Universidad de las Ciencias Informáticas

Facultad 4

Título:

Motor de Procesamiento de Peticiones Paralelas para la plataforma Génesis

Trabajo de Diploma para optar por el título de
Ingeniero en Ciencias Informáticas

Autor: Yadisnel Galvez Velázquez

Tutor: Lic. Edisel Navas Conyedo

Ciudad de La Habana, Mayo de 2009

Declaración de Autoría

Declaro que soy el único autor de este trabajo y autorizo a la Facultad 4 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio. Para que así conste firmo el presente a los _____ días del mes de _____ del año _____ .

Autor:
Yadisnel Galvez Velázquez

Tutor:
Lic. Edisel Navas Conyedo

_____

Firma del autor

_____

Firma del tutor

Agradecimientos

A nuestro Comandante Fidel y a nuestro querido Raúl: gracias por llevar adelante este sueño hecho realidad, seguiremos para siempre forjando el futuro.

A mis padres, que de tantas maneras saben llenarme de felicidad, que con tantos esfuerzos me permitieron ser yo.

A mis amigos, por su incondicional apoyo en todo momento, por sus oportunos abrazos y palabras sinceras.

A mi equipo EIPAD: ustedes son mi orgullo, siempre habrá un espacio infinito para seguir viviendo en la armonía de su trabajo, su responsabilidad y amistad. Sin ustedes nada hubiese sido posible.

A todos aquellos que dedicaron parte de su luz para alumbrar mi camino, a todos, muchas gracias!

Dedicatoria

A mis padres, por su infinito amor, confianza y dedicación. Gracias por regalarme la dicha de con sus corazones, hacer latir el mio: no hay nada más grato que ser parte y fruto de ustedes.

A mi hermano y mis abuelos, por alegrar cada segundo de mi vida, por ser sencillamente parte de mí.

Resumen

El procesamiento de grandes volúmenes de datos, en tiempos moderados, se encuentra limitado por la incapacidad de ser procesados por ordenadores con arquitectura Von Newman. La adquisición de supercomputadores para llevar a cabo tareas de cómputo complejas, unido al costo por mantenimiento una vez adquiridos, constituyen una opción no viable en la mayoría de las situaciones presentadas para nuestro país. Una solución alternativa ante esta disyuntiva, es la utilización de herramientas informáticas capaces de sustituir, mediante otros métodos, este tipo de tareas. La Computación Paralela, como rama de la informática, dedica su estudio a este tipo de herramientas y su aplicación práctica. La mayoría de las herramientas disponibles para cómputo paralelo están pensadas para entornos científicos y no poseen aplicación comercial por sí mismas careciendo de un enfoque producto, utilizándose principalmente para tareas que requieran gran volumen de cómputo en entornos controlados donde solo se prestan servicios. Un mercado incipiente en constante crecimiento se encuentra localizado en el procesamiento de grandes volúmenes de datos en empresas que necesitan Sistemas para la Toma de Decisiones, Almacenes de Datos, Minería de Datos, entre otros. La Plataforma Génesis intenta brindar una solución a estas necesidades mediante el cómputo paralelo de datos. El presente trabajo tiene el objetivo fundamental, realizar una investigación de las herramientas de cómputo paralelo existentes, cuyo resultado final culmine con la conformación de un Motor de Procesamiento de Peticiones Paralelas, que permita brindar una solución acorde a las necesidades de la Plataforma Génesis.

Índice general

Índice de figuras

Índice de cuadros

Introducción

El desarrollo acelerado de las Tecnologías de la Información y las Comunicaciones (TIC), unido al establecimiento del conocimiento y la información como activos intangibles cada vez más importantes para el desarrollo, trae aparejada la necesidad de implementación de sistemas para el procesamiento de datos en la obtención de información mucho más exigentes. El procesamiento de grandes volúmenes de datos en tiempos moderados, se ve limitado por la incapacidad de ser procesados por los ordenadores con arquitectura Von Newman. Una solución alternativa ante la imposibilidad de adquisición de superordenadores capaces de enfrentar estas necesidades de cómputo, es la utilización de los recursos individuales de un conjunto de computadoras o elementos de proceso. Por el grado de importancia que supone el procesamiento en tiempos relativamente cortos de grandes volúmenes de cómputos, se hace necesaria la adquisición de sistemas informáticos que ofrezcan una plataforma flexible para afrontar estos problemas. Sin embargo, la mayoría de las soluciones afines existentes, se enfocan más al procesamiento de grandes volúmenes de cálculos, no así de datos, lo que unido a los elevados costos de las licencias o la imposibilidad de adquirirlas influye de manera negativa en el desarrollo de los procesos productivos que los necesiten.

La creciente informatización de nuestro país permitirá una mejor administración de los recursos estatales y acelerará los procesos productivos, elementos que se verán impulsados principalmente por la implantación de sistemas ERP, que ya se encuentran en desarrollo[1]. Si bien es cierto que los sistemas anteriormente descritos influyen de manera positiva en el desarrollo de nuestro país, esto traerá aparejada la necesidad de procesar grandes volúmenes de datos en función de la toma de decisiones. Los sistemas informáticos para la gestión y planificación como los ERP no están pensados para el análisis profundo y a gran escala de todos los datos obtenidos. Elementos como predicciones económicas a largo plazo según datos históricos precedentes, entre otros, no están contenidos dentro de estos.

Desde el punto de vista investigativo, existen en nuestro país otros proyectos que necesitan de

[1]Proyecto ERP-CUBA, actualmente en desarrollo por la Universidad de las Ciencias Informáticas.

procesamiento de grandes volúmenes de datos o cálculo. En institutos científicos, como el Finlay, se necesitan consultar bases de datos gigantescas en investigaciones sobre la compatibilidad molecular para la fabricación de fármacos, elementos de gran relevancia económica para nuestro país.

En la Universidad de las Ciencias Informáticas existen algunos proyectos investigativos que poseen como línea el Procesamiento Paralelo de Datos, o la realización de cómputos paralelos en un sentido más amplio, con el objetivo de presentar soluciones a los problemas existentes. Tal es el caso del proyecto investigativo Tarenal[2]de la Facultad 6 de la Universidad de las Ciencias Informáticas. En todos los casos se plantea que no existe una plataforma para acelerar los procesos de desarrollo de software para cómputo de grades volúmenes de cálculos o datos. La principal ventaja de la realización de una plataforma frente a otras soluciones es que brinda una arquitectura mucho más flexible, que constituyendo una solución genérica, permite ser adaptable a necesidades específicas, ya sea el desarrollo de proyectos que necesiten Procesamiento Paralelo de Datos (sobre bases de datos), realización paralela de cálculos o incluso ambos elementos.

La Plataforma Génesis se sustenta sobre la base del desarrollo de tres premisas fundamentales; La Interfaz de Usuario: supervisión y control de los procesos de cómputo paralelo; Manipulación de Datos: almacenar, consultar y procesar los datos de todo el sistema de la manera más rápida posible; Gestionar las peticiones de cálculo paralelo: define la manera en que se comunican los nodos del clúster e interactúan las tareas de cómputo paralelo para la resolución de problemas. Garantizar la correcta comunicación de las tareas de cómputo entre los nodos de un Clúster constituye un elemento de vital importancia para el buen funcionamiento de los sistemas para cómputo paralelo, de esto depende la eficiencia y la eficacia para resolver problemas. No existe una herramienta informática capaz de gestionar las peticiones de cómputo paralelo de manera eficiente en el Clúster de Génesis y esto viene a ser la **Situación Problemática** que se presenta en el desarrollo de la plataforma.

