los resultados evidenciados en la Figura 3.2 se refieren al desempeño del código en una IBM PS-2, pero el código en Java fue traducido a código nativo.

El test *IS* es un test, a comparación del test *EP*, más fuerte para un ambiente de paso de mensajes, involucrando comunicaciones bidireccionales. Los resultados muestran que usando el compilador estático HPCJ, la comunicación usando MPJ es aproximadamente similar a la librería nativa de MPI.

Comparación del desempeño en comunicaciones de los bindings de MPI

Hablando generalmente, el tiempo para las comunicaciones representa una parte substancial de la penalización en desempeño que se paga cuando se trabaja con paso de mensajes en sistemas de memoria distribuída. En muchos casos, los susbsistemas de comunicación emplean un método en el cual el mensaje es divido en paquetes más pequeños. Con esto todos los mensajes son enviados juntos, lo que provoca el efecto

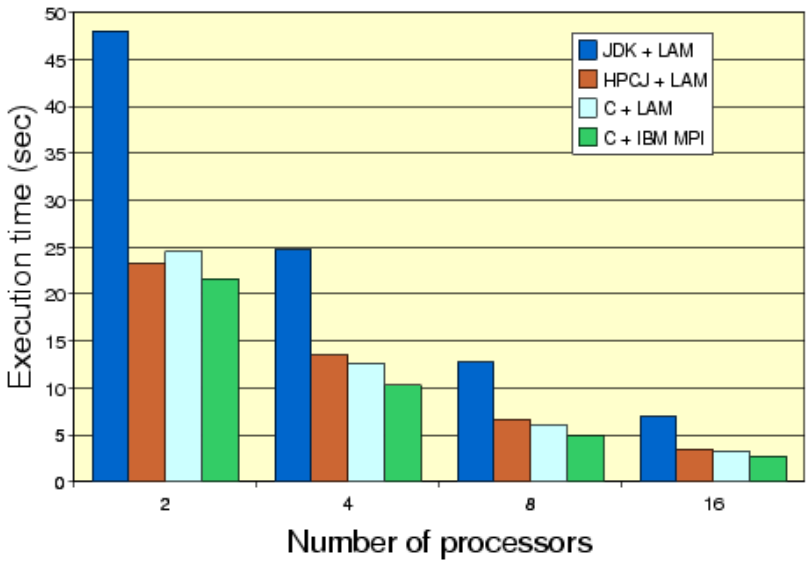


Figura 3.2: Se muestran los tiempos de ejecución para el test de benchmarking para el test *NAS parallel IS*.

llamado *intra-message pipeline*. Existe otro efecto que se da cuando se envían consecutivamente mensajes no relacionados uno tras otro, a nivel de aplicación, que se llama *inter-message pipeline*[VLA02]. En principio el comportamiento de un pipeline está definido por una relación lineal entre el tiempo y la longitud del trabajo a procesarse. El tiempo de comunicación puede modelarse como una función lineal de la longitud del mensaje, según se muestra en la Ecuación 3.4.

$$t_{com} = t_0 + t_{trans} n \tag{3.4}$$

donde t_0 es la latencia, t_{trans} es el tiempo de transmisión por byte, y n es la longitud del mensaje en bytes.

Otra forma simple y elegante de caracterizar el desempeño de la comunicación es extender la descripción de Hockney para procesadores vectoriales en pipeline, sobre el modelo anterior usando el ancho de banda asintótico, r_∞ y la *longitud de mensaje de medio-desmpeño*, *N-half* ($n_{\frac{1}{2}}$), como se muestra en la Ecuación 3.5.

$$t_{com} = \frac{n_{\frac{1}{2}} + n}{r_\infty} \tag{3.5}$$

.

El poder de esta fórmula radica en que provee un parámetro simple para ambos, el sistema de cómputo y la aplicación. En un sistema de comunicación, el ancho de banda máximo (asintótico) ocurre con mensajes de longitud infinita y se denota por r_∞. Este parámetro denota el máximo desempeño logrado por el subsistema de comunicaciones.

Por otro lado, el programador puede decir cuán real será el ancho de banda asintótico, comparando el promedio de los mensajes de su aplicación con el segundo parámetro, $n_{\frac{1}{2}}$. Si la longitud del mensaje es igual a $n_{\frac{1}{2}}$, entonces el ancho de banda real será la mitad del máximo asintótico.

Por otro lado, es necesario definir los test de benchmark que se aplicarán. Estos test pertenecen a la suite de benchmark de *Uno a Uno*. Los test *uno a uno* proveen la visión de aspectos espefíficos, relacionados a la sobrecarga asociada a la transición de mensajes de varios procesos a través de la capa de comunicación. La suite test *uno a uno* provee detalles en la sobrecarga y la escalabilidad de la comunicación entre procesos.

Posiblemente el test más simple de la medida del desempeño del paso de mensajes sea el "ping pong" o "eco" entre un par de procesos, como se muestra en la Figura 3.3. En este test, un proceso envía un buffer de longitud fija a otro proceso, quien lo recibe e inmediatamente lo regresa a quien se lo envió. El buffer contiene un arreglo de tipos primitivos. El tamaño del buffer es graduado a diferentes tamaños, para poder evaluar los efectos en latencia, ancho de banda, etc.

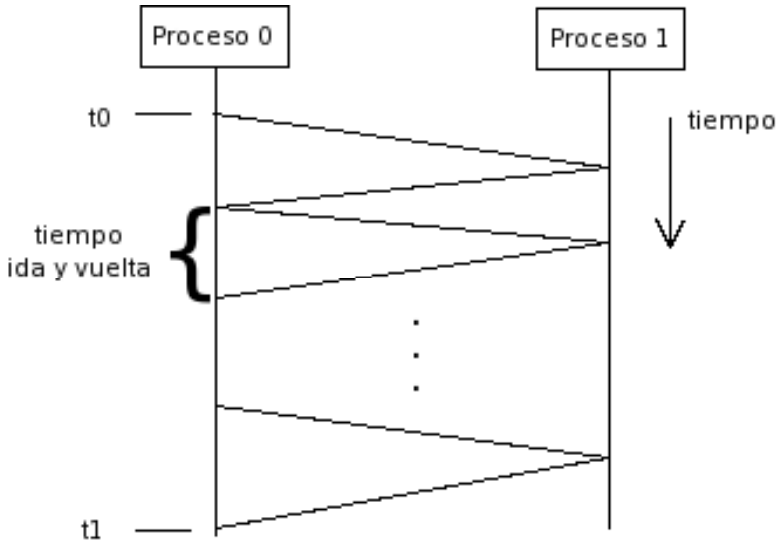


Figura 3.3: De interés en este test es el tiempo requerido para el paso de mensajes secuencial entre dos procesos.

Otro test interesante es el test "ping*-pong" entre dos procesos (ver Figura 3.4). Un proceso es responsable de enviar varios búfferes de longitud fija a un proceso receptor. Una vez que el receptor ha recibido todos los búfferes, envía una simple respuesta (ack). Los búfferes contienen tipos primitivos. A pesar de que este test es similar al test *ping-pong*, puede demostrar los efectos del *inter-message pipeline*.

Las pruebas se condujeron en una computadora IBM SP-2 con las siguientes configuraciones:

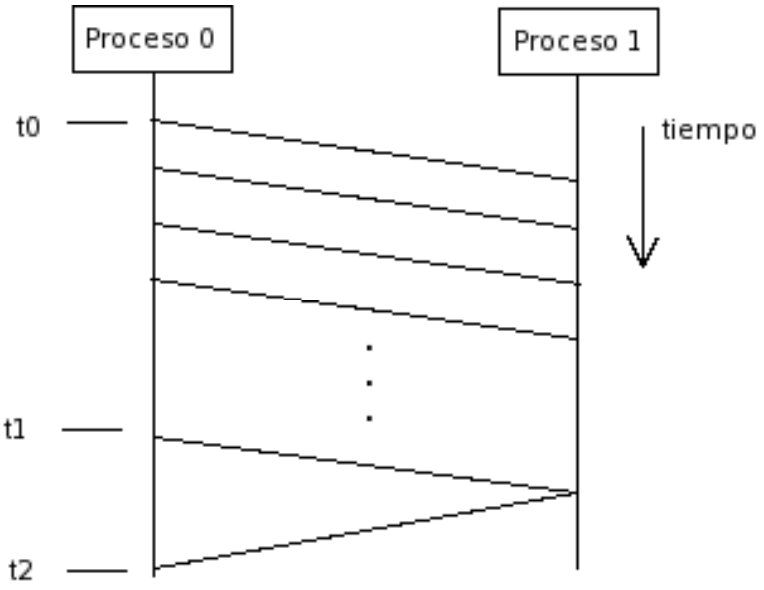


Figura 3.4: Este test pone en evidencia los efectos del Inter-Message Pipelining.

1. Código C usando MPI Nativo, la versión de MPI desarrollada por IBM para la SP-2, C/IBM-MPI.

2. Código C usando LAM-MPI, versión 6.1, C/LAM-MPI.

3. Código Java usando envoltorios JNI para la librería de LAM-MPI, Java/LAM-MPI.

La JVM y el compilador usados fueron parte del JDK 1.1.6 para la plataforma AIX. El ambiente de ejecución fue el Parallel Operating Environment (POE) de IBM, que soporta la carga y ejecución de procesos en paralelo en los nodos de la SP-2. La máquina posee 128 nodos con procesadores POWER2 Super Chip(P2SC). Los datos mostrados en los gráficos correspondientes representan los tiempos de transmisión para mensajes de longitudes totales de: 0, 40, 120, 400, 1200, 4000, 12000, 40000, 120000, 400000 y 1200000.

Los resultados de los test ping-pong se dan en las Figuras 3.5 y 3.6. La Figura 3.5 muestra los resultados cuando el buffer usado es un arreglo unidimensional de flotantes de doble precisión. La Figura 3.6 muestra los resultados cuando el buffer usado es un arreglo unidimensional de enteros.

Notables aspectos de estas mediciones son las latencias asociadas y los anchos de banda asintóticos de las configuraciones correspondientes que fueron calculados usando cuadrados mínimos. Para el escenario de búfferes de flotantes, el ancho de banda asintótico del ambiente Java/LAM-MPI es de 15.538 MByte/s para mensajes cortos, y de 40.868 MBytes/s para mensajes largos. Para el ambiente de C/LAM-MPI, el ancho

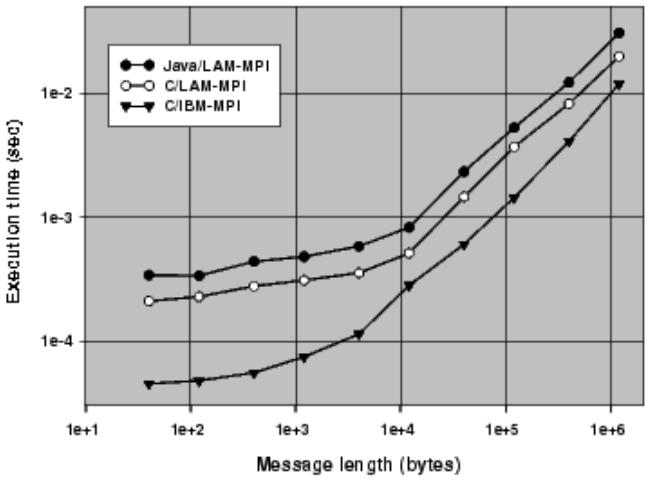


Figura 3.5: Tiempos para el test Ping-Pong, usando como contenido de los búfferes tipos de datos de doble precisión sobre la PS-2.

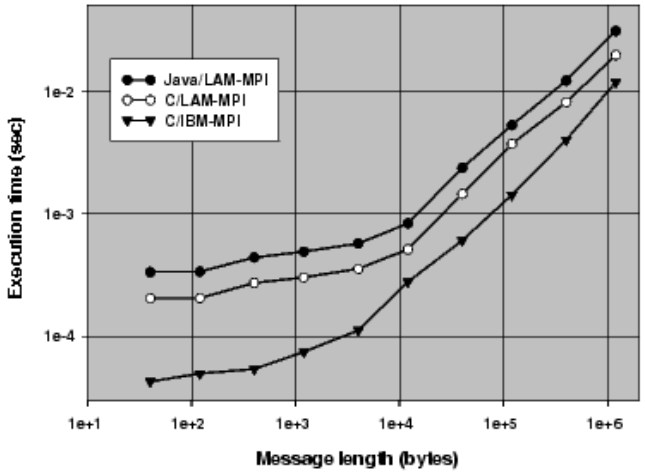


Figura 3.6: Tiempos para el test Ping-Pong, usando como contenido de los búffers tipos de datos enteros sobre la PS-2.

de banda asintótico es de 28.218 MBytes/s para mensajes cortos y 63.434 MBytes/s para mensajes largos[VLA02].

Otro aspecto importante de este test es la latencia. Para el escenario ping-pong, la latencia del ambiente Java/LAM-MPI es de 346.661μs para mensajes cortos contra 1566.916μs para mensajes largos. En contraste, la latencia para el ambiente de C/LAM-MPI es de 227.379μs para mensajes cortos y de 1136.036μs para mensajes largos. La latencia para el ambiente de C/IBM-MPI es de 45.5μs y 211.92μs correspondientemente

[VLA02].

Los resultados del test ping*-pong se muestran en las Figuras 3.7 y 3.8. La Figura 3.7 muestra los resultados cuando los búfferes usados son arreglos unidimensionales de flotantes de doble precisión. La Figura 3.8 muestra los resultados cuando los búfferes usados son arreglos unidimensionales de enteros. En el ambiente Java/LAM-MPI, el ancho de banda asintótico obtenido por cuadrados mínimos en los mensajes cortos y largos es de 50.497 MByte/s y 52.516 MByte/s. Los valores correspondientes para C/LAM-MPI son de 125.630 MByte/s y 154.088 MBytes/s. El efecto de *Inter-Message Pipeline* hace que los valores de C/IBM-MPI casi doblen el ancho de banda de C/LAM-MPI [VLA02].

Al mismo tiempo, las latencias asociadas con los ambientes decrementaron significativamente. Para Java/LAM-MPI la latencia es de 181.0μs y 175.7μs para los búfferes de flotantes y de enteros respectivamente. Los valores correspondientes para C/LAM-MPI son de 104.56μs y 105.0μs. Para C/IBM-MPI, los valores son de 8.33μs y 24.638μs respectivamente [VLA02].

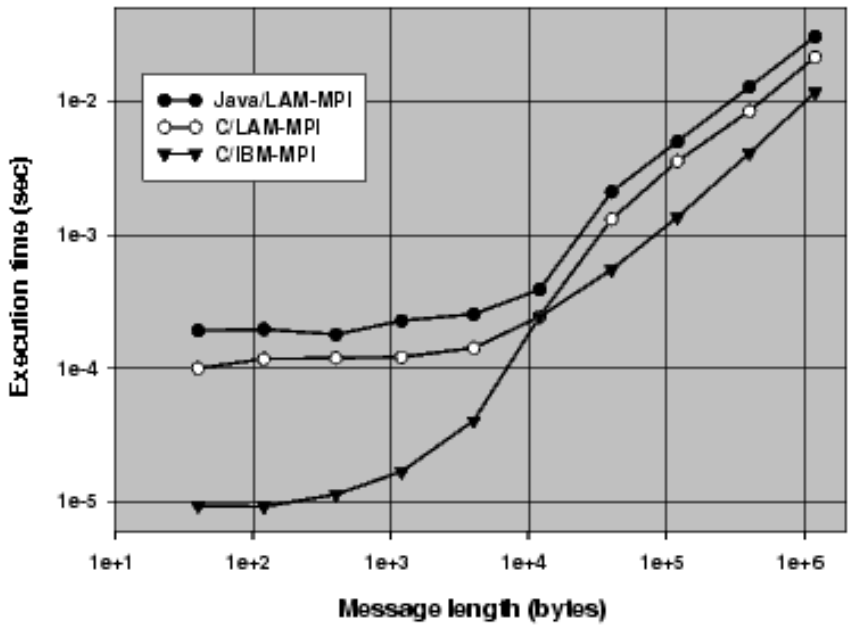


Figura 3.7: Tiempos para el test Ping-Pong, usando como contenido de los búfferes tipos de datos de doble precisión sobre la PS-2.

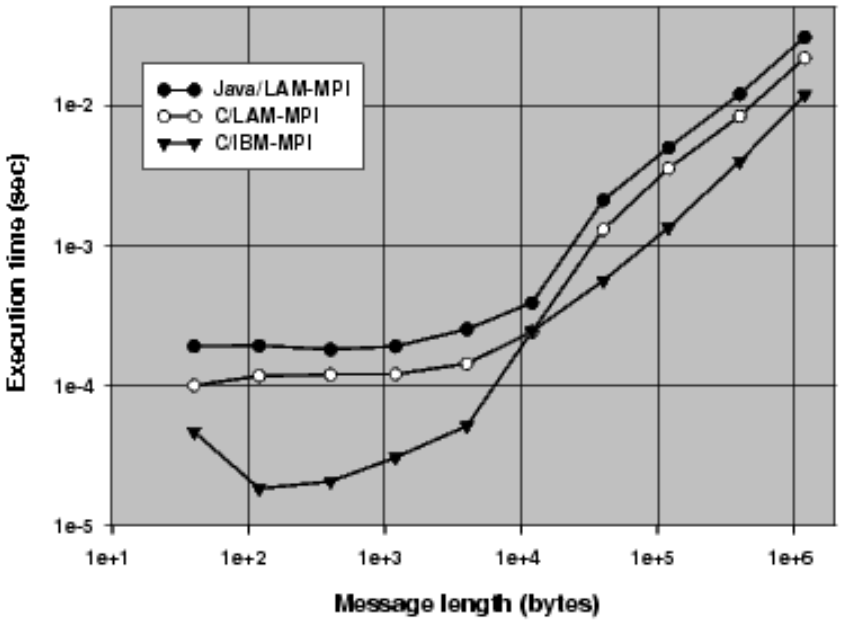


Figura 3.8: Tiempos para el test Ping-Pong, usando como contenido de los búffers tipos de datos enteros sobre la PS-2.

Capítulo 4

Análisis y Diseño

En este capítulo se presenta la descripción del problema a resolver. Partiendo de la descripción del problema, se identifican los primeros Requerimientos de Uso, que se emplean como base para el resto del análisis. Todo el análisis está compuesto por los Requerimientos de Uso más los diagramas UML de Casos de Uso, Actividades, Clases y Secuencia.

4.1. Descripción del Problema

Como se ha revisado en los capítulos anteriores, existe una gran diversidad de plataformas de hardware para la ejecución de sistemas paralelos y distribuídos. Existen topologías y tecnologías. Sin embargo, una de estas tecnologías en particular es la que está tomando gran parte del escenario, y se trata de los clusters.

Los clusters están ganando un gran terreno, ya que ofrecen niveles de desempeño aceptables por un relativo costo bajo.

En el terreno del software, existe una diversidad de herramientas que le ayuda al programador a desarrollar aplicaciones tanto paralelas como distribuídas. Algunas de estas herramientas se están estandarizando, mientras otras apenas están ganando aceptación.

Gran parte del tiempo dedicado al desarrollo de aplicaciones paralelas y distribuídas se dedica al manejo de las estructuras o medios de comunicación que ofrecen las herramientas existentes, a la administración de recursos de la plataforma y al control de eventos del sistema entre otros.

En particular, hablando del lenguaje Java, pocos son los esfuerzos que se han concentrado en desarrollar herramientas de este tipo. Algunas de estas herramientas ya se encuentran en fase madura, otras apenas se están diseñando. Gracias a la creación del Foro de Java Grande, el interés y esfuerzo se ha enfocado en la estandarización y desarrollo de tales herramientas para el lenguaje Java. Sin embargo, aún con la disponibilidad de tales herramientas, hacen falta niveles de abstracción mayores que faciliten al desarrollador la tarea de programar. Se requieren herramientas que le ayuden al desarrollador enfocar sus esfuerzos principalmente en la implementación de la solución de

su problema inicial, y que le ayuden a disminuir el tiempo que dedica a resolver detalles de comunicación, control de eventos y administración de recursos.

La Plataforma se ejecuta en un cluster de computadoras Linux, aunque por la naturaleza de Java, debe poder migrar con mínimo o ningún esfuerzo a otras plataformas.

La Plataforma esta dirigida al desarrollo de aplicaciones paralelas Java que utilicen tanto memoria compartida, como memoria distribuida y de total transparencia para el usuario. Como se ha desarrollado en Java sigue el paradigma orientado a objetos. Al usuario se le ofrecen varias interfaces de simple implementación, que lo van a ir guiando de manera adecuada para lograr el desarrollo eficiente de su aplicación en paralelo. En las aplicaciones en paralelo el manejo de la memoria compartida se convierte en un problema engorroso, el programador debe dominar el manejo de la exclusión mutua, para garantizar el acceso eficiente a los recursos compartidos, aspecto que se dificulta en gran medida, pues debe conocer el concepto de semáforos, monitores u otro mecanismo. Otra ventaja importante que ofrece esta Plataforma es el manejo de los mensajes, ya que el programador se puede olvidar del protocolo de los mismos, sólo se tiene que preocupar por decidir cual tipo de mensaje le interesa y en que momento va a usar cada uno, es decir, comunicación síncrona y asíncrona.

Se pretende que la Plataforma brinde el soporte y/o administración de lo siguiente:

- Soporte de la implementación de mpiJava. Las librerías de MPI no están originalmente diseñadas para soportar la concurrencia. Pero se pretende que la Plataforma libere al programador del manejo concurrente de las librerías de MPI.

- Soporte de *servicios* de memoria compartida. Además del soporte de paso de mensajes que tendrá a la mano el desarrollador, podrá hacer uso de zonas de memoría compartidas. La Plataforma se encargará de la consistencia de estas zonas y del envío y refresco automático de información.

- Soporte para un modelo simple de pool de hilos para cada uno de los nodos de ejecución, con los mecanismos que vigilen su buen uso y desempeño.

- Soporte para el envío de paquetes (independientemente de MPI), usando un socket UDP totalmente transparente al usuario.

- Soporte para la ejecución de N tareas paralelas usando M nodos físicos en el cluster.

- Administración de nodos que conforman parte de la red de nodos de la Plataforma.

La Plataforma se ha desarrollada en Java 1.5. A partir de la versión 1.5, Java cuenta con un paquete especializado para el desarrollo de secciones concurrentes, se trata del paquete `java.util.concurrent`. Si el usuario necesita desarrollar algún bloque concurrente, puede usar dicho paquete de manera muy simple, esto es totalmente independientemente del manejo concurrente a nivel de la Plataforma, que lo usa sustancialmente para el manejo de la memoria compartida.

El análisis y Diseño del sistema se ha realizado en UML, y se han seguido las siguientes etapas:

- Requerimientos de uso. Se partirá de los requerimientos de uso, que son la descripción de las funcionalidades que se espera del sistema.
- Casos de uso. Se generarán los diagramas de casos de uso a partir de los requerimientos de uso. Los diagramas de Casos de uso son el "contrato gráfico" que denotará las funcionalidades mínimas que debe cumplir el sistema.
- Actividades. Se generarán los diagramas de actividades. Los casos de uso denotarán algunas (si no es que todas) las actividades principales.
- Clases. Se generarán los diagramas de clases. Partiendo de los diagramas de actividades, se pueden generar de manera más precisa los diagramas de clases. Desde los diagramas de Casos de uso, se podrán detectar las primeras clases, que pueden surgir directamente de los actores. Con los diagramas de actividades pueden surgir clases directamente de alguna actividad, aunque es más común que sean los diagramas de actividades las que arrojen los métodos que deben cumplir las clases.
- Secuencia. Una vez comenzados los diagramas de clases, se puede comenzar a generar los diagramas de secuencia. En los diagramas de secuencia se termina de refinar el comportamiento que debe cumplir cada clase.

4.1.1. Requerimientos de Uso

En esta sección se describen los requerimientos de uso. Los requerimentos de uso son la descripción *verbal* (valiéndose de la expresión, ya que éste es un escrito) de las funcionalidades que se espera que cumpla el sistema. De ninguna manera se espera que la descripción de los requerimientos de uso denote la forma en que se desarrollará el sistema o cumpla cierta funcionalidad esperada, salvo casos bien localizados.

Esta lista de funcionalidades que el usuario del sistema espera que cumpla, servirá como base para detectar las tareas adicionales que se tienen que realizar para completar tales funcionalidades. Entonces, esta lista de funcionalidades es una propuesta desde el punto de vista del usuario del sistema (que en este caso también se trata de un desarrollador).

Requerimientos de uso:

- El sistema debe permitir conformar una red de ejecución (la Plataforma) sobre un cluster de computadoras o bien una red de computadoras de uso general, agregando o eliminando nodos de ejecución a la red.
- El sistema debe permitir ejecutar clases de Java en forma paralela en diferentes nodos del sistema.

- Que el sistema permita y facilite el uso de hilos, su control y administración en cada nodo.
- El sistema debe permitir el uso de llamadas a las librerías de MPI.
- El sistema debe permitir el paso de paquetes de forma adicional a MPI.
- Que el sistema ofrezca servicios de nombres de tareas que faciliten la comunicación si es necesario realizarla.
- Que el sistema ofrezca una zona de memoria compartida que pueda ser accedida de forma transparente, independientemente de la ubicación real de los datos.
- El sistema debe permitir la distribución de clases Java para su ejecución en cualquier nodo que forme parte de la red de la Plataforma.
- El sistema debe poder sincronizarse a nivel de la Plataforma. Esto puede servir, entre otras cosas, para iniciar tareas al mismo tiempo.
- El sistema debe balancear la carga de trabajo entre la red de la Plataforma, tarea relativamente sencilla, ya que todos los procesos son exactamente iguales.
- El sistema debe, al momento de repartir la carga de trabajo, intentar dejar un proceso en cada nodo. Si existen más procesos que nodos, algunos nodos deberán atender más de un proceso.
- Cuando el sistema reparte más de un proceso por nodo, debe ofrecer un mecanismo para identificar cada uno de esos procesos con la intención de controlarlos, ya sea a nivel de la Plataforma o a nivel de la aplicación del usuario.
- Cuando exista más de un proceso por nodo, las comunicaciones que emplee cada proceso no deben interferir entre los mismos.
- El sistema debe contar con un elemento centralizado que permita el control de la Plataforma completa, encargado de tareas centrales, tales como la distribución de procesos y el control de los nombres de los nodos.
- El elemento centralizado debe recibir la especificación del usuario acerca del número y nombre de las clases a ejecutar.
- El sistema debe contar con un elemento por nodo que le permita recibir las clases para iniciar su ejecución, y si es necesario, levantar más instancias de los procesos.

4.1.2. Casos de Uso

Partiendo de los Requerimientos de uso, se genera el diagrama de Casos de Uso. Para el diagrama de casos de uso, se encontraron 4 actores:

- Desarrollador: el primer actor que se detectó es el usuario de la Plataforma. Interactúa con el sistema solicitando la ejecución de sus clases. La aplicación del usuario está conformada por 2 clases principales que se denotarán en el diagrama de clases.

- Demonio Maestro: dentro del sistema, se detectó que se necesitaba un elemento que se encargue de vigilar y administrar la Plataforma. Este elemento, que es una pieza de código, se modeló como un actor por conveniencia, ya que sus actividades las realizará sobre la Plataforma.

- Demonio Esclavo: al igual que con el demonio maestro, es necesario un elemento que se encargue de iniciar la ejecución de las clases del usuario. Por conveniencia, este elemento es una pieza de código que se modela como actor.

- RMIServer: de los requerimientos se observa la necesidad de un mecanismo para la distribución de las clases del usuario. Como la Plataforma es desarrollada en Java, esta distribución se realiza con RMI. RMI es parte externa del sistema, como si fuera una pieza de caja negra, razón por la cual se modela como un actor.

El diagrama de casos de uso se muestra en la Figura 4.1.

El *Demonio Maestro* es el que hace uso de las funcionalidades de *Administración de la Plataforma*, lo que incluye:

- Balance de Carga: a través de esta funcionalidad se repartirá la ejecución de las clases a lo largo de la Plataforma.

- Distribución de Clases: a través de esta funcionalidad se envían las clases a los nodos que conforman la red de la Plataforma.

- Manejo de Nodos: uno de los requerimientos de uso establece que se debe administrar los nodos, ofreciendo las siguientes funcionalidades:

  - Servicio de Nombres de nodo: consiste en nombrar cada uno de los nodos que conforman la red de la Plataforma.
  - Agregar Nodos: consiste en agregar a la lista de nodos un nuevo nodo.
  - Eliminar Nodos: consiste en eliminar de la lista un nodo que haya abandonado la red.

- Sincronización: Permite crear puntos de ejecución en los cuales las actividades que se estén realizando en un nodo se detengan hasta que la ejecución en todos los nodos llegue al mismo punto. Es muy útil para iniciar el arranque de las tareas al mismo tiempo.

El *Desarrollador* hará uso de las funcionalidades del caso de uso de *Ejecución de Clases*, que ofrece:

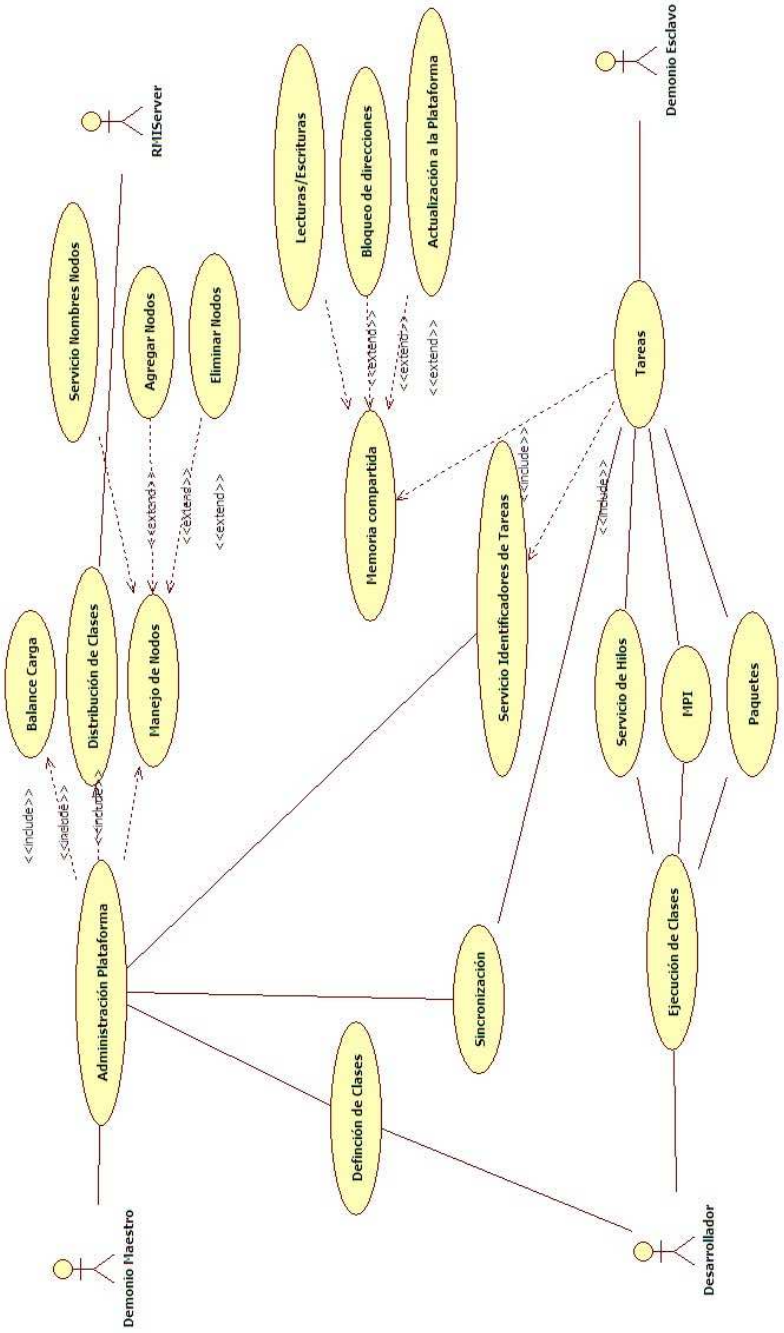


Figura 4.1: Se muestra en un diagrama único los casos de uso de la Plataforma.

- Servicio de Hilos: le ofrece al desarrollador un *pool* de hilos controlado.
- Servicio de MPI: resuelve la concurrencia que no soportan las llamadas a la librería de MPI, ahorrándole el trabajo al desarrollador.
- Paquetes: le ofrece al desarrollador el envío y recepción de paquetes de forma adicional a MPI, usando sockets UDP.