Partiendo del estudio de los objetivos que se persiguen con la construcción de Génesis, como una plataforma que garantice la correcta comunicación de los nodos y las tareas de cómputo del clúster, se identificó el siguiente **Problema:** ¿Cómo gestionar de manera eficiente la comunicación entre los nodos y tareas de cómputo en la Plataforma Génesis?

El **Objeto de Estudio** en el cual está localizado el problema es: Realización de cómputo paralelo sobre bases de datos.

El **Campo de Acción** abarcado es el siguiente: Motores de procesamiento y alternativas para

[2]Proyecto investigativo para la separación molecular desarrollado en la Facultad 6 de La Universidad de las Ciencias Informáticas.

cómputo paralelo sobre bases de datos.

Se tiene como **Objetivo General:** Implementar un sistema informático que garantice el procesamiento correcto de los datos y la realización eficiente de cómputo paralelo sobre los mismos.

Como **Objetivos Específicos** quedan definidos los siguientes:

1. Modelar los procesos que ocurren durante la interacción de los nodos y tareas de cómputo en un clúster para cómputo paralelo de sobre bases de datos.

2. Implementar una aplicación informática, que se ejecute como proceso en todos los nodos del clúster, garantizando la correcta comunicación e interacción entre nodos y tareas de cómputo.

Como guía de la investigación se plantea la siguiente **hipótesis:** Si se desarrolla una aplicación capaz de ejecutarse simultáneamente en todos los nodos del clúster, permitiendo la correcta comunicación entre nodos y tareas de cómputo, se lograrán ejecutar algoritmos para cómputo paralelo sobre bases de datos en la plataforma Génesis.

Las tareas que se llevarán a cabo para dar cumplimiento a los objetivos trazados quedan descritas como sigue:

1. Estudiar los protocolos y sistemas de comunicación en las arquitecturas para cómputo paralelas y distribuido existentes.

2. Estudiar las bibliotecas de clases de comunicación para redes de computadoras existentes en la actualidad.

3. Realizar la modelación de los componentes que se integrarán utilizando el proceso de software seleccionado.

4. Implementar una aplicación que permita la correcta comunicación entre nodos y tareas de cómputo, permitiendo la ejecución de algoritmos para cómputo paralelo sobre bases de datos.

El presente trabajo está compuesto por Introducción, cuatro capítulos, veintidós epígrafes, treinta y cuatro subepígrafes, Conclusiones, Recomendaciones, Bibliografía , Glosario de Términos y un anexo.

En el Capítulo 1, Fundamentación Teórica, se realiza un estudio sobre las historia y evolución del procesamiento paralelo y el surgimiento de la computación paralela. Se analizan las herramientas para cómputo paralelo existentes, así como los protocolos de comunicación para el intercambio de mensajes.

Al finalizar el capítulo se describen las herramientas que serán utilizadas durante el desarrollo del Motor de Procesamiento de Peticiones Paralelas.

En el Capítulo 2, Desarrollo del Motor de Procesamiento de Peticiones Paralelas, se realiza la modelación y descripción de los requisitos funcionales, el desarrollo de los componentes del motor así como su interrelación. Por otro lado se analiza el manejo de la concurrencia en la protección de los datos.

En el Capítulo 3, Comunicación Cliente-Servidor, se profundiza en la Arquitectura CORBA utilizada en la implementación de los componentes de red para la comunicación entre aplicaciones MPPP. Son analizados a detalle los principales servicios de comunicación implementados y su relación.

En el Capítulo 4, Adición de módulos y configuración, queda descrita la metodología para añadir módulos dinámicamente al motor, ya sean módulos de cómputo o soporte para diferentes tipos de bases de datos. Se ilustra la manera en que deben ser desarrollados y añadidos los algoritmos paralelos y los resultados arrojados en algunas de las pruebas realizadas al motor con los mismos. Además son detalladas las diferentes secciones de configuración del motor y el significado que poseen para el mismo.

Capítulo 1

Fundamentación teórica

1.1. Introducción

El presente capítulo aborda los tipos de arquitecturas paralelas existentes así como el desarrollo y la evolución de los clúster como base fundamental para la implementación de sistemas de cómputo paralelo. También se realiza un estudio sobre el estado actual de los sistemas para Procesamiento Paralelo, las arquitecturas y protocolos para la comunicación y las tecnologías que serán empleadas para asumir la modelación e implementación del Motor de Procesamiento de Peticiones Paralelas para la plataforma Génesis.

1.2. La Computación Paralela

La complejidad de un algoritmo para ser procesado está determinada principalmente por la complejidad temporal del mismo y por el volumen de los datos sobre el cual opera. Sobre este concepto muchos investigadores han basado modelos de fabricación de dispositivos de cómputo que a nivel de hardware o software permitan disminuir el tiempo de ejecución. La Computación Paralela engloba un conjunto de técnicas que permiten minimizar cómputos que suponen un tiempo elevado para su terminación a un tiempo moderado. Mientras que el cómputo paralelo se refiere al instante de tiempo (simultaneidad) en el que se ejecutan las instrucciones de los algoritmos, el procesamiento distribuido se refiere al lugar en el que estas se realizan por lo que estas definiciones no son excluyentes.

Así el Procesamiento Paralelo[2] queda definido como:

"Método de procesamiento que puede ejecutarse únicamente en una computadora que contenga dos o más procesadores en funcionamiento simultáneo (...)."

Esta definición hace referencia claramente a procesadores como elementos de proceso pues actualmente en un microprocesador pueden estar empaquetados múltiples núcleos y cada cual procesar en paralelo a través de más de un hilo de procesamiento.

Por otro lado la Computación Paralela[3] está definida como:

"El uso de varias computadoras o procesadores para resolver un problema o realizar una determinada tarea en común."

Nuevamente y de manera más clara se especifica que es el uso de computadoras o procesadores, no computadoras de manera general, esta definición incluye cualquier elemento de proceso, los cuales en la práctica son muchos. Esto significa que puede desarrollarse computación paralela sobre una computadora (nodo) con un solo procesador con múltiples núcleos, múltiples procesadores o con múltiples computadoras conectadas entre sí.

1.2.1. Breve reseña histórica de la Computación Paralela

La Computación Paralela comienza su historia con el interés de compañías productoras de circuitos electrónicos para computadoras con capacidad de procesamiento paralelo. Para el año 1955, IBM desarrolla circuitos aritméticos paralelos binarios unido a una unidad de punto flotante para el IBM 704[35], que aceleraba los procesos de cómputo numérico de manera considerable frente a las soluciones existentes (aritmético-lógicas). Algunas mejoras a este diseño fueron añadidas como la utilización de canales independientes en la entrada y salida frente al modelo de un solo canal que ralentizaba el procesamiento. Así ya para el año 1956 IBM se traza la meta de desarrollar una computadora 100 veces más rápida que las de su época, resultado de un proyecto llamado IBM 7030[34].

De esta manera fueron desarrollándose un conjunto de proyectos que tenían como meta la construcción de superordenadores que basaban su rapidez en la realización de cómputo paralelo pero no existían aún contribuciones importantes en el soporte paralelo del software desarrollado para estos. Un avance notable fue el lanzamiento por Bull[1] en 1958 de la "Gamma 60" que incluía dentro del flujo de instrucciones el ***join*** y el ***fork***, que permitían desarrollar algoritmos para cómputo paralelo brindando el soporte para sistemas multitarea. En ese mismo año, dos empleados de IBM, John Cocke y Daniel Slotnick, plantean el uso del paralelismo en cálculos numéricos para acelerar los procesos de cómputo lo que constituyó la base para trabajos posteriores.

Para el año 1959 Sperry Rand, entrega la computadora LARC (Livermore Automatic Research Computer) que unido a la STRETCH (IBM 7030) de IBM constituyen las primeras computadoras de gran envergadura en la utilización de los transistores. Utilizando lenguaje ensamblador en vez de lenguaje de máquina, poseía un procesador de entrada y salida independiente que podía operar en paralelo con dos unidades de procesamiento. Esta fue desarrollada para procesar grandes volúmenes de datos en laboratorios de investigación de energía atómica, solo fueron construidas dos.