Además de las anteriores, el desarrollador también puede hacer uso de la *Definición de Clases*, que le ayudará a definir qué clases se ejecutarán en los nodos.
Por otra parte, el *Demonio Esclavo* soporta las funcionalidades de las *Tareas*:

- Servicio de Identificadores de Tareas: recibe del *Demonio Maestro* el nombre de la tarea y lo asigna a la tarea.
- Ofrece la implementación del Servicio de hilos, MPI y Paquetes.
- Manejo de memoria: uno de los requerimientos de uso establece que se debe ofrecer una zona de *memoria compartida*, ofreciendo las siguientes funcionalidades:
  - Lecturas/Escrituras: Permite la lectura o escritura de cualquier dirección de memoria, resolviendo la paginación a disco por el tamaño de la memoria y la concurrencia con otras tareas del mismo nodo.
  - Bloqueo de direcciones: Al igual que la implementación de una memoria compartida en hardware, se ofrece un mecanismo sencillo de bloqueo de direcciones. Con esto, una tarea puede bloquear una dirección, teniendo acceso exclusivo a dicha dirección para lectura o escritura. Solamente la tarea propietaria del bloqueo puede liberar dicho bloqueo.
  - Actualización a la Plataforma: Todos los cambios que se realicen a la memoria compartida en un nodo se deben reflejar en el resto de la Plataforma para el resto de las tareas. Esto supone una tarea un tanto compleja, ya que en cada nodo se está realizando paginación a disco.

4.1.3. Diagramas de Actividades

Con la definición de los casos de uso, ya se puede a comenzar a describir las actividades que se espera que cumpla el sistema. En las siguientes subsecciones se mostrarán los diagramas que representan tales actividades.

Diagrama de Actividades General

Se parte del diagrama General de actividades, que se muestra en la Figura 4.2. En este diagrama se listan las actividades generales que se realizan sobre la Plataforma. Las actividades se ejecutan secuencialmente en el siguiente orden:

1. Se inicia el Demonio Maestro, quien controla la Plataforma

Figura 4.2: Las actividades en general que suceden en la Plataforma.

2. Se inician los Demonios Esclavos, uno en cada nodo de la red.
3. Cuando se inicia cada Demonio Esclavo, éste comunica al Demonio Maestro de su existencia y es registrado por el Demonio Maestro.
4. El usuario envía la *Definición de Clases* al Demonio Maestro.
5. Conforme la Definición de Clases del usuario, el Demonio Maestro se comunica con los Demonios Esclavos para hacer la distribución de las clases.
6. En cada nodo donde exista un Demonio Esclavo, se inicia una Tarea que es la instancia de la clase del usuario.
7. Al final de la ejecución de las Tareas, se pueden cerrar los Demonios Esclavos.
8. Una vez cerrados los Demonios Esclavos, se puede cerrar el Demonio Maestro.

En el diagrama de actividades de la Figura 4.2, se denotan las actividades mencionadas anteriormente.

Diagrama de Actividades de los Demonios

En la Figura 4.3(a) se encuentran las actividades que suceden cuando se inicia un Demonio Esclavo:

1. Se realiza la conexión al Demonio Maestro a un puerto bien definido.
2. Se envía el nombre del nodo al Demonio Maestro.
3. Se registra el nombre del nodo (Demonio Esclavo) en el Demonio Maestro.

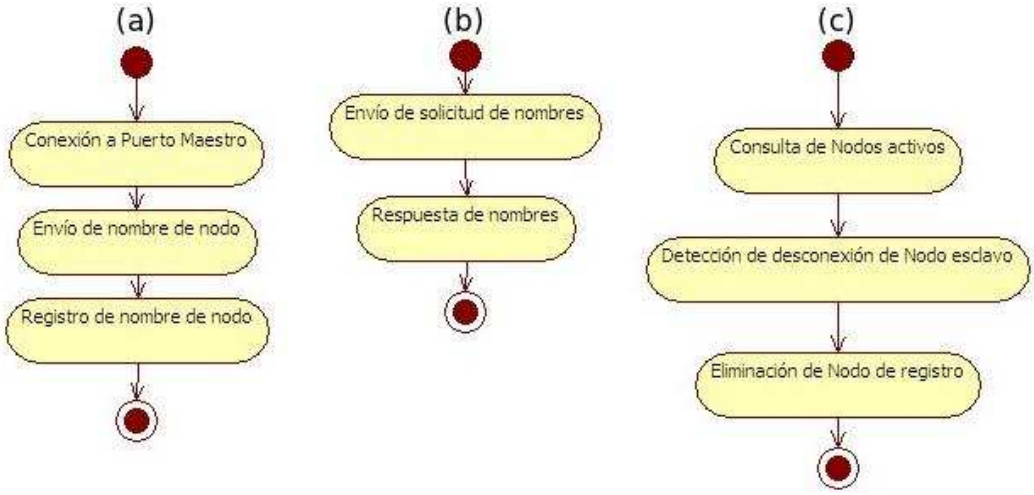


Figura 4.3: (a) y (b): Actividades de los demonios esclavos. (c): Actividades del demonio maestro.

En la Figura 4.3(b) se encuentran las actividades que suceden cuando se solicitan los nombres de los nodos (Demonios Esclavos):

1. Se envía la solicitud de los nombres al Demonio Maestro.
2. El Demonio Maestro envía la lista de nombres.

En la Figura 4.3(c) se encuentran las actividades que suceden cuando se desconecta un nodo de la red:

1. Se hace la consulta de los nodos activos mediante la red de comunicación que se levantó cuando cada Demonio Esclavo se anunció al Demonio Maestro.
2. Se detectan los Demonios Esclavos que se desconectaron.
3. Se remueven de la lista del Demonio Maestro los Demonios Esclavos que no están activos.

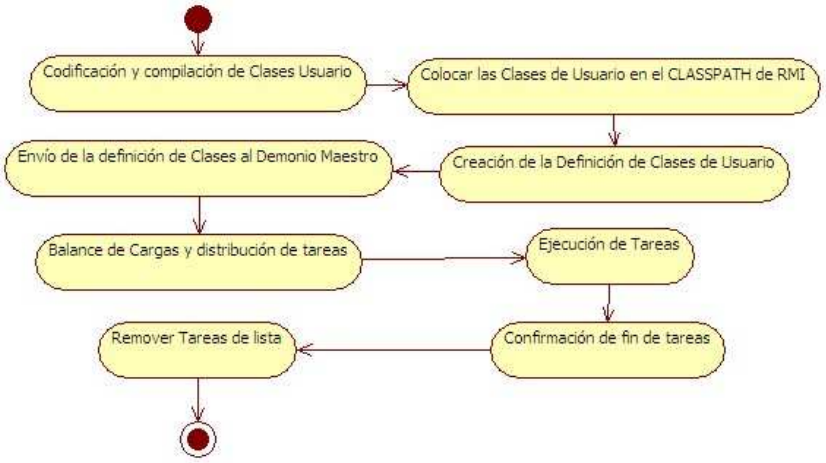


Figura 4.4: Se muestran las actividades que se llevan a cabo para realizar la ejecución de las clases del usuario.

Diagrama de Actividades de las Tareas de Usuario

En la Figura 4.4 se encuentran las actividades que suceden desde que el usuario crea sus clases hasta que se ejecutan:

1. El usuario crea las clases que implementan la solución a su problemática, codificándolas y compilándolas
2. El usuario coloca sus clases en algún directorio dentro del `CLASSPATH` del servidor `RMI`.
3. El usuario define la cantidad de instancias de cada una de sus clases, junto con sus nombres.
4. El usuario envía la Definición de Clases al Demonio Maestro.
5. El Demonio Maestro hace el balance de carga y el envío de clases, distribuyendo las tareas. Conforme se van distribuyendo las clases, el Demonio Maestro les asigna su nombre o identificador de tarea.
6. Los Demonios Esclavos inician la ejecución de las Tareas.
7. Cuando la tarea ha terminado de ejecutarse, se envía la confirmación al Demonio Maestro.
8. El Demonio Maestro remueve la Tarea de la lista de Tareas registradas.

4.1.4. Diagramas de Clases

Con base en los diagramas de Casos de Uso y de Actividades, se identifican las primeras clases que se requieren para el desarrollo de la Plataforma.

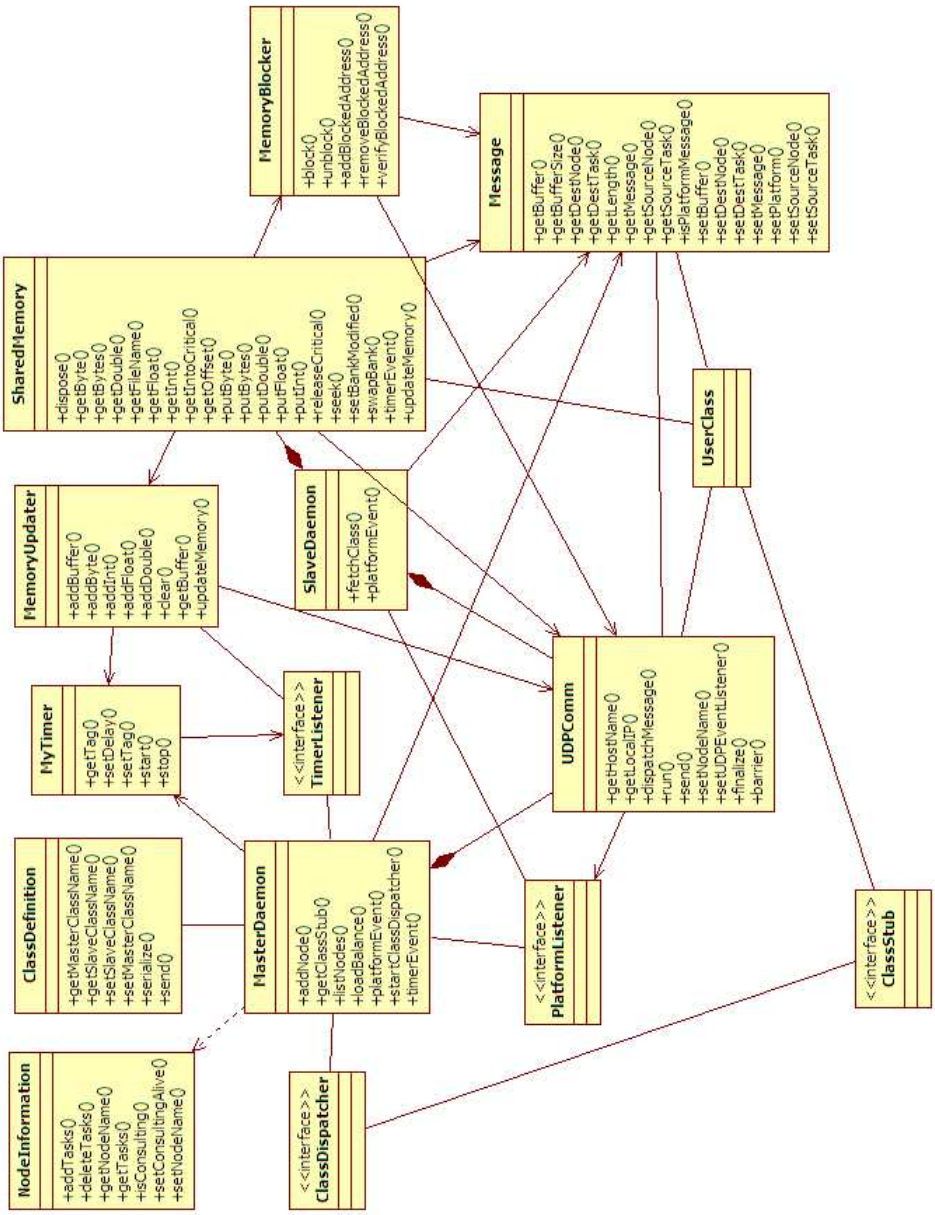


Figura 4.5: Se muestran las clases y las relaciones que guardan entre sí. Este diagrama muestra todas las clases de la Plataforma.

Al ir desarrollando a la par los diagramas de secuencia, aparecen las clases necesarias para coordinar y complementar el desarrollo.

En la Figura 4.5 se encuentran todas las clases que conforman el desarrollo de la Plataforma. En el diagrama 4.5 se encuentran todas la clases y las relaciones que las unen. A continuación se discuten los 4 diagramas de las clases principales que conforman el sistema, omitiendo las clases y relaciones a las cuales no tenga acceso la clase en cuestión.

Diagrama de Clases Maestro

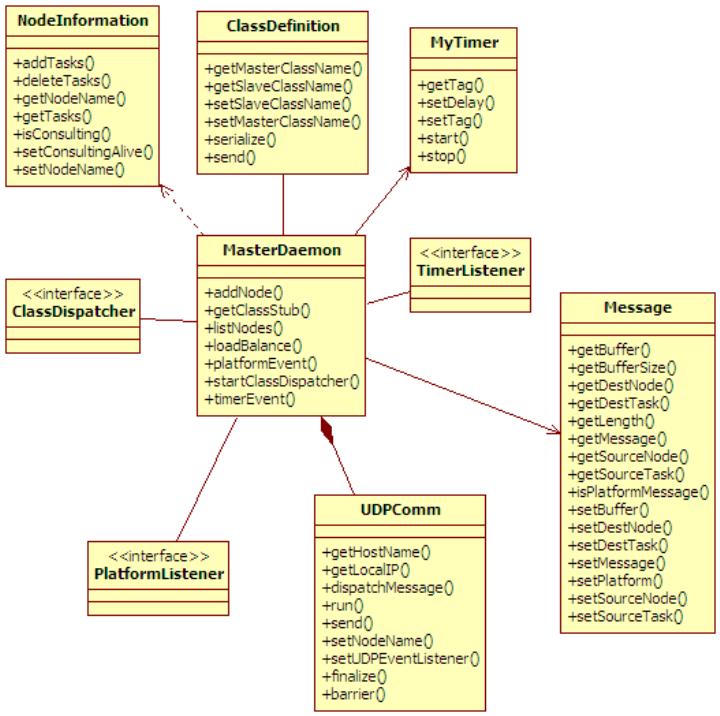


Figura 4.6: Se muestran las clases relacionadas con la clase MasterDaemon, omitiendo las clases no relacionadas.

En la Figura 4.6, se muestran las clases relacionadas directamente con la clase `MasterDaemon`. La clase `MasterDaemon` será la responsable de vigilar y administrar toda la Plataforma, de balancear la carga de clases y de administrar los nodos que conformen la Plataforma.

La clase `MasterDaemon` guarda una relación de dependencia con la clase `NodeInformation`. La clase `NodeInformation` almacena la información de los nodos que conforman la Plataforma, junto con la carga de tareas que tiene cada nodo. Cada vez que se registra un nodo, o que abandona la Plataforma, los cambios se almacenan en la clase

`NodeInformation`. La clase `NodeInformation` no tiene noción de la existencia de la clase MasterDaemon.

La clase `MasterDaemon` guarda una relación directa con la clase `ClassDefinition`. La clase `ClassDefinition` debe ser heredada por una clase de usuario para poder definir el nombre de las clases principal y secundarias, junto con el número de instancias de cada una de ellas, que se requieren en la Plataforma. La clase `MasterDaemon` conoce de la existencia de la clase `ClassDefinition`, y viceversa.

La clase `MasterDaemon` guarda una relación direccional con la clase `MyTimer`. La clase `MyTimer` es empleada por la clase `MasterDaemon` para generar eventos a intervalos regulares configurables. La clase `MasterDaemon` implementa la interfaz `TimerListener` para poder atender los eventos que genere la clase `MyTimer`. Cuando se genera un evento por `MyTimer`, la clase `MasterDaemon` genera un mensaje *PING* que es enviado a todos los nodos, que a su vez responderán para indicarle al `MasterDaemon` que siguen vivos. La clase `MyTimer` no tiene noción de la existencia de la clase `MasterDaemon`.

La clase `MasterDaemon` guarda una relación direccional con la clase `Message`. La clase `MasterDaemon` emplea la clase `Message` para generar mensajes que después puede enviar o recibir a través de la clase `UDPComm`. Para recibir los mensajes, la clase `MasterDaemon` implementa la interfaz `PlatformListener`. La clase `Message` no tiene noción de la existencia de ninguna otra clase. La clase `UDPComm` no tiene conocimiento de la clase `MasterDaemon`, pero guarda una relación de composición con ésta última.

Por último, esta clase implementa la interfaz remota `ClassDispatcher`, que es expuesta por el servidor RMI de Java para la distribución de las clases del usuario.