Luego de estos proyectos se desarrollan avances importantes en el hardware impulsado por el uso de los transistores pero no es hasta el año 1965 cuando General Electric, el MIT y AT&T Bell Laboratories, comienzan el desarrollo de Multics[30], un sistema operativo de memoria compartida, multiprocesamiento y tiempo compartido, que aunque abandonado por Bell en 1969, aportó ideas para el desarrollo de posteriores sistemas operativos. Así desarrolladores de Multics, influenciados por el proyecto se dieron a la tarea de construir el sistema operativo Unix.

Un aporte importante en el área fue el planteamiento del Problema de las Regiones Críticas por Edsger Dijkstra en 1965, donde describe el uso de los semáforos como solución a la manipulación de recursos compartidos (direcciones de memoria). Este aporte constituyó un avance notable para el desarrollo de sistemas operativos multitarea y programas con múltiples hilos y múltiples procesos. En este mismo año James W. Cooley y John W. Tuke describen el algoritmo de la Transformada Rápida de Fourier[8] el cual es considerado un gran consumidor de ciclos de puntos flotantes, el cual tiene gran aplicación en la multiplicación de números enteros grandes, filtrado digital y tratamiento digital de señales resolviendo ecuaciones diferenciales parciales.

La Ley de Amdahl[8], publicada por Gene Amdahl, científico de IBM, estableció en 1967 un marco teórico que describe matemáticamente el aceleramiento que se puede esperar al realizar de manera simultánea, series de tareas en una arquitectura paralela. Este aporte constituye sin dudas la base de la ingeniería del cómputo paralelo en sistemas multiprocesador o de clúster.

El comienzo de la construcción de ordenadores vectoriales fue marcado en 1976 por Cray Research con la Cray-1 que daba un salto revolucionario en la velocidad de procesamiento, dado que esta tecnología se basa en la posibilidad de ejecutar operaciones matemáticas sobre múltiples datos de manera simultánea.

Alrededor del año 1983, los protocolos de comunicación tenían la madurez necesaria para compartir recursos, se establecieron las condiciones para crear sistemas distribuidos. Esto conllevó al surgimiento del clúster como otra tendencia en la súper computación, pudiendo unir un conjunto de computadoras (nodos) mediante una red de alta velocidad para que trabajen de forma cooperativa en la realización de

tareas complejas. El surgimiento de Internet marcó el desarrollo de nuevas herramientas para minimizar los tiempos de procesamiento y surge la Computación Distribuida como un nuevo modelo para resolver problemas de computación masiva.

1.2.2. Tipos de paralelismo

Dentro del cómputo paralelo podemos tener dos tipos básicos de paralelismo atendiendo al nivel donde ocurre el mismo en la ejecución del software. Por una parte si el paralelismo ocurre a nivel de instrucción es *implícito* y por otra si es está incluido directamente dentro del código del programador es *explícito*. En el primer caso está dado por la capacidad con que ha sido equipado el procesador y demás hardware del ordenador para realizar procesamiento paralelo; por ejemplo algunos ordenadores están dotados de múltiples unidades aritmético-lógicas (ALU) procesando la aritmética entera separada de la aritmética de punto flotante, consiguiendo paralelismo a nivel de instrucción, otro ejemplo son los procesadores vectoriales que operan un conjunto de instrucciones sobre diferentes flujos de datos en algoritmos que pueden ser implementados de forma paralela de manera natural. Desde otra perspectiva, el paralelismo explícito hace referencia a otra forma de obtener comportamiento paralelo que se realiza de la mano del programador, ya sea a través de procesos, hilos de ejecución o paso de mensajes sobre una o varias unidades de procesamiento, como los sistemas con múltiples procesadores o de múltiples núcleos de Intel, clúster de computadoras u otras tecnologías, por citar los métodos más comunes.

Podemos clasificar de manera genérica la resolución de problemas computacionales, de acuerdo al modo de realizar los algoritmos resolutivos, en dos formas diferentes:

Computación secuencial: El algoritmo es realizado paso a paso, cada paso es comenzado cuando su predecesor es terminado.

Computación paralela: El algoritmo es realizado en cualquier orden, dos o más pasos pueden estar siendo ejecutados de manera simultánea.

Precisamente la Computación Paralela viene a solventar el "cuello de botella" que se genera debido a la explotación de la velocidad de un único procesador. Las aplicaciones más beneficiadas por la computación paralela son aquellas en las que la naturaleza del problema que resuelven, permite subdividir el mismo en problemas más pequeños. Es necesario resaltar que existen problemas que son imposibles de subdividir y por consiguiente nunca podrán paralelizarse.

1.2.3. Descomposición de los algoritmos

Los algoritmos pueden descomponerse, si es posible, desde el punto de vista funcional o de los datos o una combinación de ambos para lograr paralelizarlo. Las descomposiciones anteriores son descritas a continuación:

Descomposición de datos: El algoritmo es el mismo, se distribuye sobre un conjunto de procesadores a los cuales se le hace llegar un subconjunto de los datos. Un ejemplo claro podría ser el siguiente: se desean multiplicar dos matrices muy grandes, a cada procesador le son enviadas una fila y una columna de las matrices multiplicando, el algoritmo residente en cada uno de estos consiste en multiplicar de forma vectorial las mismas y devolver el elemento resultante, conformando de esta manera la matriz producto.

Descomposición funcional: El algoritmo es dividido en subtareas de forma lógica, asignando cada una o alguna de estas a un procesador. Datos específicos son enviados a cada procesador para que las tareas trabajen sobre estos y den por terminado el algoritmo. Un ejemplo podría ser el siguiente: Se tiene un algoritmo que está compuesto básicamente de tres integrales distintas, a tres procesadores se le asignan tres subtareas que resuelven cada una de estas integrales respectivamente. Datos específicos para cada subtarea son enviados a cada procesador, dándole solución a las mismas de manera simultánea y conformándose con la unión de todas un resultado final.

A diferencia de los procesadores secuenciales, los cuales ejecutan una instrucción por vez, las arquitecturas paralelas ejecutan más de una instrucción de forma simultánea, combinando múltiples procesadores y múltiples CPU.

1.2.4. Taxonomía de Flynn

La Taxonomía de Flynn[10] constituye una clasificación de las arquitecturas de computadoras atendiendo al flujo de instrucciones que estas realizan y sobre el flujo de datos sobre los cuales las mismas operan. De esta manera Michel J. Flynn (1972) definió las siguientes clasificaciones:

SISD (Single Instruction Single Data): Un solo flujo de instrucciones, un solo flujo de datos. Se refiere a un ordenador con un procesador que ejecuta las operaciones de forma secuencial, para que este modelo pueda realizar cómputo paralelo debe realizarlo de manera implícita. Ejemplos de esta arquitectura lo constituyen los Mainframe o los PC con un solo procesador y un solo núcleo.

SIMD (Single Instruction Multiple Data): Un solo flujo de instrucciones, múltiples flujos de datos. Este modelo describe varios elementos de proceso operando las mismas instrucciones pero sobre con-

juntos diferentes de datos de manera sincronizada. Las arquitecturas con procesamiento vectorial como las Cray están contenidas dentro de esta clasificación.

MISD (Multiple Instruction Single Data): Varios flujos de instrucciones, un solo flujo de datos. Este modelo describe sistemas donde un solo flujo de datos es procesado por varios flujos de instrucciones. Tiene especial utilidad donde se necesiten múltiples sistemas redundantes para asegurar la disponibilidad de un recurso ante el fallo del sistema principal, por ejemplo los programas de control de tráfico aéreo.