Diagrama de Clases Esclavo

En la Figura 4.7 se presentan las clases que tienen relación directa con la clase `SlaveDaemon`. Las clases no relacionadas se omiten. La clase `SlaveDaemon` se instanciará en cada uno de los nodos que conformen la red de la Plataforma. Es tarea del usuario levantar cada una de estas instancias. La clase `SlaveDaemon` es responsable de solicitar las clases del usuario, proporcionarle a las clases del usuario las facilidades de comunicación, control de hilos, comunicaciones empleando MPI, métodos de sincronización y de memoria compartida.

La clase `SlaveDaemon` guarda una relación direccional con la clase `Message`; cada clase `SlaveDaemon` emplea la clase `Message` para generar mensajes y enviarlos por red, o para recibirlos. Con el objetivo de recibir los mensajes, la clase `SlaveDaemon` implementa la interfaz `PlatformListener`. Para lograr el envío de mensajes, la clase `SlaveDaemon` usa a la clase `UDPComm`. La clase `Message` no tiene noción de la existencia de ninguna clase. La clase `UDPComm` no tiene noción de la existencia de la clase `SlaveDaemon`, y guarda una relación de composición con ésta.

La clase `SharedMemory` guarda una relación de composición con la clase `SlaveDaemon`, ya que su tiempo de vida depende de la duración de ámbito de `SlaveDaemon`.

La clase `SlaveDaemon` tiene una relación con la interfaz `ClassStub`. La clase `SlaveDaemon` no tiene noción directa de la existencia de la clase `UserClass`, solo la conoce por la interfaz `ClassStub` que ésta implementa.

Figura 4.7: Se muestran las clases relacionadas con la clase SlaveDaemon, omitiendo las clases no relacionadas.

Diagrama de las Clases de Usuario

En la Figura 4.8, se presentan las clases que tienen relación directa con las clases del usuario. De antemano la Plataforma no conoce en lo absoluto las clases del usuario. Lo único que conoce acerca de ellas, es que implementan la interfaz `ClassStub`.

Las clases del usuario, una vez compiladas, se deben encontrar en el `CLASSPATH` del `rmiregistry`, para que este pueda distribuirlos a los nodos clientes. El punto de contacto entre los nodos esclavos y el nodo maestro, para la distribución de las clases, se encuentra en la interfaz remota que implementa el nodo maestro, `ClassDispatcher`.

Las clases del usuario guardan relación directa con las clases `SharedMemory`, `Message`, `SlaveDaemon` y `UDPComm`. Estas clases no tienen noción de la existencia de las clases del usuario, pero esta última sí tiene noción de la existencia de las clases.

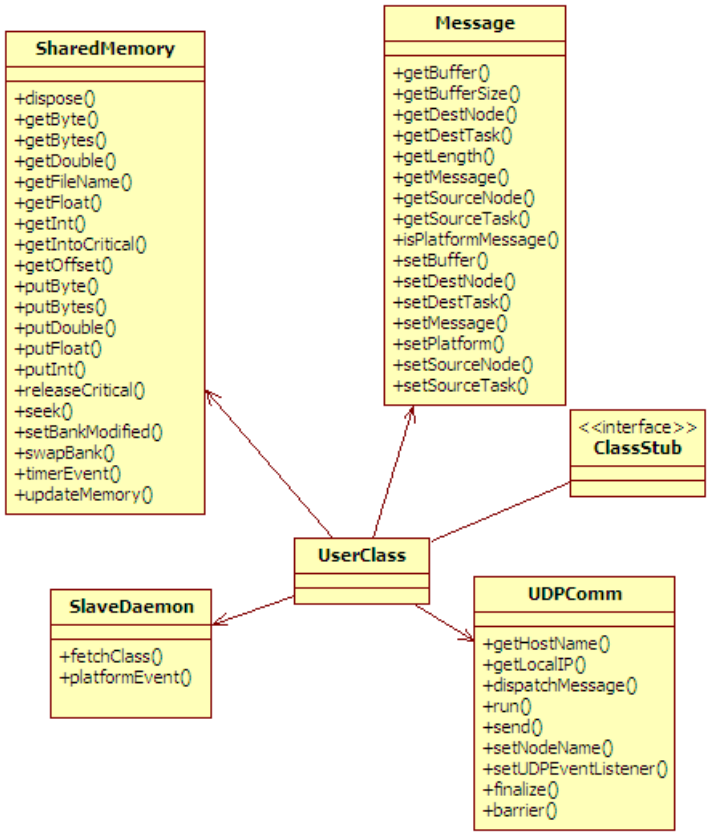


Figura 4.8: Se muestran las clases relacionadas con las clases del usuario, omitiendo las clases no relacionadas.

Diagrama de las Clases de Memoria

La clase `SharedMemory` se emplea para ofrecer el servicio de memoria compartida, de hecho, esta clase puede funcionar aún sin tener acceso a la red, en cuyo caso hace las veces de memoria local. La clase `SharedMemory` ofrece un conjunto limitado de direcciones de memoria, en el cual se pueden almacenar tipos de datos primitivos. Si esta clase tiene acceso a la red, puede comunicar a otras instancias de esta clase que se encuentren en otros nodos acerca de los cambios que se realicen en ella, manteniéndose actualizada.

La clase `SharedMemory` tiene una relación direccional con la clase `MemoryUpdater`. La clase `MemoryUpdater` es la responsable de llevar el rastreo de los cambios realizados en la clase `SharedMemory`, y los comunica después de cierto intervalo de tiempo al resto de las instancias. La clase `MemoryUpdater` emplea una instancia de la clase `MyTimer`, e implementa la interfaz `TimerListener`, para poder hacer la notificación de cambios

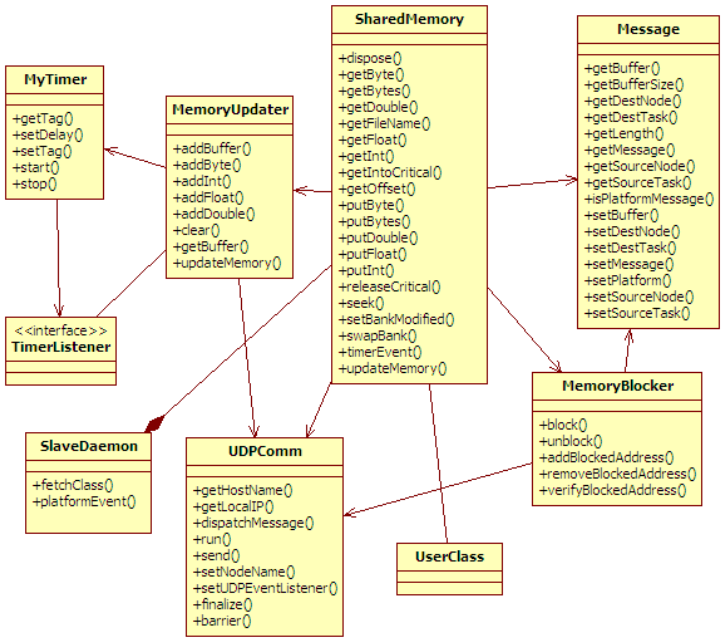


Figura 4.9: Se muestran las clases relacionadas con la clase SharedMemory, omitiendo las clases no relacionadas.

a intervalos regulares. Los cambios que son notificados por parte de las instancias de la clase `SharedMemory` de otros nodos, se reciben en `SharedMemory`, aunque es la clase `MemoryUpdater` la encargada de hacer las actualizaciones.

La clase `SharedMemory` guarda una relación de composición con la clase `SlaveDaemon`.

La clase `MemoryBlocker` es empleada por la clase `SharedMemory` para registrar las direcciones de memoria que el usuario ha bloqueado. La clase `MemoryBlocker` no tiene noción de la existencia de la clase `SharedMemory`, lo que a primer instancia puede sonar extraño. La clase `MemoryBlocker` solamente lleva el registro de "direcciones" como valores absolutos, razón por la cual no necesita conocer nada acerca de la clase `SharedMemory`.

Las clases `MemoryUpdater` y `MemoryBlocker` tienen una relación direccional con la clase `UDPComm`, a la cual emplean para anunciar por red los cambios que hayan sufrido.

Las clases de usuario hacen uso directo de la clase `SharedMemory`, aunque esta última no tenga noción de la existencia de tales clases.

4.1.5. Diagramas de Secuencia

Con los diagramas de clases definidos, se pueden generar los diagramas de secuencia. En los diagramas de secuencia se muestra el orden en que suceden algunas tareas para llevar a cabo una tarea determinada. Se presentan a continuación 3 diagramas de clases

que muestran las operaciones más importantes de la Plataforma: el arranque de la Plataforma, la ejecución de una clase del usuario y la actualización de la memoria compartida.

Diagrama de Secuencia del Arranque de la Plataforma

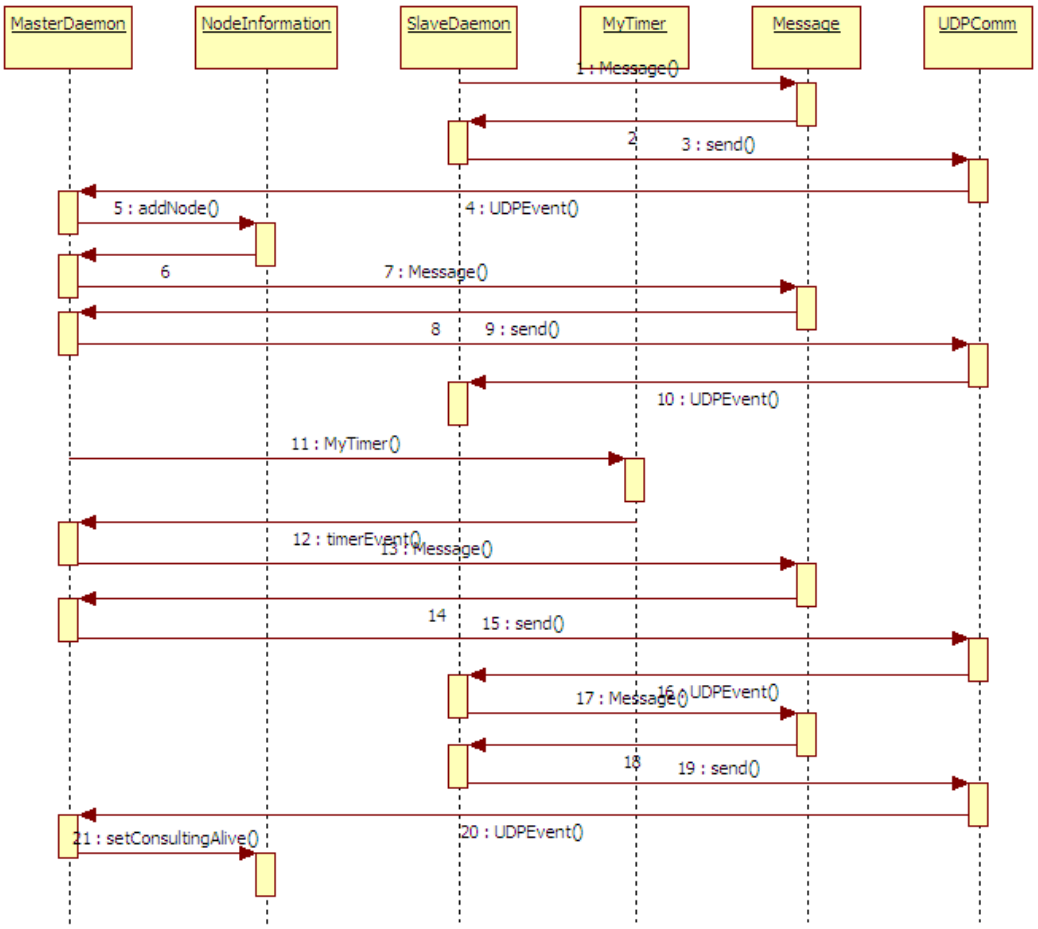


Figura 4.10: Se muestra la secuencia que sucede cuando se arranca la Plataforma.

Para efectos de simplicidad, el diagrama supone que una instancia de `MasterDaemon` ya se encuentra ejecutando. Cuando arranca la Plataforma sucede la siguiente secuencia (Figura 4.10):

1. `SlaveDaemon` genera una instancia de `Message`, con el mensaje `ADD_NODE`.
2. El mensaje se construye y el flujo de ejecución regresa a `SlaveDaemon`.

3. `SlaveDaemon` emplea a `UDPComm` para enviar el mensaje recién creado a `MasterDaemon`.
4. El `UDPComm` de `MasterDaemon` recibe el mensaje, y genera un `UDPEvent()` para notificarle del mensaje.
5. El `MasterDaemon` recibe el mensaje y agrega los datos del nodo en `NodeInformation`.
6. Los datos del nodo nuevo se agregan, y el flujo de ejecución regresa a `MasterDaemon`.
7. `MasterDaemon` genera una instancia de `Message`, con el mensaje `LIST_NODES`.
8. El mensaje se construye y el flujo de ejecución regresa a `MasterDaemon`.
9. `MasterDaemon` emplea a `UDPComm` para enviar el mensaje recién creado a `SlaveDaemon`.
10. El `UDPComm` de `SlaveDaemon` recibe el mensaje, y genera un `UDPEvent()` para notificarle del mensaje al `SlaveDaemon` de los nodos disponibles.
11. En determinado momento (al arranque, de hecho), el `MasterDaemon` crea una instancia de `MyTimer`.
12. En cualquier otro momento, el `MyTimer` instanciado genera un evento `timerEvent()` que notifica a `MasterDaemon`.
13. `MasterDaemon` genera una instancia de `Message` con el mensaje `PING`, que es usado para solicitar a un nodo una respuesta `PONG`.
14. El mensaje se cosntruye y el flujo de ejecución regresa a `MasterDaemon`.
15. `MasterDaemon` emplea a `UDPComm` para enviar el mensaje recién creado a `SlaveDaemon`.
16. El `UDPComm` de `SlaveDaemon` recibe el mensaje, y genera un `UDPEvent()` para notificarle del mensaje al `SlaveDaemon`.
17. El `SlaveDaemon` genera una instancia de `Message` con el mensaje `PONG`.
18. El mensaje se construye y el flujo de ejecución regresa a `SlaveDaemon`.
19. `SlaveDaemon` emplea a `UDPComm` para enviar el mensaje recién creado a `MasterDaemon`.
20. El `UDPComm` de `MasterDaemon` recibe el mensaje, y genera un `UDPEvent()` para notificarle del mensaje al `MasterDaemon`.
21. El `MasterDaemon` invoca a `setConsultingAlive()` sobre `NodeInformation` para marcar el nodo como 'vivo'.

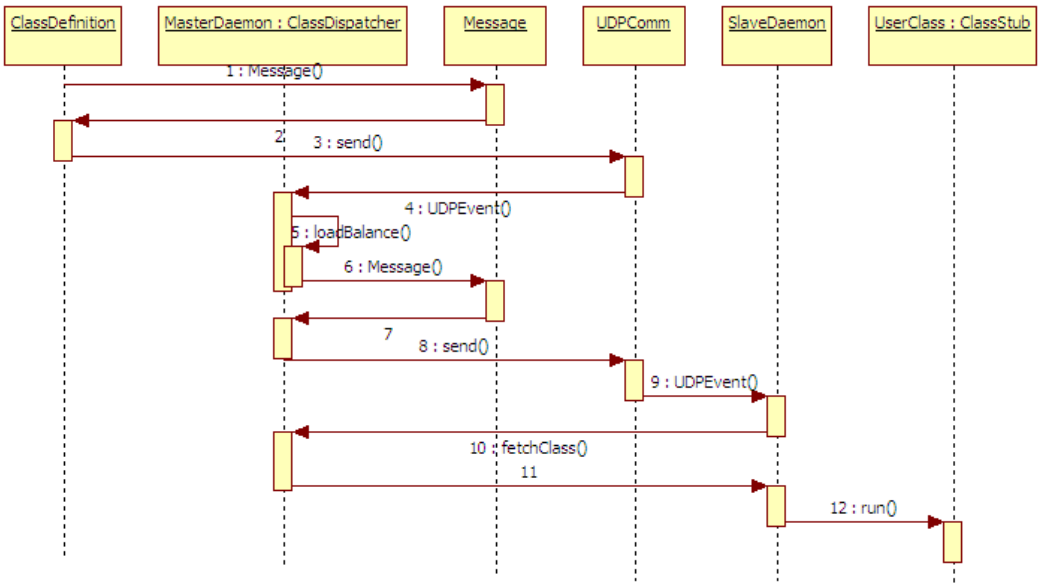


Figura 4.11: Se muestra la secuencia que sucede para ejecutar una clase de usuario.