MIMD (Multiple Instruction Multiple Data): Varios flujos de instrucciones, varios flujos de datos. En este modelo se ejecutan de manera simultánea múltiples algoritmos, que pueden ser diferentes, sobre conjuntos de datos los cuales también pueden ser diferentes. Utilizando en la mayoría de las ocasiones memoria distribuida, constituye la clasificación donde comúnmente se encuentra un clúster de computadoras.

1.2.5. Clasificación de acuerdo a la utilización de la memoria

Arquitecturas con memoria compartida: tienen su fundamento principalmente en la posibilidad de que puedan compartir un espacio de direcciones de memoria único por varios procesadores. La memoria puede estar constituida incluso por varias memorias físicas. Este modelo puede ser implementado utilizando hilos de ejecución (Threads) en vez de procesos lo cual hace mucho más fácil compartir espacios de direcciones de memoria. Los hilos de ejecución o también llamados procesos ligeros están ampliamente soportados en la mayoría de los sistemas operativos actuales. Existen múltiples bibliotecas de clases que los implementan, por ejemplo la biblioteca Posix Threads[22], desarrollada por el proyecto GNU[12] principalmente para sistemas GNU/Linux.

A la especificación del tipo MIMD con memoria compartida se le nombra SMP (Symmetric Multiprocessors), cuya implementación consiste en la unión de un conjunto de procesadores que funcionan como un único ordenador teniendo acceso a la misma memoria.

Arquitectura con memoria distribuida: Se basan principalmente en el intercambio de mensajes. Un conjunto de computadoras se unen entre sí a través de una red de alta velocidad y la memoria de cada una es consultada por sus procesadores, conformando de esta manera, una computadora más potente. En este caso, compartir la memoria a través de mensajes por red, trae aparejado problemas de rendimiento por latencia. Al utilizar una red como intermediaria entre las computadoras, se introduce un costo en la complejidad temporal de los algoritmos. La ventaja está en que cada procesador opera

sobre un conjunto particular de datos de forma independiente dando solución a la subtarea asignada. En estos casos debe buscarse un equilibrio entre la cantidad de mensajes intercambiados por red y el tamaño de los mismos en aras de disminuir el efecto de la latencia. Existen librerías ampliamente difundidas para realizar este tipo de implementaciones, podrían mencionarse por ejemplo la librería MPI[16] o PVM[23] por hacer alusión a las más utilizadas.

Algunos ejemplos de sistemas con memoria compartida lo constituyen:

MPP (Massively Parallel Processor): Es un sistema constituido por un conjunto de procesadores (nodos) que están conectados entre sí por una red de alta velocidad. Cada nodo posee una copia de un sistema operativo, el cual no tiene por que ser el mismo. Además cada nodo presenta una memoria independiente y de uno o más procesadores. Algunos nodos pueden estar preparados para funcionalidades extra como almacenes de datos etc.

Clúster: La palabra clúster proviene del inglés y significa racimo. En el contexto de la informática un clúster es un conglomerado de computadoras (nodos) que se encuentran interconectadas por una red de alta velocidad en una misma localización geográfica. Un clúster es de tipo Beowulf si fue construido con componentes de hardware económicos: en la mayoría de los casos los nodos de los clúster Beowulf poseen componentes mínimos para trabajar, como tarjeta madre, procesadores, tarjetas de red, incluso muchas veces carecen de disco duro. Por otro lado se dice que un clúster es de tipo NOW (Network of Workstations), si el mismo es construido sobre redes de computadoras destinadas a otros usos (ej. laboratorios docentes etc.).

Dentro de las aplicaciones de los clúster existen cuatro necesidades principales actualmente:

1. Lograr alta disponibilidad: Conformar mediante técnicas informáticas un sistema que permita disponibilidad permanente de determinado servicio o recurso minimizando las fallas del mismo.

2. Lograr alta confiabilidad: Conformar mediante técnicas informáticas un sistema que permita disponibilidad permanente de determinado servicio o recurso eliminando completamente las fallas del mismo.

3. Lograr alto rendimiento: Realizar, mediante diversas técnicas informáticas, la ejecución de un algoritmo o proceso, que usualmente son imposibles de de realizar o su tiempo de realización es muy elevado, en un tiempo relativamente corto en comparación con el modelo secuencial.

4. Lograr balance de carga: Realizar mediante técnicas paralelas la distribución uniforme de trabajo en proporción con las capacidades individuales de una agrupación de computadoras (nodos) que

han sido destinadas a cumplir una tarea en conjunto.

1.3. Software y soluciones para cómputo paralelo

Los sistemas informáticos para procesamiento paralelo son ya muy variados en la actualidad, cada uno con ventajas y desventajas que los los hacen resaltar según el problema tratado. Existen por ejemplo lenguajes para procesamiento paralelo los cuales han sido utilizados como Mentat, ACLAN, pero de manera general no poseen gran aceptación pues las aplicaciones desarrolladas con los mismos poseen escasa portabilidad.

Por otro lado pueden utilizarse macros que realizan paralelización de determinadas funciones las cuales aunque utilizadas no suplen las necesidades existentes pues tienen la limitante de su difícil mantenimiento y escasas funcionalidades ante la variabilidad y diversidad de los problemas nacientes.

Otra tendencia para el desarrollo de sistemas para procesamiento paralelo es la utilización de implementaciones de librerías, las cuales son incluidas en lenguajes de programación estándares como C/C++, Java, C#, Python, etc. y que proveen al desarrollador un conjunto de primitivas que constituyen una API generalmente para trabajo sobre una máquina virtual, que abstrae al desarrollador sobre la complejidad de la red. Como ejemplo de estas podemos citar las ampliamente utilizadas PVM[23] y MPI[16] con sus diferentes implementaciones y extensiones. Estas APIs aunque optimizadas para el cómputo paralelo, principalmente para problemas de alta complejidad matemática, no están pensadas para trabajar sobre bases de datos relacionales, sobre las cuales no ofrecen ninguna funcionalidad.

Además de las tecnologías anteriormente descritas existen proyectos que intentan dar solución a problemas de cómputo paralelo a nivel de sistema operativo, tal es el caso del proyecto openMosix[19], que ofrece módulos integrados al núcleo del sistema operativo permitiendo a los programas desarrollados poseer un dominio mayor de los recursos. Otra característica importante de este proyecto es que realiza balance de carga dinámico, esto quiere decir que migra los procesos (algoritmos de cálculo) de los nodos más cargados a los menos cargados aunque estos estén en plena ejecución.

En otra arista podemos encontrar las Grids[18], las cuales se basan en la utilización de un middlewarepara enmascarar entornos heterogéneos y proporcionar un modelo de programación más flexible y ágil para los desarrolladores de aplicación. Las tecnologías Grids están pensadas principalmente para redes de estaciones de trabajo orientadas a Internet, aprovechando los recursos no utilizados por usuarios que los comparten en diferentes localizaciones geográficas. Ejemplo de estas tecnologías podemos citaren framework Faucets[9] que basa su fortaleza en la utilización de “ciclos de CPU” no utilizados

para realizar cómputo paralelo.

En general existen muchas soluciones, cada una con características que resaltan sobre los objetivos para las cuales fueron creadas; tareas como la minería de datos aun no podemos encontrarlas fácilmente implementadas en plataformas dedicadas a estas, sino que se localizan proyectos que implementan soluciones de minería específicas ya sea con clúster, Grids etc.