Diagrama de Secuencia de la Ejecución de una Clase de Usuario

Para ejecutar una clase de usuario, sucede la siguiente secuencia (Figura 4.11):

1. El usuario crea su definición de clases, indicando el nombre de la clase principal y secundaria, y el número de instancias de cada una de ellas. Se genera una instancia de `Message` con el mensaje `CLASS_DEFINITION`.

2. El mensaje se construye y el flujo de ejecución regresa a `ClassDefinition`.

3. `ClassDefinition` emplea a `UDPComm` para enviar el mensaje recién creado a `MasterDaemon`

4. El `UDPComm` del `MasterDaemon` recibe el mensaje, y genera un `UDPEvent()` para notificarle del mensaje al `MasterDaemon`.

5. El `MasterDaemon` invoca al método `loadBalance()` a sí mismo para balancear la carga de clases en la Plataforma.

6. El `MasterDaemon` genera una instancia de `Message` con el mensaje `FETCH_CLASS`.

7. El mensaje se construye y el flujo de ejecución regresa a `MasterDaemon`.

8. `MasterDaemon` emplea a `UDPComm` para enviar el mensaje recién creado a `SlaveDaemon`.

9. El `UDPComm` del `SlaveDaemon` recibe el mensaje, y genera un `UDPEvent()` para notificarle del mensaje a `SlaveDaemon`.

10. El `SlaveDaemon` invoca `fetchClass()` sobre `MasterDaemon` para obtener la clase que se le indicó.

11. El `MasterDaemon` retorna una instancia de la clase a `SlaveDaemon`.

12. El `SlaveDaemon` invoca el método `run()` de la clase del usuario.

Diagrama de Secuencia de la Memoria Compartida

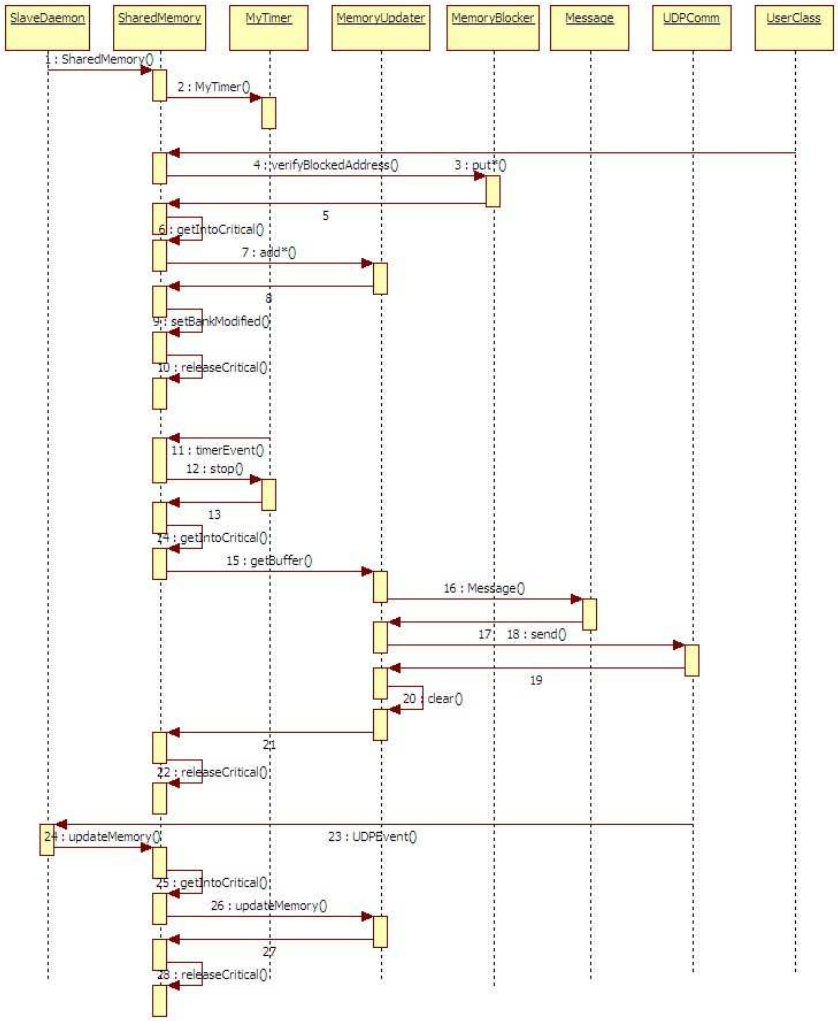


Figura 4.12: Se muestra la secuencia que sucede en la memoria compartida.

La siguiente lista es la secuencia que sucede en la memoria compartida (Figura 4.12):

1. Al momento de que se crea el `SlaveDaemon`, se crea una instancia de `SharedMemory`. Todas las instancias de las clases de usuario que se generen en este nodo, compartirán esta instancia de `SharedMemory`.

2. `SharedMemory` crea una instancia de `MyTimer`.

3. En cualquier momento, la clase del usuario invoca cualquiera de los métodos `put*()` sobre `SharedMemory`.

4. `SharedMemory` verifica que la dirección solicitada no esté bloqueada. De estarlo, este hilo se bloqueará hasta que la tarea propietaria del bloqueo lo libere.

5. La llamada a `verifyBlockedAddress()` regresa el control a `SharedMemory`.

6. `SharedMemory` invoca sobre sí misma el método `getIntoCritical()`, para asegurar que ningún otro hilo acceda a la memoria en ese momento.

7. `SharedMemory` invoca cualquiera de los métodos `add*()` sobre `MemoryUpdater`, quien registrará todos los cambios que se hayan realizado a la memoria por la clase del usuario.

8. La llamada sobre `add*()` regresa el control a `SharedMemory`.

9. `SharedMemory` invoca sobre sí a `setBankModified()`, con lo que se inicia el `MyTimer`, quien al vencerse iniciará el proceso de actualización al resto de la Plataforma. Esto sucede en el paso 11.

10. `SharedMemory` libera la región crítica invocando sobre sí misma el método `releaseCritical()`.

11. Al vencerse el tiempo de `MyTimer` se genera un evento `timerEvent()` que es comunicado a `SharedMemory`.

12. `SharedMemory` invoca `stop()` sobre `MyTimer` para detenerlo hasta la próxima escritura.

13. La llamada a `stop()` regresa el control a `SharedMemory`.

14. `SharedMemory` invocará sobre sí el método `getIntoCritical()`, para asegurar que ningún otro hilo acceda a la memoria en ese momento.

15. `SharedMemory` solicita a `MemoryUpdater` todos los cambios que haya registrado desde que se arrancó el conteo de su tiempo al momento mediante el método `getBuffer()`.

16. `MemoryUpdater` genera un buffer con todos los cambios y construye un `Message` con el mensaje `MEMORY_MESSAGE`.

17. El mensaje es creado y el flujo de ejecución regresa a `MemoryUpdater`.
18. `MemoryUpdater` emplea su `UDPComm` para enviar el mensaje recién creado
19. El mensaje es enviado por el `UDPComm` y el flujo de ejecución regresa a `MemoryUpdater.`
20. `MemoryUpdater` invoca sobre sí mismo el método `clear()`, para borrar los cambios que tenía registrados.
21. El flujo de control regresa a `SharedMemory`.
22. `SharedMemory` invoca sobre sí mismo el método `releaseCritical()` para permitir a otros hilos acceder a la memoria.
23. En otro momento, en el `UDPComm` de otro nodo, se recibe el mensaje que se envió. Genera un evento `UDPEvent()` que es comunicado a `SlaveDaemon`.
24. `SlaveDaemon` invoca el método `updateMemory()` sobre `SharedMemory`, para indicarle que recibió por la red, cambios a aplicar a la memoria.
25. `SharedMemory` invoca sobre sí mismo `getIntoCritical()` para evitar que otros hilos accedan a la memoria.
26. Una vez estando en la región crítica, `SharedMemory` invoca sobre `MemoryUpdater` el método `updateMemory()`, para comunicarle los cambios que deben realizarse.
27. Los cambios se aplican sobre la memoria, y el flujo de control regresa a `SharedMemory`.
28. `SharedMemory` invoca sobre sí misma el método `releaseCritical()`, para permitir a otros hilos acceder a la memoria.

4.2. Arquitectura de la Plataforma

En secciones anteriores se establecieron algunas de las clases y las interacciones que éstas tienen. Se han planteado también, los servicios y/o funcionalidades que se pondrán a disposición del usuario. Sin embargo, aún se carece de una descripción global del sistema, que denote *cómo interactuarán* las instancias de las clases entre los nodos de la Plataforma entera, y aún dentro de un solo nodo. Del mismo modo, es necesario ubicar y clasificar los servicios que se ofrecen al usuario dentro de un marco bien definido. Para hacerlo, se planteará la topología lógica y la arquitectura de la Plataforma.

4.2.1. Topología Lógica

Comenzaremos estableciendo la topología lógica de la arquitectura. Esto ayudará a aclarar la forma en que un conjunto de computadoras conectadas por una red TCP/IP se convierte en nuestra Plataforma de desarrollo y ejecución.

Para explicar la topología lógica de la Plataforma, primero se debe apreciar la topología física de la conexión de las computadoras que conforman la Plataforma.

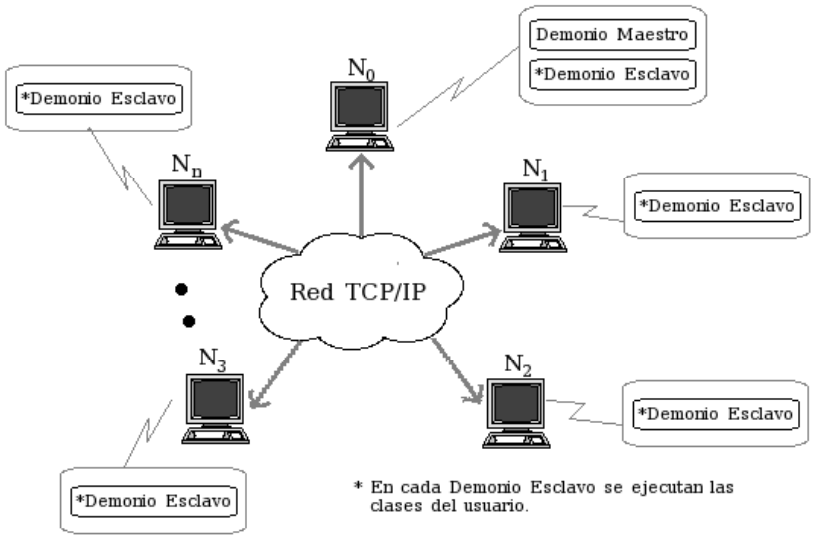


Figura 4.13: La topología física tiene como base una red TCP/IP.

Dichas computadoras se deben encontrar interconectadas mediante una red TCP/IP, ya sea en una red o mediante un switch de alta velocidad. Lo importante, es que cada una de las computadoras tenga una dirección IP dentro del mismo rango de direcciones, es decir, se encuentren dentro de la misma subred. Esto es un requisito para que la Plataforma pueda funcionar. Los medios físicos para implementar la red TCP/IP no alteran (excepto en desempeño) la ejecución de la Plataforma. La plataforma fue desarrollada en un sistema Linux, pero debería funcionar sin ninguna alteración con cualquier otro sistema operativo que soporte Java en su versión 1.5 o posterior.

Una vez interconectadas las computadoras mediante la red TCP/IP, se debe elegir una de ellas como nodo maestro. Para el resto de las computadoras no habrá diferencia. En el nodo maestro se debe arrancar el Demonio Maestro, y opcionalmente, se puede arrancar una o más instancias del Demonio Esclavo. Solamente debe existir una instancia del Demonio Maestro en toda la Plataforma. En el resto de las computadoras de la Plataforma se deben levantar el resto de los Demonios Esclavos, uno o más por nodo.

Es importante que el primero en arrancar sea el Demonio Maestro, ya que éste lleva registro de los demonios que se van levantando. Si se levantase un Demonio Esclavo antes de haber levantado el Demonio Maestro, el Demonio Esclavo será omitido en la Plataforma.

Al levantar el Demonio Maestro, éste abre un socket en multicast. Un socket en multicast, es un socket UDP con la capacidad de unirse a *grupos* de sockets que también se encuentren en multicast.

Un grupo en multicast se especifica mediante una dirección IP de clase D y un número de puerto UDP estándar. Las direcciones de clase D se encuentran en el rango

224.0.0.0 al 239.255.255.255, inclusive. La dirección 224.0.0.0 está reservada y no debe usarse.

En un socket en multicast, cuando alguno de los participantes del grupo envía algún paquete, todo el grupo, incluyendo al emisor, reciben el paquete que se envió. Esto es importante y es empleado intensivamente en la Plataforma, especialmente para la Memoria Compartida.

Al irse levantando el resto de los Demonios, se irán uniendo a este grupo de multicast. Mediante esta construcción, usando sockets en multicast y un único grupo de participantes, se conforma la Plataforma de ejecución.

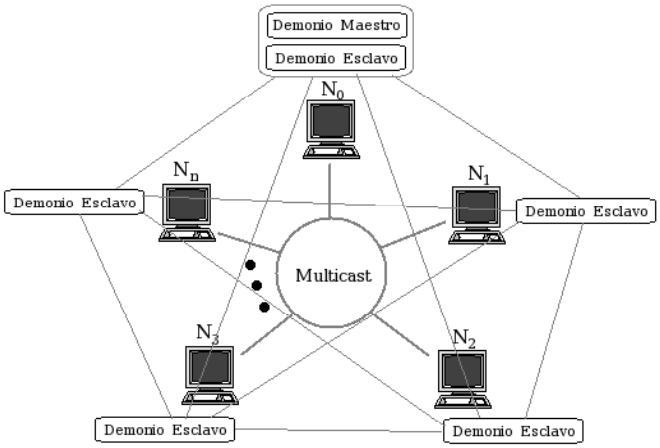


Figura 4.14: Esta topología mantiene una conexión de todos con todos totalmente transparente.

Note que en este nivel de abstracción, dentro de la topología lógica de la Plataforma, el hablar de nodos físicos carece de sentido. En su lugar, se considera como si cada Demonio, Maestro o Esclavo, fueran nodos particulares e independientes, totalmente interconectados entre ellos. En este nivel, no hay diferencia entre dos Demonios si están en la misma computadora o en diferentes computadoras. A partir de ahora, al hablar de nodos nos estaremos refiriendo a una instancia de un Demonio Esclavo, a menos de indicar lo contrario.

Se ha mencionado que se puede levantar más de una instancia de un Demonio Esclavo dentro de un mismo nodo, porque ya se explicó que para efectos de la topología lógica, cada uno de ellos no serán más que un par de nodos lógicos más que se agregan a la Plataforma. Con esto, en una red de 10 computadoras, podría levantarse la Plataforma con 10 Demonios, con 100 o con más, dando a nivel de aplicación la apariencia de una Plataforma con dicho número de nodos. Sin embargo, a pesar de parecer deseable hacer lo anterior, tiene sus desventajas, siendo la más plausible el consumo de los

recursos de cada computadora.

Al levantar un Demonio, se debe crear toda una instancia de la máquina virtual de Java. Al levantar un segundo Demonio, no se vuelve a levantar en su totalidad la JVM, pero si una fracción de ella. Por cada Demonio subsecuente que se levante, se levantará también una copia parcial de la JVM. Esto consume recursos, principalmente de memoria.

Recuerde que cada uno de los Demonios Esclavos de la Plataforma, es capaz de ejecutar N tareas (clases) del usuario. Continuando con nuestra red de 10 computadoras, si se requiriese ejecutar 100 tareas, es preferible levantar una Plataforma con 10 Demonios cada uno de ellos atendiendo 10 tareas, a levantar 100 Demonios cada uno de ellos atendiendo 1 tarea.

Incluso en cuestiones de espacio, es preferible levantar un Demonio por computadora, debido a la memoria compartida. Cada Demonio posee su propia instancia de la Memoria Compartida. Si en la Memoria Compartida es necesario realizar una paginación de 100MB, de levantar 10 Demonios en la misma computadora, se estará realizando la paginación de 10×100MB. Siendo las operaciones de E/S las más lentas, esto repercute no sólo en espacio, sino también en el desempeño general de los Demonios de esta computadora.

Es recomendable levantar un Demonio por Computadora. Sin embargo, también es recomendable levantar un Demonio Esclavo en la misma computadora donde se levantó el Demonio Maestro. Esto se debe a que el Demonio Maestro no posee Memoria Compartida, y no realiza el procesamiento que deben realizar los Demonios Esclavos (cuya carga depende de las clases del usuario). Las tareas del Demonio Maestro son relativamente sencillas, y después de haber hecho el balance de carga y la distribución de las clases, se reduce prácticamente a la consulta periódica de nodos (poleo) y la sincronización. Si no se emplease el mismo nodo para levantar un Demonio Esclavo, se puede estar desperdiciando recursos.

4.2.2. Arquitectura

El planear y modelar la Plataforma a través de una arquitectura trae beneficios sustanciables. El primero de ellos es que es posible cambiar la implementación de los servicios de las capas que subyacen sin necesidad de que se afecte el resto de los servicios. Por ejemplo, es posible cambiar la implementación de la Capa del Canal de Comunicaciones que en su mayor parte se encuentra con base en un socket UDP en multicast, y se puede implementar con un socket TCP.

El siguiente beneficio es que se pueden agregar nuevos servicios en cualquiera de las capas, enriqueciéndola y extendiendo así la plataforma.

En la Figura 4.15 se presenta la arquitectura para los Demonios Maestro y Esclavo. Las capas difieren en los Demonios ya que tanto los servicios que ofrecen como las tareas que cumplen no son iguales.

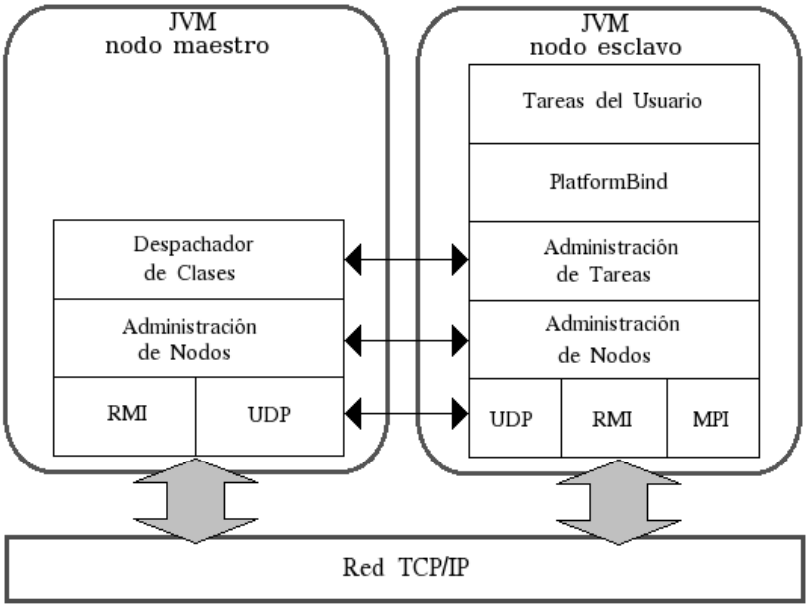


Figura 4.15: Presenta las capas de los Demonios Maestro y Esclavo.

Demonio Maestro

En esta pila tenemos 3 capas de servicios:

- Capa de Canal de Comunicación
- Capa de Administración de Nodos
- Capa de Despachador de Clases

En la capa de Canal de Comunicación se encuentran los servicios a través de los cuales el Demonio Maestro se comunica con el resto de la Plataforma. Esta capa está subdividida en dos partes, UDP y RMI. En la parte de UDP se ofrece el servicio de envío de mensajes y una estuctura cómoda para enviar mensajes, que son ampliamente usados por toda la Plataforma. En la parte de RMI se ofrece el servicio de distribución de clases, las cuales se exponen a través de una interfaz remota.

En la capa de Administración de Nodos se registran todos los Demonios (nodos) que van iniciando en la Plataforma. Además, también se hace el poleo de los mismos para ver su disponibilidad. Se emplean los servicios de la Capa de Canal de Comunicación para poder tener contacto con el resto de los nodos.

En la capa de Despachador de Clases, se hace uso de los servicios de la capa de Administración de Nodos para conocer todos los nodos que conforman en ese momento la Plataforma. Además, desde aquí se atiende la Definición de Clases del usuario para poder hacer el balance de carga. Empleando la información de los nodos, se les puede indicar a cada uno de ellos que obtenga las clases que se les indique ejecutar.

Demonio Esclavo

En esta pila tenemos 5 capas de servicios:

- Capa de Canal de Comunicación
- Capa de Administración de Nodos
- Capa de Administración de Tareas
- PlatformBind
- Tareas del Usuario

En la capa de Canal de Comunicación se encuentran los servicios a través de los cuales el Demonio Esclavo se comunica con el resto de los Demonios de la Plataforma. Esta capa está subdividida en tres partes, UDP, RMI y MPI. En la parte de UDP se ofrecen el servicio de envío de mensajes y una estructura para el envío de mensajes. En la parte de RMI se encuentra la facilidad para poder obtener las clases del Demonio Maestro. En la parte de MPI se encuentran las funciones de MPI propiamente codificadas, resolviendo los problemas de concurrencia que posee la librería.

Existe una capa de Administración de Nodos, en la que basicamente se genera el nombre del nodo, y también puede solicitar el nombre del resto de los nodos.

En la capa de Administración de Tareas, se gestionan las tareas que se reciben desde el Demonio Maestro. De hecho, se recibe del Demonio Maestro la orden para solicitar una clase. El Demonio Maestro indica qué clase se debe obtener, incluyendo su identificador de tarea. Toda esta comunicación se realiza usando los servicios que ofrece la Capa del Canal de Comunicación.

Posteriormente viene una capa especial. Es la capa del PlatformBind. Esta capa es el intermediario entre la Plataforma y las clases del usuario. A través de esta capa se exponen todos los servicios que ofrece la Plataforma a las tareas del usuario. En esta capa se ofrecen los servicios de Comunicación entre tareas usando mensajes por el socket UDP, paso de mensajes usando MPI, administración de hilos y el servicio de Memoria Compartida.

Finalmente, vienen las tareas del usuario. En esta capa se realiza la ejecución de las tareas del usuario. A pesar de que las tareas se gestionan en la capa número 3, no es sino en esta capa que se realiza su ejecución. Aunque parezca mera convención la distinción entre la capa 3 y 5, ello ayuda a establecer correctamente los papeles de cada una de las capas y los servicios que se ofrecen en ellas, además de fortalecer el concepto de extensibilidad, ya que es posible modificar la manera en que se ejecutan las tareas alterando la capa de Tareas del Usuario para colocarlas, por ejemplo, en un pool de tareas.

Capítulo 5

Implementación de la Plataforma

En este capítulo se presentan algunos detalles de la implementación de la Plataforma. Se mencionan las razones por las cuales se decidieron ciertas formas de implementar algunos mecanismos, y en la mayoría de los casos se da una explicación sobre cómo funcionan esos mecanismos. La mejor fuente definitiva para consultar la implementación de la Plataforma, es el código fuente, que se encuentra completamente documentado.

5.1. Implementación del Canal de Comunicaciones

5.1.1. Nombrado de Nodos

La plataforma se conforma por un conjunto de computadoras, en las cuales se estará ejecutando el Demonio Esclavo, el Demonio Maestro, o ambos. Al iniciar el Demonio en una computadora determinada, el Demonio se encarga de anunciar al resto de la Plataforma que el nodo se está agregando. Recuerde que para la Plataforma, un nodo es la instancia de un Demonio Esclavo (vea Figura 4.15).

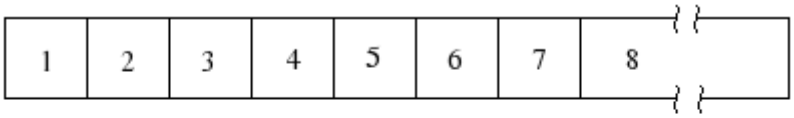
Desde el punto de vista del usuario, los nombres de los nodos no es de relevancia, ya que el usuario no sabe en qué nodo se ejecutará alguna de sus clases.

Sin embargo, para el envío de mensajes, es necesario conocer el destinatario, y en ocasiones, el origen. Para ello, cada uno de los nodos debe tener, para efectos de la plataforma, un nombre único. Inicialmente, al arrancar el Demonio, éste intentará tomar su nombre directamente del nombre de la computadora, para evitar duplicidades. Estas duplicidades en ocasiones no pueden salvaguardarse, y puede deberse a que diversas causas, siendo algunas de ellas, a que no estén bien configurados los nombres de las computadoras o bien que se levante más de un Demonio en la misma computadora.

Para resolver este problema, al momento de iniciar algún Demonio, el Demonio intenta obtener el nombre de la computadora. Si el nombre que obtiene es *localhost*, automáticamente el Demonio renombra (solo a nivel de la aplicación) el nodo, asignándole como nombre la cadena *NODE_XYZ*, donde *XYZ* es un número aleatorio.

5.1.2. Estructura del mensaje

Se pretende emplear un sólo punto de comunicaciones, que sea empleado por todas las funcionalidades de la plataforma: la memoria, la administración de la plataforma y los mensajes de usuario. Para ello, es necesario que el mensaje que se envíe, ofrezca una estructura que permita distinguir el origen, el destino y la finalidad del mensaje.



1. [0] Source Node
2. [50] Destination Node
3. [100] Source Task
4. [104] Destination Task
5. [108] Platform Tag
6. [109] Message
7. [113] Message Length
8. [117] Buffer

Figura 5.1: Es la estructura de un mensaje. En la parte de abajo se indica el orden y el índice de comienzo de cada campo.

En la plataforma, los mensajes se representan por la clase `Message`, quien realiza el envío y recepción de bufferes con la siguiente estructura, tal como se ve en la Figura 5.1:

- *Source Node*, Nodo Origen (50 bytes): Almacena el nombre del nodo origen en texto claro.
- *Destination Node*, Nodo Destino (50 bytes): Almacena el nombre del nodo destino en texto claro.
- *Source Task*, Tarea Origen (4 bytes, un entero): Almacena el identificador de la tarea que envía el mensaje. Tiene el valor de 0 si quien genera el mensaje es algún elemento de la plataforma.
- *Destination Task*, Tarea Destino (4 bytes, un entero): Almacena el identificador de la tarea que debe recibir el mensaje. Tiene el valor de 0 si quien debe recibir el mensaje es algún elemento de la plataforma.
- *Platform Tag*,Bandera Plataforma: (1 byte): Tendrá un valor de 1 si el mensaje es exclusivamente para la plataforma, y un valor de 0 si el mensaje es para el

usuario. Si el mensaje es para la plataforma, ésta se encargará de que el mensaje nunca le llege al usuario.

- *Message*, Mensaje (4 bytes, un entero): Es un entero que identifica el contenido del mensaje. Si el mensaje es del usuario, éste puede emplear este campo como desee. Si el mensaje es de la plataforma, puede tener alguno de los siguientes valores:

    1. DROP_NET: Le indica a todos los nodos de la plataforma que terminen la aplicación.
    2. ADD_NODE: Indica que algún nodo se ha registrado en la red de la plataforma.
    3. LIST_NODES: Es un mensaje que genera el demonio maestro para avisarle a cada nodo la lista de nodos disponibles y registrados en la plataforma.
    4. PING: Es un mensaje que genera el demonio maestro para solicitar a los nodos que le avisen si aún están activos.
    5. PONG: Es un mensaje que generan los demonios esclavos para confirmar que aún están activos.
    6. IP_MASTER: Es un mensaje que genera el demonio maestro para informar a los nodos de la dirección IP del nodo maestro.
    7. FETCH_CLASS: Es un mensaje para informarle a los demonios esclavos que obtengan la clase que deben ejecutar del servidor.
    8. CLASS_DEFINITION: Es un mensaje que genera el usuario para informarle al demonio maestro sobre la definición de las clases que empleará.
    9. RENAME_NODE: Es un mensaje que se genera cuando un nodo cambia de nombre.
    10. MEMORY_MESSAGE: Es un mensaje que contiene los cambios que se le han hecho a la memoria compartida y que deben replicarse al resto de los nodos de la red.
    11. BLOCKER_MESSAGE: Es un mensaje que contiene las direcciones de la memoria compartida que se han bloqueado.
    12. RELEASER_MESSAGE: Es un mensaje que contiene las direcciones de la memoria compartida que se han liberado.
    13. BARRIER_MESSAGE: Es un mensaje para indicar que un nodo se ha sincronizado.
    14. CLEAR_BARRIER: Es un mensaje para indicar que los nodos se liberen para continuar su ejecución.

- *Message Length*,Longitud del mensaje (4 bytes, un entero): Es un entero que contiene la longitud total del mensaje, incluyendo el encabezado más el buffer de datos.

- *Buffer*, Buffer de datos (n bytes): Es un espacio libre para datos.

5.1.3. Sockets para Comunicación

Para realizar la comunicación a través de la plataforma, se emplea un socket UDP en multicast. El uso del socket UDP implica que la entrega de paquetes no se garantiza, así como tampoco el orden. Sin embargo, la plataforma está orientada a un cluster o una red de computadoras LAN. Esto supone que las computadoras se encuentran en el mismo segmento de subred, y de hecho se deben encontrar de esta manera, de lo contrario, el socket UDP en multicast no funcionaría. Si las computadoras no se encuentran separadas por un enrutador, la probabilidad de perder un paquete o de recibirlos en desorden, es practicamente despreciable.

El empleo de sockets UDP trae como ventaja la simplicidad y la rapidez del protocolo, además de que en TCP no existe una implementación en multicast.

Si se hubieran empleado sockets TCP, en una red con N nodos, cada nodo debería crear $N-1$ sockets para tener comunicación con cada uno de los nodos restantes de la plataforma. Usando sockets UDP podría suceder algo similar, pero se tiene a disposición los sockets UDP en multicast.

Las direcciones IP de clase D, que se encuentran entre `224.0.0.0` y `239.255.255.255` inclusive, se emplean para multicast. Independientemente de la dirección asignada a cualquier interfaz de red, se puede emplear alguna dirección en el rango anteriormente indicado en alguna aplicación. La dirección en el rango indicado, más un número de puerto estandar UDP, y se pueden recibir y enviar mensajes en multicast.

En multicast, más de un proceso, o nodo, pueden emplear la misma dirección IP. Con ello, cualquier envío que se realice a tal dirección en multicast, será recibido por todos aquellos que estén empleando esa dirección.

5.2. Memoria Compartida

La plataforma ofrece una construcción que hace las veces de memoria compartida. Con ella, es posible ofrecerle al desarrollador una región de memoria que se encuentra compartida entre todos los nodos de la Plataforma. La región de memoria que se ofrece es vista por todos los nodos de la plataforma, mediante réplicas. Cada nodo posee una réplica completa de la memoria compartida.

La Memoria Compartida esta diseñada para realizar paginación a disco mediante páginas de tamaño pequeño. Con esto, se le ofrece al desarrollador una memoria de tamaño teóricamente limitado por la capacidad de disco.

El tamaño de la memoria compartida depende de la capacidad de disco y la capacidad del sistema operativo para el manejo de archivos, ya que de toda la memoria compartida, solamente una fracción de ésta se encuentra cargada en memoria. La administración de la memoria compartida se realiza a través de la paginación de bancos de memoria a disco. Los bancos tienen un tamaño original (y configurable por el usuario) de 4096 bytes.

Todas las instancias de las clases de usuario que se encuentren en el mismo nodo comparten la misma instancia de memoria compartida. Por ello, se dice que la memoria

compartida ofrece concurrencia en dos niveles, primero a nivel de nodo y luego a nivel de plataforma.

El acceso a la memoria compartida se realiza a través de métodos `put*()` y `get*()`, que permiten leer y escribir tipos primitivos a la memoria (byte, entero, float, double y arreglos de bytes). Todas las direcciones de la memoria compartida contienen un byte, por lo que es responsabilidad del usuario realizar los desplazamientos necesarios para poder almacenar tipos primitivos que ocupen más de un byte. La Plataforma ofrece constantes con los tamaños de cada uno de los tipos primitivos.

Al igual que con la implementación de una memoria compartida en hardware, esta implementación también ofrece el bloqueo de direcciones específicas. Los bloqueos se registran junto con el identificador de la tarea que realiza el bloqueo. Con esto, se puede garantizar a la tarea que realiza el bloqueo, el acceso exclusivo a dicha dirección, por toda la Plataforma. Cuando cualquier otra tarea en cualquier nodo intente leer o escribir a dicha dirección de memoria, se bloqueará hasta que la primer tarea libere el bloqueo.

Esta funcionalidad se puede emplear como un mecanismo de sincronización básico. Sin embargo, hay que revisar ciertas consideraciones sobre latencia, ya que al realizar un bloqueo, tomará cierto tiempo antes de actualizarse a toda la Plataforma.

5.2.1. Actualización y Réplica

Cada vez que un proceso (una tarea) realiza una actualización en la instancia de la memoria compartida que posee en su nodo(mediante una escritura usando alguno de los métodos `put*()`), esta actualización debe comunicarse a las demás instancias de la memoria compartida para que el resto de las tareas tengan conocimiento de los nuevos valores.