1.4. Medidas de rendimiento del la Computación Paralela

Un algoritmo no siempre es paralelizable; existen problemas inherentemente secuenciales que impiden el desarrollo de algoritmos paralelos para brindarle solución a los mismos. Un algoritmo es paralelizable, en primera instancia, si su flujo de instrucciones puede ser subdividida en subtareas secuenciales no dependientes entre sí, las cuales pueden ser ejecutadas en paralelo. Puede encontrarse también el caso en el cual sea prudente subdividir los datos de un dominio antes que al algoritmo y aplicarle el mismo de manera casi íntegra a cada parte del dominio de datos, obteniendo en cada caso un resultado parcial tributa a la solución general. Por ejemplo, tenemos un algoritmo que consta de tres pasos, primero multiplicar dos matrices, luego calcular una integral numérica y tercero unir los dos resultados anteriores con una multiplicación escalar. En este caso los pasos uno y dos pueden ser ejecutados en paralelo pero el flujo de instrucciones para hacer cada uno es diferente, en este caso nos referimos a subdividir el algoritmo en tres subtareas diferentes desde el punto de vista algorítmico. Por otro lado si quisiéramos imprimir todos los números primos que existen entre cero y **n** pudiera pensarse en subdividir este rango en rangos más pequeños y aplicarle el mismo algoritmo a cada uno, imprimiendo cada uno una cantidad parcial de números primos y en total obtendríamos el resultado deseado. En este último caso fue evidente subdividir el dominio de datos.

La primera limitante que posee el procesamiento paralelo es que los algoritmos poseen, en la mayoría de los casos, partes que no son paralelizables y por ende disminuyen el rendimiento de los sistemas. Por otro lado la paralelización es un proceso donde se intercambian datos entre elementos de proceso, ya sea a través de una memoria compartida o distribuida lo cual requiere de software y tiempo extras a la concepción del algoritmo paralelo. En todos los casos un análisis de los recursos disponibles y los modelos matemáticos e informáticos a aplicar conllevarán a desarrollar la solución más eficiente. Un elemento importante en el desarrollo de los algoritmos para procesamiento paralelo lo constituye la monitorización del rendimiento de los sistemas paralelos y su comportamiento ante nuestros algoritmos, ya que a partir de los mismos podemos comprender bajo que circunstancias nuestros algoritmos son

más eficientes.

1.4.1. Medidas generales

La medida del rendimiento de un ordenador depende de varios factores, no es posible definir por simple observación de sus componentes su rendimiento. De esta forma factores como frecuencia de memoria RAM, frecuencia de microprocesador, revoluciones por minuto del disco duro, ancho de banda del FSD (Front Side Bus), memoria caché, cantidad de núcleos del procesador, soporte para múltiples hilos, entre otros, determinan el rendimiento total del sistema.

Una medida común del rendimiento es la cantidad de ciclos de reloj por instrucción (CPI) la cual queda definida como sigue:

$CPI=\frac{N_c}{N_i}$ 1)

donde N_c es la cantidad de ciclos de reloj consumidos y N_i la cantidad de instrucciones ejecutadas. De esta manera si un programa tiene **n** instrucciones el tiempo total que demora un ordenador en ejecutarlas puede determinarse por la siguiente ecuación:

$t = CPI * n * \frac{1}{f}$ 2)

donde el término $\frac{1}{f}$ es el inverso de la frecuencia de reloj y por tanto el tiempo en realizar un ciclo de reloj.

En el procesamiento paralelo, una manera de medir la aceleración obtenida al paralelizar un algoritmo, es establecer una proporción entre el tiempo consumido por el mismo con un solo procesador y el tiempo consumido por el mismo algoritmo con **n** procesadores. De acuerdo a lo anterior puede plantearse la siguiente proporción:

$a = \frac{t_1}{t_n}$ 3)

donde $a$es la aceración obtenida (speed up) o ganancia de velocidad con **n** procesadores y t_1y t_2 el tiempo que demora en realizar el algoritmo para 1 y **n** procesadores respectivamente. Nótese que el término t_n teóricamente, tiende a disminuir por lo que esto nos da la medida de cuantas veces es más rápido nuestro algoritmo paralelo en la medida que aumentamos la cantidad de procesadores.

Si en la ecuación 3) tomamos para condiciones ideales el tiempo para un procesador $t_1 = 1$, teóricamente pudiera alcanzarse un tiempo ideal para **n** procesadores $t_n = 1/n$ lo cual nos indica que la ganancia de velocidad ideal (a_{ideal}) de un sistema de **n** procesadores es **n.**

$a_{ideal} = \frac{t_1}{t_n} = \frac{1}{1/n} = n$ 4)

A la proporción entre la ganancia de velocidad real y la ganancia de velocidad ideal se le denomina

eficiencia, la cual está determinada como sigue:

$E = \frac{a}{a_{ideal}} = \frac{t_1}{n*t_n}$ 5)

la cual nos da la medida en que se aprovechan los recursos. Esta ecuación nos sugiere que para un numero de procesadores **n** constante y un tiempo t_1(tiempo consumido por el algoritmo para un procesador) también constante, la optimización de nuestro algoritmo para **n** procesadores aumentará la eficiencia, sin embargo en la medida que aumenten la cantidad de procesadores y nuestro algoritmo no los aproveche la eficiencia se verá disminuida.

1.4.2. Ley de Amdahl

Podría pensarse que mientras mayor sea la cantidad de procesadores, menor será siempre el tiempo que un algoritmo paralelo se demora, esto no siempre es así ya que los procesos suelen tener partes no paralelizables, en otras palabras puramente secuenciales, lo que establece para cada caso un límite superior a la ganancia de velocidad. El científico de IBM, Gene Amdahl (1967) propuso una ley[15] que establece una forma de medir este límite superior.

Si tenemos que S_p es la parte paralelizable de un programa y N_p es la parte no paralelizable, entonces:

$t_1 = S_p + N_p$ 6)

$t_n = \frac{S_p}{n} + N_p$ 7)

por tanto la ganancia de velocidad (a) de la ecuación 3) podría ser transformada en:

$a = \frac{S_p+N_p}{\frac{S_p}{n}+N_p}$ 8)

dividiendo numerador y denominador por $t_1 = S_p + N_p$ obtenemos la expresión:

$a = \frac{1}{\frac{S_p}{n(S_p+N_p)}+\frac{N_p}{S_{p+N_p}}}$ 9)

Si llamamos f a la fracción de tiempo que representa la parte no paralelizable sobre el total, obtenemos que:

$f = \frac{N_p}{S_p+N_p}$ 10)

lo cual nos a una medida del tiempo que representa el tiempo secuencial con respecto al total y es llamada **cuello de botella secuencial** . Si transformamos 9) en términos de f entonces:

$a = \frac{1}{\frac{S_p}{n(S_p+N_p)}+f}$, como de 10) $S_p = \frac{N_p(1-f)}{f}$ entonces:

$a = \frac{1}{\frac{1-f}{n}+f} = \frac{n}{1-f+nf} = \frac{n}{1+f(n-1)}$ 11)

Si el numero de procesadores crece indefinidamente puede plantearse la cota superior para la ganancia de velocidad como:

$a_{lim} = lim_{n\to\infty} \frac{n}{1+f(n-1)} = \frac{1}{f}$ 12)

Análogamente, sustituyendo 4) y 11) en 5), la eficiencia estará dada como:

$E = \frac{1}{1+f(n-1)}$ 13)

donde si la cantidad de procesadores crece indefinidamente:

$E_{lim} = lim_{n\to\infty} \frac{1}{1+f(n-1)} = 0$ 14)

Las cotas superiores 12) y 14) constituyen en si mismas la Ley de Amdahl[15]. Estas son de gran ayuda para medir el rendimiento de nuestros sistemas y ajustar nuestros algoritmos a las mejores configuraciones. Esta ley establece una cota superior para problemas donde al escalar el sistema, se mantenga fija la parte que puede ser realizada en paralelo, lo cual claramente no es lo más cercano a la realidad pues en la práctica estos dos factores suelen crecer en proporción uniforme.

1.4.3. Ley de Gustafson

John Gustafson, científico que ha dedicado especial empeño a la computación de alto rendimiento, estableció en 1988 una ley que propone como medir la mejora máxima que se puede esperar de un sistema paralelo mientras se escala. La principal diferencia con la Ley de Amdahl, es que a diferencia de esta, la Ley de Gustafson tiene en cuenta que en la práctica al escalar un sistema paralelo la parte que puede ser realizada en paralelo aumenta conjuntamente con el número de procesadores y la parte secuencial tienda a cero, siendo esta ley más optimista.

Bajo estas premisas pudieran replantearse las ecuaciones 6) y 7) como sigue:

$t_1 = n * S_p + N_p$ 15)

$t_n = S_p + N_p$ 16)

siendo t_1 el tiempo para un sistema con un solo procesador y t_n el tiempo para **n** procesadores. Esto responde a que en un sistema con un procesador, las partes paralelas aún son realizadas de manera secuencial y con varios procesadores los tiempos se solapan, lo cual constituye la esencia del procesamiento paralelo.

Según las ecuaciones 15) y 16), la ganancia de velocidad (speed up),descrita en 8), tomaría la forma:

$a = \frac{n*S_p+N_p}{S_p+N_p}$ 17)

la cual en términos de la función de cuello de botella no paralelizable (f) quedaría descrita como sigue:

$a = n - f(n-1)$ 18)

Y esta ecuación nos da una medida mucho más cercana a la realidad para medir la aceleración que

puede ser obtenida al escalar un sistema paralelo donde el número de partes paralelizables también crece. Las definiciones de la Ley de Amdahl y de la Ley de Gustafson no son contradictorias, sus premisas son diferentes, por tanto caracterizan situaciones diferentes.

1.5. Tecnologías para la comunicación

En el desarrollo de sistemas informáticos para procesamiento paralelo, juega un papel fundamental la eficiencia en las comunicaciones. Siendo en este caso la red un componente indispensable para el funcionamiento de aplicaciones paralelas y distribuidas, los protocolos o arquitecturas empleadas para las comunicaciones deben ser debidamente seleccionados según los objetivos que se desean perseguir. Dentro de la capa de más bajo nivel podemos encontrar librerías que nos permiten desarrollar la intercomunicación entre aplicaciones a través de redes. Podríamos citar por ejemplo la utilización de librerías para el trabajo con sockets sobre redes con paquetes TCP/IP o UDP sobre cualquier sistema operativo o plataforma. Si bien las comunicaciones sustentadas sobre estas son sumamente rápidas, se introducen problemas que debe solucionar el desarrollador como la compatibilidad entre diferentes plataformas de hardware (tamaño de palabra de 32 o 64 bits). Muchos protocolos están desarrollados sobre TCP/IP o UDP abstrayendo al desarrollador de las complejidades de su utilización. Cuando se desarrolla una aplicación para cómputo paralelo con PVM por ejemplo, la comunicación entre demonios[1] se realiza mediante UDP y la comunicación entre tareas sobre TCP/IP, esto se debe a que, aunque UDP es más rápido, no es orientado a conexión a diferencia de TCP/IP que sí lo es.

El desarrollo de middleware constituye una alternativa interesante cuando se desean implementar sistemas paralelos o distribuidos. Podríamos decir, desde cierto punto de vista, que tecnologías como PVM o MPI tienen características que implementa un middleware, debido a que abstraen al desarrollador de funcionalidades de bajo nivel pero el concepto middleware va más allá. Como regla general estas tecnologías (middleware) ofrecen una capa con muchas más funcionalidades para el desarrollo de aplicaciones distribuidas o de procesamiento paralelo como servicios de seguridad, nombres, transacciones, persistencia, notificaciones etc. unido a la abstracción de los tipos de redes, protocolos y plataformas de hardware subyacentes, lo cual hace mucho más universales y flexibles a las soluciones.

Las arquitecturas representativas para middleware son muy variadas en general todas con sus ventajas y desventajas que las hacen candidatas a resolver determinados problemas, no existe una solución ideal sino que a la hora de enfrentar proyectos de desarrollo, según las características del problema a

[1] Proceso que se ejecuta en cada nodo conformando la Máquina Virtual Paralela de PVM.

resolver se pasaría a elegir la más adecuada.

Podríamos comenzar citando dentro de las tecnologías para comunicación entre aplicaciones (estén o no distribuidas), a la tecnología RMI[24] diseñada específica y únicamente para Java que ofrece la funcionalidad de invocar métodos pertenecientes a aplicaciones remotas como si los mismos fueran locales abstrayendo al desarrollador de todo el hardware de red subyacente. Esta tecnología introduce la complicación de que no ofrece la posibilidad de interactuar con aplicaciones desarrolladas sobre otras plataformas diferentes a Java.

Otra tecnología es DCOM[13], desarrollada por Microsoft que se basa en el desarrollo de componentes de software distribuido para aplicaciones que se comunican entre sí. Aunque presenta grandes ventajas sobre otras arquitecturas solo puede desarrollarse sobre plataformas Microsoft concretamente en el sistema operativo Windows siendo como debe suponerse una tecnología propietaria.

Por otro lado podemos encontrar al protocolo XML-RPC[33] el cual se basa en la implementación de llamadas a procedimientos remotos usando XML para codificar los datos y HTTP como protocolo de transmisión de mensajes. Orientado principalmente a Internet brinda un conjunto de funcionalidades y servicios que hacen más fácil la comunicación.

También pensado principalmente para aplicaciones de Internet encontramos a SOAP que constituye la evolución de XML-RPC. Basa sus principios en el intercambio de datos mediante XML para manejar servicios web. Fue desarrollado por Microsoft e IBM.

Con características muy interesantes encontramos a CORBA que permite la invocación de métodos remotos en sistemas distribuidos bajo el paradigma de programación orientada a objetos. Con independencia de lenguaje o plataforma provee un conjunto de servicios que le confieren una gran ventaja sobre otros protocolos para aplicaciones de escritorio. Eso quiere decir, por ejemplo, que una aplicación CORBA desarrollada con C++ sobre GNU/Linux en cualquier plataforma hardware puede comunicarse sin ningún problema con otra implementada en Java sobre Windows o Mac sin importar incluso el protocolo de red (TCP/IP, IPX etc) subyacente.

A continuación podemos observar una tabla comparativa[17] donde se muestra el comportamiento de un conjunto de los protocolos descritos usando aplicaciones desarrolladas con el lenguaje de programación Python:

Cuadro 1.1: Comparación entre tecnologías de paso de mensajes.

Tecnología	Tc	Ec	Rc	Ec	Lc	Ls	TMC	TME
Sockets	0.002242	0.001377	0.00135	6.74067	57	25	2.279	85.863
CORBA	0.000734	0.004601	0.00218	1.52379	37	18	2.090	27.181
XML-RPC	0.007040	0.082755	0.05019	100.337	29	17	4.026	324.989
SOAP	0.000610	0.294198	0.27934	1324.29	32	10	4.705	380.288

Tc: Tiempo de conexión.
Ec: Tiempo enviando 21000 caracteres.
Rc: Tiempo recibiendo 22000 caracteres.
Ec: Tiempo enviando 5000 enteros.
Lc: LOC en el desarrollo del cliente.
Lc: LOC en el desarrollo del servidor.
TMC: Tamaño del mensaje actual enviando 1000 caracteres.
TME: Tamaño del mensaje actual enviando 100 enteros.