El notificar al resto de los nodos *cada una* de las actualizaciones que se realicen a la memoria conlleva un alto número de mensajes de red que deben enviarse. Eso provoca congestiones en la red, consumo de recursos y tiempo extra dedicado a atender los mensajes.

Para sobresalvar el problema anterior, las actualizaciones se realizan en forma tardía. Esto parte de la idea de que una escritura regularmente va a acompañada por otras escrituras, y raramente se realizan en forma aislada.

Las actualizaciones tardías descansan sobre el uso de un temporizador (de tiempo configurable por el usuario). Cuando el usuario realiza una actualización a la memoria, el temporizador inicia el conteo de tiempo. Al vencerse el tiempo del temporizador, éste colecta todas las actualizaciones que se realizaron a la memoria y las comunica al resto de la Plataforma en conjunto. Con ello se reduce la sobrecarga de red y el consumo de recursos. Sin embargo, puede conllevar un rezago en el tiempo de respuesta de la aplicación del usuario, al introducir un tiempo extra de espera. Esto se puede manejar alterando el tiempo del temporizador: cuanto menor sea el tiempo, menor será la espera por los procesos, pero introduce un mayor número de mensajes que se deben enviar por la red.

5.2.2. Bloqueo de Direcciones

La memoria compartida permite el bloqueo de direcciones de memoria. Cada vez que se bloquea una dirección de memoria, el bloqueo se registra con el identificador de la tarea que realizó el bloqueo. A partir de ese momento, solamente esa tarea puede acceder a esa dirección, ya sea para lectura o escritura. Cualquier otro proceso que realice un acceso a esa dirección de memoria, ya sea para lectura o escritura, se bloqueará hasta que la dirección se libere por el proceso propietario del bloqueo.

Los bloqueos de memoria se comunican al resto de la plataforma a través de un mensaje. A diferencia de los mensajes de actualización de contenido, los mensajes para indicar los bloqueos se realizan al momento que el proceso de usuario lo realiza. Si al momento que una tarea realiza un bloqueo o desbloqueo, la instancia de la memoria compartida se encarga de liberar todas las actualizaciones (de existir) al resto de la Plataforma. Con esto se asegura que el contenido sea el correcto antes de realizar el bloqueo o desbloqueo.

Regularmente, las tareas emplearán los bloqueos de memoria como una forma de asegurar la concurrencia, y como una forma de mantener sincronización entre ellas. Es por ello que la notificación de bloqueos o liberación de direcciones de memoria se realiza al momento que la tareas lo realice, no de forma tardía como la actualización y réplica de contenido.

5.3. Sincronización de la Plataforma

Las diversas plataformas de hardware paralelas ofrecen mecanismos de sincronización. Según el tipo de plataforma, los procesadores pueden estar sincronizados mediante una unidad de control a nivel de instrucción, o bien pueden sincronizarse a intervalos regulares en ciertos puntos de ejecución.

Los puntos de sincronización son una construcción común en las herramientas de software para el desarrollo de aplicaciones distribuídas y paralelas. A nivel de software, un punto de sincronización consiste en lograr que los diferentes procesos en ejecución alcancen un punto en la ejecución en el cual uno a uno de los procesos se detendrán hasta que todos ellos lleguen ahí, momento en que su ejecución continúa.

En el diseño de esta plataforma, se ha incluído una construcción para crear puntos de sincronización, bajo el nombre de *Barriers*. Estos puntos de sincronización ocurren en dos etapas. En la primer etapa, se busca la sincronización a nivel de nodo. En cada nodo puede ejecutarse más de un proceso de usuario, por lo que el nodo llegará al punto de sincronización cuando todos los procesos que está ejecutando se sincronicen. En la segunta etapa, toda la plataforma logrará la sincronización cuando todos los nodos hayan completado la primer etapa de sincronización. El nodo maestro será el responsable de recibir las notificaciones de cada nodo sobre su sincronización. Tras haber recibido las notificaciones de sincronización de todos los nodos, el nodo maestro enviará un mensaje de liberación a todos los nodos y la ejecución en cada uno de ellos continuará.

Para poder lograr la sincronización, se emplearon dos clases de las utilerías de concurrencia de Java, el `CyclicBarrier` y el `CountDownLatch`.

El `CyclicBarrier` permite a un conjunto de hilos la espera a que cada uno de ellos alcance un punto en común, y entonces son liberados y su ejecución continúa. El `CyclicBarrier` es una construcción útil en programas que involucran un número fijo de hilos que deben ocasionalmente esperar al resto. Se dice que es cíclico por que puede ser reusado después de que los hilos son liberados. Al momento de la construcción, el `CyclicBarrier` soporta recibir una tarea (`Runnable`), que será ejecutada cada vez que todos los hilos alcancen el punto en común, y antes de que los hilos sean liberados.

El `CountDownLatch` es una construcción que permite que uno o más hilos se bloqueen hasta que un conjunto de operaciones que se realizan en otros hilos, se completen. El `CountDownLatch` se inicializa con un contador. El método `await()` bloquea el hilo invocador hasta que el conteo llega a cero, dadas las invocaciones del método `countDown()`. Cuando el contador llega a cero, todos los hilos bloqueados (por haber invocado `await()`) se liberan y cualquier invocación subsecuente de `await()` regresa inmediatamente. Esto es un fenómeno único, ya que el contador no puede ser reinicializado.

Un empleo de `CountDownLatch` muy versátil es cuando se inicializa con el valor de 1, ya que sirve como una compuerta: todos los hilos que invoquen `await()` se bloquearán hasta que la compuerta se libera por el primer hilo que invoque `countDown()`. Una instancia de `CountDownLatch` que se inicialice con N puede usarse para bloquear un hilo hasta que N hilos han completado alguna operación, o una operación se ha completado N veces. Una propiedad útil de `CountDownLatch` es que no requiere que los hilos que invoquen `countDown()` se bloqueen hasta que el contador llegue a cero antes de proceder, ya que simplemente previene que cualquier hilo continúe hasta que todos los hilos (en conjunto) puedan continuar.

En el diseño de la plataforma se incluyeron ambas clases, `CyclicBarrier` y `CountDownLatch` para lograr la sincronización. Se mencionaba que la sincronización se realiza en dos etapas: a nivel nodo y a nivel plataforma. Para lograr la sincronización a nivel nodo, se emplea un `CyclicBarrier`, que es inicializado con una tarea (`Runnable`) responsable de informar al nodo maestro de la sincronización del nodo. Tal tarea no se iniciará sino hasta que todos los procesos de usuario que se estén ejecutando en el nodo alcancen el punto común. Una vez en el punto común, se genera y se envía el mensaje. Enviado el mensaje, los procesos continuarán, pero se volverán a bloquear inmediatamente bajo el efecto de un `CountDownLatch`.

El `CountDownLatch` se emplea para detener todos los procesos del usuario, hasta que se reciba el mensaje del nodo maestro acerca de la sincronización. El `CountDownLatch` se inicializa con el valor de 1, por lo que realmente se está empleando como una compuerta para detener todos los procesos de usuario hasta que se reciba el mensaje por parte del nodo maestro.

Por su parte, el nodo maestro lleva un registro del número de nodos que se han reportado como sincronizados. Cuando el número de nodos reportados equivale al número de nodos registrados en la plataforma, el nodo maestro envía el mensaje de liberación, y

cada nodo invoca el método `CountDownLatch.countDown()`, para liberar la *compuerta* y todos los hilos puedan continuar.

Capítulo 6

Ejemplos de uso de la Plataforma

En este capítulo se presentan las funcionalidades de la Plataforma, y la forma en que éstas son expuestas al usuario. Tras haber mostrado las funcionalidades y la forma de accederlas, se presentan dos ejemplos que se ejecutan sobre la Plataforma.

6.1. Funcionalidades de la Plataforma

Se ha mencionado que la Plataforma ofrece varias funcionalidades que el desarrollador puede emplear para la creación de sus aplicaciones. Tales funcionalidades están implementadas a diversos niveles dentro de la arquitectura de los demonios tanto maestro como esclavos. Es necesario concentrar estas funcionalidades en un único punto y exponerlas al desarrollador. Esto se consigue con la clase `PlatformBind` y la interfaz `ClassStub`.

6.1.1. Exposición de las funcionalidades a través de la clase PlatformBind

Se ha dicho que las funcionalidades de la Plataforma se encuentran implementadas en diferentes capas de la arquitectura de los demonios. Es posible clasificar estas funcionalidades en los siguientes grupos:

- Envío y recepción de mensajes
- Memoria compartida
- Nombres de Tareas
- Pool de hilos
- Funciones de MPI
- Sincronización

Las anteriores se encuentran implementadas en diversas clases. Sin embargo, tanto por transparencia para el usuario como por estabilidad de la Plataforma, el usuario no debe tener acceso directamente a dichas clases. Las funcionalidades están concentradas en un punto único, ocultando al usuario dónde se encuentran realmente implementadas. Este punto único es la clase `PlatformBind`.

A través de la clase `PlatformBind` las funcionalidades están expuestas al usuario en forma de métodos. De acuerdo a la clasificación anterior se tienen los siguientes métodos:

- Envío y recepción de mensajes:
  - `public void send(Message message)`: método para enviar un mensaje.
  - `public Message receive()`: método para recibir un mensaje.
  - `public int getQueueSize()`: método para obtener el número de mensajes en la cola en espera de recibirse por la tarea del usuario.
- Memoria compartida
  - `public void putByte(int offset, byte b)`: agrega un byte a la memoria compartida en la dirección especificada.
  - `void putInt(int offset,int i)`: agrega un entero a la memoria compartida en la dirección especificada.
  - `void putFloat(int offset,float f)`: agrega un flotante a la memoria compartida en la dirección especificada.
  - `void putDouble(int offset, double d)`: agrega un doble a la memoria compartida en la dirección especificada.
  - `void putBytes(int offset,byte[] b)`: agrega un arreglo de bytes a la memoria compartida en la dirección especificada.
  - `public byte getByte(int offset)`: obtiene un byte de la memoria compartida de la dirección especificada.
  - `public int getInt(int offset)`: obtiene un entero de la memoria compartida de la dirección especificada.
  - `public float getFloat(int offset)`: obtiene un flotante de la memoria compartida de la dirección especificada.
  - `public double getDouble(int offset)`: obtiene un doble de la memoria compartida de la dirección especificada.
  - `public byte[] getBytes(int offset, int length)`: obtiene un arreglo de bytes de la memoria compartida de la dirección especificada.
  - `public void lock(int offset)`: bloquea la dirección de memoria para acceso exclusivo de la tarea que lo invoca.
  - `public void unlock(int offset)`: desbloquea la dirección de memoria para que otras tareas accedan a la dirección de memoria.

- Nombres de Tareas:
  - `public int getTask()`: obtiene el identificador de tarea dentro de la Plataforma para la tarea que lo invoca.
  - `public int getTotalTasks()`: obtiene el número total de tareas que el usuario solicitó instanciar en la Plataforma.
- Pool de hilos
  - `void addTask(Runnable task)`: agrega una tarea al pool de hilos del `PlatformBind`.
- Funciones de MPI
  - `public synchronized void MPI_Initialize(String[] args)`: es la implementación concurrentemente segura de `MPI.Init()`.
  - `public synchronized void MPI_Finalize()`:es la implementación concurrentemente segura de `MPI.Finalize()`.
  - `public synchronized int MPI_Rank()`:es la implementación concurrentemente segura de `MPI.COMM_WORLD.Rank()`.
  - `public synchronized void MPI_Barrier()`:es la implementación concurrentemente segura de `MPI.COMM_WORLD.Barrier()`.
  - `public synchronized void MPI_Send(...)`:es la implementación concurrentemente segura de `MPI.COMM_WORLD.Send()`.
  - `public synchronized Status MPI_Recv(...)`:es la implementación concurrentemente segura de `MPI.COMM_WORLD.Recv()`.
  - `public synchronized int MPI_Size()`:es la implementación concurrentemente segura de `MPI.COMM_WORLD.Size()`.
- Sincronización
  - `public void barrier()`: solicita un punto de sincronización para todas las tareas que se están ejecutando en la Plataforma.

Los métodos que están relacionados con MPI son solamente un envoltorio para resolver la concurrencia que no soportan las funciones de la librería de MPI. Estos envoltorios invocan los mtodos `MPI.*` y `MPI.COMM_WORLD.*` implementados en mpiJava.

Con todos los métodos listados, se agrupan todas las funcionalidades a las que puede tener acceso al usuario, y por tanto sólo falta comunicarle al usuario una instancia del `PlatformBind`.

6.1.2. Acceso a las funcionalidades a través de la interfaz ClassStub

Todas las funcionalidades a las que tienen acceso las clases del usuario se encuentran agrupadas dentro de la clase `PlatformBind`. Una instancia de esta clase es comunicada

```
public interface ClassStub extends Serializable{
     public void setProperties(PlatformBind platformBind);
     public void execute();
}/*ClassStub*/
```

Figura 6.1: Código de la interfaz ClassStub

a cada tarea del usuario que se crea dentro de la Plataforma, y esto se hace a través de la interfaz `ClassStub`.

El código para la interfaz `ClassStub` se encuentra en la Figura 6.1.

Todas las tareas del usuario que se vayan a ejecutar en la Plataforma deben implementar la interfaz `ClassStub`. Con esto, las clases del usuario deben implementar los métodos `setProperties()` y `execute()`.

A través del método `setProperties()` se le comunicará a la tarea del usuario una instancia del `PlatformBind` que empleará para hacer uso de las funcionalidades de la Plataforma. Este método se invocará antes de la ejecución del método `execute()`. En el método `execute()` debe encontrarse el código que el usuario desee que se ejecute.

Con la implementación de la interfaz `ClassStub`, se logrará exponer las funcionalidades de la Plataforma al usuario.

6.2. Ejemplo 1: Envío de Mensajes

En este ejemplo se hace uso del envío de mensajes. Esto se demuestra a través de una aplicación que levanta m instancias de una clase llamada `Anillo`. De esas tareas, comunica la tarea con identificador n con la tarea $n + 1$; la tarea $n = m$ se comunica con la tarea de identificador 0. La comunicación consiste en un mensaje enviado en una cadena.

El listado de la clase `Anillo` se encuentra en la Figura 6.2.

La clase `Anillo` implementa la interfaz `ClassStub`, la cual extiende de la interfaz `Serializable`, lo que obliga a la clase a definir el `serialVersionUID`. En la línea 6, almacenamos la instancia del `PlatformBind` que emplearemos dentro del método `execute()`.

Dentro del método `execute()` lo primero que hacemos es solicitar un punto de sincronización (línea 10), para esperar a que todas las tareas se sincronicen. Definimos la variable que contendrá el mensaje que se enviará (línea 15). En la línea 17, verificamos si la tarea es la última, en cuyo caso enviará el mensaje a la tarea 0. En la línea 18 se construye el mensaje con la información necesaria para enviar el mensaje; en orden los

```
1 :public class Anillo implements ClassStub{
2 :   private static final long serialVersionUID = 1L;
3 :   PlatformBind platformBind;
4 :
5 :   public void setProperties(PlatformBind platformBind){
6 :      this.platformBind=platformBind;
7 :   }//setProperties
8 :
9 :   public void execute(){
10:      platformBind.barrier();
11:
12:      System.out.println("Total tasks: "+platformBind.getTotalTasks());
13:      System.out.println("Task no:"+platformBind.getTask());
14:
15:      Message message;
16:
17:      if(platformBind.getTask()==platformBind.getTotalTasks()-1)
18:         message=new Message(platformBind.getTask(),
                              UDPComm.WILDCARD_ADDRESS,
                              0,1,
                              ("Soy el "+platformBind.getTask()).getBytes());
19:      else
20:         message=new Message(platformBind.getTask(),
                              UDPComm.WILDCARD_ADDRESS,
                              platformBind.getTask()+1,1,
                              ("Soy el "+platformBind.getTask()).getBytes());
21:
22:      platformBind.send(message);
23:      try{
24:         Message m=platformBind.receive();
25:         System.out.println(new String(m.getBuffer()));
26:      }catch(Exception e){}
27:   }//execute
28:}//Anillo
```

Figura 6.2: Código de la clase Anillo

parámetros que especificamos son:

- `platformBind.getTask()`: para indicar la tarea de destino.
- `UDPComm.WILDCARD_ADDRESS`: para indicar el nodo de destino.
- `0`: la tarea del destino.
- `1`: un entero identificador del destino. No es de importancia para el ejemplo.
- `("Soy el "+platformBind.getTask()).getBytes()`: Es el mensaje que se le envía a la tarea de destino.