Realizando un análisis del comportamiento de los diferentes protocolos, vemos como la utilización de XML-RPC y SOAP permiten un desarrollo más rápido para aplicaciones sobre Internet, siendo SOAP la mejor opción en cuanto a velocidad de desarrollo y el menos favorecido de ambos en cuanto a rapidez de las aplicaciones. En otro extremo podemos encontrar las aplicaciones sobre Sockets como las más rápidas en la comunicación para el trabajo con caracteres, siendo CORBA más rápido en el trabajo con números pero en ambos casos las diferencias son mínimas. La verdadera fortaleza de CORBA reside en la cantidad de servicios con que dispone la plataforma unido al nivel de abstracción que presenta para el desarrollo de aplicaciones multiplataforma y multilenguajes.

1.6. Tecnologías a utilizar en el desarrollo

1.6.1. Sistema Operativo

Distribución Debian GNU/Linux

Debian es un sistema operativo (S.O.) libre. Utiliza el núcleo Linux, pero la mayor parte de las herramientas básicas vienen del Proyecto GNU; de ahí el nombre GNU/Linux. Ofrece más que un S.O. puro, viene con más de 20000 paquetes, programas precompilados distribuidos en un formato que hace

más fácil su instalación. Cuenta con varios repositorios en el mundo y específicamente en la Universidad de la Ciencias Informáticas, sus actualizaciones se realizan diariamente.

1.6.2. Programación

1.6.2.1. Lenguaje

El lenguaje de programación C++ es uno de los más empleados en la actualidad. Se puede decir que C++ es un lenguaje híbrido, ya que permite programar tanto en estilo procedimental (como si fuese C), como en estilo orientado a objetos, como en ambos a la vez. Además, también se puede emplear mediante la programación basada en eventos para crear programas que usen interfaz gráfico de usuario.

Las principales ventajas que presenta el lenguaje C++ son:

- Difusión: al ser uno de los lenguajes más empleados en la actualidad, posee un gran número de usuarios y existe una gran cantidad de libros, cursos y páginas web dedicados a él.
- Versatilidad: C++ es un lenguaje de propósito general, por lo que se puede emplear para resolver cualquier tipo de problema.
- Portabilidad: C++ está estandarizado y un mismo código fuente se puede compilar en diversas plataformas.
- Eficiencia: C++ es uno de los lenguajes más rápidos en cuanto ejecución.
- Herramientas: existe una gran cantidad de compiladores, depuradores y librerías que permiten el uso de este lenguaje.

1.6.2.2. IDE

EasyEclipse CDT es un IDE basado en eclipse con un plugin para programar en entornos C/C++ con control de versiones a través de Subversion.

EasyEclipse es:

- Libre y open-source.
- Fácil de manipular y con excelentes herramientas para el desarrollo integrado.
- Simple de mantener, sin problemas de versiones ni dependencias.

1.6.2.3. Librerías para programación concurrente

Pthreads[22], desarrolladas por el proyecto GNU[12] permiten realizar programas con múltiples hilos de ejecución, ideales para desarrollar aplicaciones del tipo Cliente/Servidor. Poseen primitivas y mecanismos para la creación, sincronización y manipulación de manera general de hilos de ejecución.

1.6.2.4. Arquitectura para la comunicación

CORBA, Bajo la implementación OmniORB[11], disponible bajo licencia de software libre, posee un conjunto de funcionalidades y servicios estandarizados para la mayoría de las plataformas y lenguajes, teniendo un carácter universal.

1.6.3. Bases de Datos

PgCluster

Desarrollado sobre PostgreSQL, es un gestor de bases de datos distribuido que tiene implementadas funcionalidades de replicación parcial y total, balance de carga mediante el método de la replicación multimaestro. Estas características le confieren un elevado valor cuando se desea que los datos a consultar tengan una disponibilidad permanente con integridad en varias fuentes (nodos). Está disponible bajo licencia de software libre.

Librería Libpq

Libpq es la librería interfaz para PostgreSQL desde C, la constituyen un conjunto de funciones que permiten a los programas clientes pasar las consultas al servidor PostgreSQL y recibir los resultados de estas consultas. Es también el motor subyacente para muchas otras interfaces de aplicación a PostgreSQL, incluyendo las programadas con C + +, Perl, Python, Tcl y ECPG. Está disponible bajo licencia de software libre.

PgAdmin3

Cliente con entorno gráfico para la administración de gestores de bases de datos PostgreSQL. Con herramientas de configuración, prueba, entre otros, puede manejarse de manera fiable cualquier gestor. Está disponible bajo licencia de software libre.

1.6.4. Ingeniería y Gestión de Software.

Visual Paradigm

Excelente herramienta de modelación UML para el desarrollo de proyectos. Ofrece un conjunto amplio de funcionalidades con versiones para plataforma GNU/Linux. La Universidad de las Ciencias Informáticas posee una licencia par su utilización.

Control de versiones.

SVN, herramienta para el control de versiones de excelentes prestaciones y de fácil configuración como servidor (repositorio de control de versiones). Plugin de SVN cliente integrado a EasyEclipse CDT. Está disponible bajo licencia de software libre.

1.7. Conclusiones

En la primera parte de este capítulo se analizó la fundamentación de las arquitecturas paralelas existentes y sus diferentes clasificaciones de acuerdo a características concretas. En una segunda parte se analizan los sistemas informáticos para procesamiento paralelo y sus ventajas y desventajas permitiendo ubicarlas de acuerdo a sus prestaciones. Como parte de las tecnologías para cómputo paralelo y distribuido se abordó en una tercera parte sobre los protocolos y arquitecturas para la comunicación entre aplicaciones de este tipo y se realizó una comparación sobre las prestaciones de cada uno de ellos. En la parte final de este capítulo se describen las principales herramientas que serán utilizadas para el desarrollo, aplicando para su selección los criterios de soberanía tecnológica y prestaciones adecuadas. En el estudio realizado no se encontró una herramienta con las características deseadas para la Plataforma Génenesis. El Motor de Procesamiento de Peticiones Paralelas será desarrollado de forma íntegra en el lenguaje de programación C++ utilizando como sistema para la comunicación a la arquitectura CORBA y como gestor de bases de datos se utilizará el PgCluster.

Capítulo 2

Desarrollo del Motor de Procesamiento de Peticiones Paralelas

2.1. Introducción

En el presente capítulo se describen las características y los principales componentes del Motor de Procesamiento de Peticiones Paralelas (MPPP). Estos componentes se desarrollan respondiendo a los Requisitos Funcionales de la plataforma. Algunos de estos componentes son detallados a mayor profundidad resaltando su importancia. Se brinda especial atención al manejo de la concurrencia y la protección de los datos al establecerse en el modelo una solución que incluye programación concurrente mediante múltiples hilos de ejecución.

2.2. Requisitos funcionales

El Motor de Procesamiento de Peticiones Paralelas constituirá el núcleo de la plataforma Génesis. A continuación son enunciados los requisitos funcionales que debe cumplir el mismo para la plataforma.

1. Funcionar en modo máster:

    a) Permitir conexiones de esclavos autorizados.

    b) Consultar la base de datos en busca de tareas de cómputo.

    c) Asignar tareas de cómputo a los esclavos.

   d) Realizar balance de carga en la asignación de trabajos.

   e) Planificar subtareas en esclavos ante peticiones de planificación de algunos de estos.

   f) Replicar información de creación de subtareas a esclavos.

   g) Generar identificadores de subtareas.

2. Funcionar en modo esclavo:

   a) Conectarse al máster.

   b) Asumir tareas de cómputo asignadas por el máster.

   c) Reportar mal funcionamiento o terminación de subtareas al máster.