La línea 20 difiere de la 18 unicamente en el parámetro 3, en el cual la tarea de destino no es 0, sino `platformBind.getTask()+1`, es decir, uno más que el identificador de la tarea que lo invoca.

En la línea 22 enviamos el mensaje a la tarea de destino, y en la línea 24 esperamos el mensaje que debemos recibir de la tarea que nos enviará (que debe ser la tarea $n-1$).

Para ejecutar nuestra aplicación, debemos comunicar al demonio maestro el nombre de la clase. En este caso, solicitaremos que la clase se ejecute como clase Esclava, y solicitaremos 3 instancias. La clase para hacer el envío de la definición de clases se encuentra en la Figura 6.3.

Después de invocar el método `c.sendDefinition()` en la línea 15, el demonio maestro recibirá la definición de clases y solicitará a cada uno de los demonios esclavos que obtengan una instancia de la clase `Anillo` hasta completar las 3 instancias que el usuario solicitó. Recuerde que la clase `Anillo` debe estar en el `CLASSPATH` del servidor de RMI.

6.3. Ejemplo 2: Memoria Compartida

En este ejemplo se demostrará el uso de la memoria compartida. La aplicación está divida en dos clases, `Promedio1` y `Promedio2`, que corresponderán a nuestras clases maestra y esclava. La aplicación calcula el promedio de 10 números. Para hacerlo, la clase maestra solicita al usuario 10 números que coloca en la memoria compartida, desde donde dos instancias de la clase esclava, llamada `Promedio1`, calcularán la suma de 5 números cada una de ellas. El resultado de esa suma se coloca de regreso en la memoria compartida, y la clase maestra realizará la división final.

Al iniciar la ejecución, las clases esclavas bloquearán la dirección de memoria en donde colocarán el resultado de las sumas que realizarán. Después se pondrán en espera de un punto de sincronización mediante una llamada al método `barrier()`. Mientras tanto, la clase maestra solicitará los datos al usuario y los irá colocando en la memoria compartida. Al terminar de leer los datos, invocará el punto de sincronización mediante `barrier()`. En ese momento, las clases esclavas se liberarán del punto de

```
1 :public class DefinicionAnillo{
2 :   /**
3 :    * El metodo main usado para enviar la definicion de las clases.
4 :    * @param args Los argumentos al main. No se emplean en este caso.
5 :    */
6 :   public static void main(String[] args){
7 :      //Declaramos la variable
8 :      ClassDefinition c=new ClassDefinition();
9 :
10:      //Indicamos que deseamos tres instancias de Anillo
11:      c.setSlaveClass("Anillo", 3);
12:
13:      try{
14:         //Intentamos enviar la definicion al demonio
15:         c.sendDefinition();
16:      }catch(Exception e){
17:         System.err.println(e.getMessage());
18:      }//end try
19:   }//main
20:
21:   public DefinicionAnillo(){}
22:}//DefinicionAnillo
```

Figura 6.3: Código de la clase DefinicionAnillo

sincronización y realizarán las sumas que les corresponden, tomando los valores de la memoria compartida. Al mismo tiempo, la clase maestra intentará leer el resultado de tales sumas de la memoria compartida, pero al haber sido bloqueadas tales direcciones por los procesos esclavos, la clase maestra se bloqueará. Cuando los esclavos terminen de sumar y colocar el resultado, liberarán el candado sobre las direcciones donde han colocado sus resultados. Tras esto, la clase maestra podrá leer el dato y continuará con la última suma y la división final.

El listado de la clase maestra, `Promedio1`, está en la Figura 6.4.

En la línea 20 se almacena el i-ésimo valor leído del usuario. La dirección donde se almacena se calcula usando el tamaño del entero dentro de la Plataforma. Esto se hace porque la memoria compartida es direccionable por bytes, no por enteros; por eso se debe calcular correctamente el desplazamiento dentro de la memoria. Después de haber leído los 10 valores del usuario, se invoca el punto de sincronización, en la línea 24.

En la línea 26, se obtiene el resultado de la suma de los primeros cinco números, que la primer clase esclava colocará en el lugar del onceavo entero. Si para este momento la clase esclava no ha terminado de realizar la suma, la clase maestra se bloqueará hasta que la clase esclava libere el candado que aplicó sobre la dirección. Pasará lo mismo cuando intente obtener la suma de los siguientes 5 números de la segunda clase esclava (línea 29). Finalmente hace la suma de los dos números y termina.

El listado de la clase esclava, `Promedio2`, se encuentra en la Figura 6.5

Lo primero que realiza la clase esclava es bloquear la dirección en la que colocará el resultado de la suma (línea 15). Sabemos que la aplicación constará de una instancia de la clase maestra y dos instancias de la clase esclava. Por la forma en que la Plataforma asigna los identificadores de tarea, sabemos que la clase maestra tendrá el identificador cero, y las clases esclavas tendrán los identificadores 1 y 2, respectivamente. En la línea 15 se calcula la dirección del resultado de la suma mediante `9+p.getTask()`. Los números que la clase maestra leerá del usuario estarán en los índices del 0 a 9. Para la primer clase esclava, con identificador 1, su resultado estará en $9 + 1$, o sea 10, el onceavo entero. Sucederá lo mismo con la segunda clase esclava, cuyo resultado de la suma lo colocará en $9 + 2$.

En la línea 17, se invoca un punto de sincronización mediante una llamada a `barrier()`. Es aquí donde las clases esclavas estarán en espera de que la clase maestra termine de leer los datos del usuario. Pasando este punto de sincronización, se comenzará a leer los datos. En la línea 20, se calcula el índice de inicio de lectura de la memoria mediante `(p.getTask()-1)*5`. Esto da como resultado 0 para la primer clase esclava (con identificador de tarea 1) y 5 para la segunda (con identificador de tarea 2).

Al terminar la suma, el resultado se coloca en la dirección que se bloqueó desde el

```
1 :import java.io.BufferedReader;
2 :import java.io.InputStreamReader;
3 :
4 :public class Promedio1 implements ClassStub{
5 :  private static final long serialVersionUID = 13L;
6 :  private PlatformBind p;
7 :
8 :  public void setProperties(PlatformBind platformBind){
9 :    this.p=platformBind;
10:  }//setProperties
11:
12:  public void execute(){
13:    System.out.println("+++++++++++++++++");
14:    System.out.println("+ Clase Maestra +");
15:    System.out.println("+++++++++++++++++");
16:    for(int i=0;i<10;i++){
17:      System.out.print("Introduzca numero ["+i+"]: ");
18:      int t=readInt();
19:      System.out.println(">>Almacenando en memoria compartida.");
20:      p.putInt(i*SharedMemory.INT_SIZE, t);
21:    }//next i
22:
23:    System.out.println("Punto de sincronizacion.");
24:    p.barrier();
25:
26:    int t1=p.getInt(10*SharedMemory.INT_SIZE);
27:    System.out.println("Suma primeros 5 numeros: "+t1);
28:
29:    int t2=p.getInt(11*SharedMemory.INT_SIZE);
30:    System.out.println("Suma ultimos 5 numeros: "+t2);
31:    double t3=(double)(t1+t2)/10.0;
32:    System.out.println("Promedio: "+t3);
33:  }//execute
34:
35:  public static int readInt(){
36:    BufferedReader standard = new BufferedReader(
                                  new InputStreamReader(System.in));
37:    try{
38:      String tmp=standard.readLine();
39:      int num=Integer.parseInt(tmp);
40:      return num;
41:    }catch (Exception e){
42:      return 0;
43:    }//end try
44:  }//readInt
45:}/*Promedio1*/
```


Figura 6.4: Código de la clase Promedio1

```
1 :public class Promedio2 implements ClassStub{
2 :  private static final long serialVersionUID = 12L;
3 :  private PlatformBind p;
4 :
5 :  public void setProperties(PlatformBind platformBind){
6 :    this.p=platformBind;
7 :  }//setProperties
8 :
9 :  public void execute(){
10:    System.out.println("++++++++++++++++++");
11:    System.out.println("+ Clase Esclava +");
12:    System.out.println("++++++++++++++++++");
13:
14:    System.out.println("Bloqueando la direccin "+(9+p.getTask()));
15:    p.lock((9+p.getTask())*SharedMemory.INT_SIZE);
16:    System.out.println("Esperando punto de sincronizacin...");
17:    p.barrier();
18:
19:    int acum=0;
20:    int index=(p.getTask()-1)*5;
21:    for(int i=0;i<5;i++){
22:      acum+=p.getInt((i+index)*SharedMemory.INT_SIZE);
23:    }//next i
24:    System.out.println("Almacenando en direccin "+(9+p.getTask()));
25:    p.putInt((9+p.getTask())*SharedMemory.INT_SIZE, acum);
26:    System.out.println("Liberando direccin "+(9+p.getTask()));
27:    p.unlock((9+p.getTask())*SharedMemory.INT_SIZE);
28:  }//execute
29:}/*Promedio2*/
```

Figura 6.5: Código de la clase Promedio2

```
1 :public class Promedio0{
2 :  public static void main(String[] args){
3 :    //Declaramos la variable
4 :    ClassDefinition c=new ClassDefinition();
5 :
6 :    //Indicamos que deseamos una instancia de Promedio1
7 :    c.setMasterClass("Promedio1", 1);
8 :
9 :    //Queremos dos instancias de la clase Promedio2
10:    c.setSlaveClass("Promedio2", 2);
11:
12:    try{
13:      //Intentamos enviar la definicin.
14:      c.sendDefinition();
15:    }catch(Exception e){
16:      System.err.println(e.getMessage());
17:    }//end try
18:  }//main
19:
20:  public Promedio0(){}
21:}//Promedio0
```

Figura 6.6: Código de la clase Promedio0

principio (línea 25), y se libera el candado para que la clase maestra pueda leerlo (línea 27).

En el listado de la Figura 6.6 se muestra la definición de clases empleada para este ejemplo.

En el listado de la Figura 6.6, solicitamos una instancia de la clase `Promedio1` y pedimos que sea tratada como clase maestra (línea 7). También solicitamos dos instancias de la clase `Promedio2` como clases esclavas. Finalmente, se envía la definición de las clases al demonio maestro (línea 14).

Capítulo 7

Conclusiones

Con el objetivo de resolver problemas de optimización a gran escala usando algoritmos paralelos, se decidió crear una plataforma propia, encaminada directamente a aprovechar la arquitectura de los clusters. En la facultad de Ciencias de la Computación de la Benemérita Universidad Autónoma de Puebla, se dispone de un cluster de 16 nodos duales.

Una de las herramientas más populares en el ámbito científico para el desarrollo de soluciones usando el mecanismo de paso de mensajes, es MPI. Sin embargo, la solución de algunos problemas se puede modelar de forma más natural empleando una memoria compartida. Con MPI, no existe ninguna construcción que permita la memoria compartida, y de requerirse, queda a tarea del desarrollador implementarla. Este problema quedó totalmente resuelto con el diseño que se realizó de la Plataforma. Dicho diseño incluye una construcción que hace las veces de memoria compartida, que existe entre todos los nodos de la misma, y todas las instancias de las clases del usuario tienen acceso a ella. Esta construcción de la memoria compartida, fue desarrollada empleando los mecanismos de concurrencia que ofrece el lenguaje de programación Java.

Las facilidades que ofrece el lenguaje para la creación de pool de hilos, permite su administración, reduciendo la sobrecarga que implica la creación y destrucción de hilos. También tiene soporte para la creación y aplicación de políticas sobre la ejecución de los hilos. Muchas de las tareas de la plataforma, e incluso la aplicación del usuario, pueden hacer uso intensivo de hilos de corta o larga vida, ya que la administración que se consigue con los pools de hilos de la versión 1.5 de Java, reduce la sobrecarga y hace eficiente su uso.

Las utilerías de Java para la concurrencia, incluyen una cola asíncrona. Con las colas asíncronas, ya no es necesario crear regiones críticas o semáforos para la adición o sustracción de elementos, ya que incluyen el control de la concurrencia dentro de la misma clase. En la Plataforma se emplearon colas asíncronas para la recepción de mensajes.

Algunas partes de la plataforma, especialmente en la memoria compartida, incluyen el manejo de los semáforos de Java.

En cuanto a la sincronización de la plataforma, se emplearon *CountDownLatches* y *CiclicBarriers*, ambos parte de las utilerías de Java para la concurrencia, para crear los puntos específicos de sincronización en dos etapas, primero a nivel nodo y luego a nivel Plataforma.

Todas las utilerías de concurrencia que ofrece el lenguaje Java a partir de su versión 1.5, no sólo son eficientes, sino que están probadas y se consideran óptimas para el lenguaje. El empleo de las utilerías de concurrencia, redujo la complejidad del desarrollo y simplificó las pruebas.

El hecho de disponer de esta plataforma permitirá a los estudiantes de la Facultad de Ciencias en Computación de la Benemérita Universidad de Puebla, el desarrollo de aplicaciones concurrentes, paralelas y distribuídas sobre un único lenguaje, solamente utilizando las clases y facilidades que ofrece la Plataforma.

En este momento del trabajo, se están desarrollando diferentes algoritmos en paralelo, usando diversos mecanismos de comunicación, como son el modelo maestro-esclavo y el modelo de isla; además, se están desarrollando aplicaciones concretas que emplean activamente el uso de la memoria compartida.

Otro objetivo en el cual se está trabajando, es el desarrollo de un algoritmo para resolver el problema de la *Mochila Multidimensional*, usando el esquema de comunicación maestro-esclavo y aprovechando todas las ventajas que ofrece la plataforma, para posteriormente compararlo con un algoritmo similar desarrollado sobre MPI y utilizando 4 nodos duales. Con esta implementación realizada sobre MPI, se presentaron problemas de sobrecarga en el nodo maestro y en el manejo de la memoria, situación que se pretende resolver ampliamente por la forma en que se manejan los bloques de memoria dentro de la Plataforma.

La Plataforma que se ha presentado aún es suceptible de muchas mejoras, especialmente en cuanto al desempeño. Todas las características que se presentan de la Plataforma son funcionales, pero pueden optimizarse para obtener menores tiempo de ejecución. Dos ejemplos de estas mejoras son la serialización de clases para la distribución y la administración de la memoria compartida.

Actualmente la serialización de clases y su distribución se realiza mediante RMI, lo que implica tener que registrar el demonio de RMI y registrar el servicio. En cuanto a la administración de memoria, se pueden emplear técnicas para realizar lecturas y escrituras por adelantado empleando dos hilos de ejecución, lo que reduciría tiempos de espera.

Aunque las mejoras mencionadas son importantes, pueden no ser las únicas. Estas mejoras eventualmente se irán agregando a la Plataforma.

Bibliografía

[WAL01] Waldo Jim, Wyant Geoff (1994), A Note on Distributed Computing.

[PAU01] Black Paul E.(2004), Algorithms and Theory of Computation.

[THO01] Thompson C.D.(1980), A complexity Theory for VLSI.

[IAN01] Foster Ian(1995), Introduction to Parallel Computers and Computation.

[DAV01] Davidson David (2001), A review: Java Packages and Classloaders.

[GOE01] Goetz Brian (2003), Concurrent Collection Classes in Java.

[FIL01] Filman Robert E.; Daniel P. Friedman(1984), Coordinated Computing: Tools and Techniques for Distributed Software

[AMD01] Amdahl Gene (1967), Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities.

[BON01] Bondi Andr B.(2000), Characteristics of scalability and their impact on performance

[NAG01] Nagarajayya Nagendra , J. Steven Mayer (2002), Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory

[YEF01] Yefim Shuf, Manish Gupta, Hubertus Franke(2002), Creating and Preserving Locality of Java Application at Allocation and Garbage Collection Times.

[PAS01] Pascal Felber (2003), Semi Automatic Paralellization of Java Applications.

[VLA01] Getov Vladimir, Philippsen Michael(2001), Java Communications for Large-Scale Computing.

[GRA01] Fagg Graham E(2003), Message Passing with MPI.

[LAM01] LAM/MPI Team at Open Systems Lab (2007), LAM/MPI Users Guide.

[BAK01] Baker Mark (2002), Thoughts on the structure of an MPJ reference implementation.

[BRY01] Carpenter Bryan, Geoffrey Fox, Sung-Hoo Ko (2003), mpiJava 1.2: API Specification.

[VLA02] Getov Vladimir, Gray Paul, Sunderam Vaidy(2001), MPI and Java-MPI: Contrast and Comparison of Low-Level Communication Performance.