3. Enviar y recibir datos entre nodos y subtareas de cómputo de tipo:

   a) Caracter de 8 bits sin signo.

   b) Caracter de 8 bits con signo.

   c) Entero de 16 bits sin signo.

   d) Entero de 16 bits con signo.

   e) Entero de 32 bits con signo.

   f) Entero de 32 bits sin signo.

   g) Entero largo de 32 bits con signo.

   h) Entero largo de 32 bits sin signo.

   i) Entero de 64 bits sin signo.

   j) Entero de 64 bits con signo.

   k) Flotante de 32 bits.

   l) Flotante de 64 bits de doble precisión.

   m) Bolean de 8 bits.

   n) Cadena de caracteres.

4. Crear subtareas de cómputo en nodos.

5. Generar identificadores universales.

6. Ejecutar algoritmo de cómputo paralelo.

Los requisitos anteriores responden a una arquitectura cliente-servidor. El MPPP podrá funcionar en dos modos (máster y esclavo), en el primer caso actuará como el nodo rector: buscando, aceptando y distribuyendo tareas de cómputo. Como esclavo se comportará aceptando trabajos: conectándose al nodo máster en busca de tareas, ejecutando las mismas completamente siempre que esto sea posible y reportándose luego de cada terminación para que le sean asignados nuevos trabajos (subtareas). Profundizando en los requisitos se tiene que:

Al funcionar en modo máster debe:

- Permitir conexiones de esclavos autorizados:

  El servidor al ejecutarse debe permitir que clientes esclavos se registren para que le sean asignados trabajos (tareas de cómputo), esto debe ser realizado mediante hilos de ejecución independientes para que no se obstruya el funcionamiento restante del sistema y se obtenga una mejora en el rendimiento en nodos con más de un procesador o procesadores de más de un núcleo.

- Consultar la base de datos en busca de tareas de cómputo:

  El motor, para su funcionamiento, interactúa con una base de datos de sistema, donde persisten datos propios de la aplicación y dónde se localizan órdenes de ejecución de tareas de cómputo, estas órdenes deben ser consultadas y ejecutadas por el mismo.

- Asignar tareas de cómputo a los esclavos:

  Una vez consultadas las órdenes, si las mismas son de realización de cómputo de tareas, el nodo máster debe asignar estas a los nodos esclavos para que estos las realicen, entre todos, de forma paralela.

- Realizar balance de carga en la asignación de trabajos:

  Para la asignación de subtareas de cómputo el nodo máster debe realizar balance de carga para que el clúster de alto rendimiento no se vea descompensado. Esto quiere decir que un trabajo nuevo siempre será asignado al nodo que menos trabajo tenga.

- Planificar subtareas en esclavos ante peticiones de planificación de algunos de estos:

Los nodos esclavos pueden pedir la realización de subtareas de cómputo al nodo máster. El máster debe ser responsable de crear estas subtareas en otros esclavos y responder al nodo esclavo que la solicitó sobre su creación y su localización.

- Replicar información de creación de subtareas a los esclavos:

  Cuando una o varias subtareas son creadas, el máster debe replicar a todos los nodos del clúster la localización de las mismas. Las subtareas pueden contener algoritmos paralelos que impliquen relación entre más de una subtarea, replicar la localización de cada subtarea es de vital importancia para que cada nodo conozca en cada momento como comunicarse con cualquier nodo que contenga a cualquier subtarea.

- Generar identificadores de subtareas:

  Al generarse una subtarea de cómputo, el nodo máster es responsable de generar un identificador para esta subtarea que no este siendo utilizado por otra subtarea en ese instante, impidiendo de esta forma errores en la comunicación.

Al funcionar en modo esclavo debe:

- Conectarse al máster:

  Los nodos esclavos se deben comunicar una vez arrancado el sistema con el nodo máster para que este les asigne trabajos. Este comportamiento permite saber en todo momento la capacidad y el rendimiento total del clúster, pudiendo escalarse el sistema sin ningún problema.

- Asumir tareas de cómputo asignadas por el máster:

  Al conectarse al nodo máster, este les debe enviar tareas de cómputo que deben ser asumidas por el esclavo. Si este nodo no conociese como llevar a cabo esta tarea debe informarlo al nodo máster en ese instante.

- Reportar mal funcionamiento o terminación de subtareas al máster:

  Al recibir una orden de ejecución de una subtarea de cómputo, el nodo esclavo debe asumirla hasta su terminación. En cualquiera de los caso posibles para la terminación de la subtarea, el nodo debe informar al máster sobre el estado de la misma.

Envío y recepción de datos según los tipos descritos:

- En el desarrollo de algoritmos paralelos, es de vital importancia la existencia de funciones de envío y recepción de datos entre subtareas. Este envío debe ser transparente para el desarrollador de algoritmos paralelos. Los envíos y recepciones de datos entre subtareas de cómputo solo deben implicar el conocimiento por parte del programador de los identificadores de las subtareas orígenes y destino (elementos obtenidos también a través de funciones) sin importar la localización física dónde estas se encuentran en el clúster, esta localización es responsabilidad del MPPP.

Crear subtareas de cómputo en nodos:

- Al un nodo esclavo recibir una orden de creación de subtareas del MPPP máster o al estar funcionando el nodo máster permitiendo la creación de subtareas locales, se debe proceder a la creación de subtareas que respondan a cada petición, esto debe implicar la creación de un hilo de ejecución que atienda cada petición hasta su terminación.

Generar identificadores universales:

- Con el fin de evitar la ambigüedad entre los mensajes generados por los nodos, es de vital importancia que estos posean identificadores únicos. Los identificadores deben ser de 128 bits y universales, esto quiere decir que puedan ser generados cuantos se deseen en el instante que se desee y jamás deben generarse dos iguales.

2.3. Componentes del motor

Los componentes del MPPP responden a los requisitos funcionales de la plataforma Génesis. En el desarrollo de la propuesta de solución se realizó la modelación de los mismos a través de componentes de software reutilizables para cada modo de ejecución del motor (máster o esclavo). Esto es así pues se diseñó el motor para funcionar en cualquiera de los dos modos, dependiendo esto solo de la configuración deseada para cada nodo. Como regla general, en el clúster deberá tenerse al menos un nodo máster y **n** nodos esclavos por cada máster.

Cada uno de los componentes descritos en los apartados siguientes describen a que modo pertenecen y cual es su función en el mismo. En su mayoría pertenecen a los dos modos, en cualquier caso será descrito su comportamiento dentro de cada modo.

Describiendo el funcionamiento general esperado de la plataforma Génesis, podemos decir primeramente que fue diseñada para realizar procesamiento paralelo de datos como objetivo principal. Las

tareas de cómputo son ordenadas a través de una aplicación Web que localiza en la base de datos del sistema, "genesis", las peticiones. Estas peticiones son consultadas y procesadas por el motor (modo máster), quién las convierte en tareas de cómputo. Por cada tarea de cómputo son creadas $\boldsymbol{k}$ subtareas en $\boldsymbol{x}$ nodos que realizan cada una una porción de la tarea de manera simultánea, obteniéndose al unir todos los resultados la solución a dicha tarea de manera más rápida. Los valores de $\boldsymbol{k}$ y $\boldsymbol{x}$ dependen de la cantidad de subtareas concebidas en el algoritmo paralelo y la cantidad de nodos disponibles en el clúster respectivamente. Estos elementos son combinados y distribuidos mediante un algoritmo de balance de carga localizado en el nodo máster el cual se encarga de distribuir la carga de manera uniforme según las cargas individuales de cada nodo, compensando de esta forma el clúster. Por otra parte los resultados parciales son insertados por los nodos esclavos directamente en la base de datos (clúster de base de datos), solo se le notifican al nodo máster el estado de terminación de las subtareas.

2.3.1. Explorador de solicitudes

El "Explorador de solicitudes" es el encargado de interactuar con la base de datos en busca de nuevas peticiones de procesamiento. El motor invocará al explorador de solicitudes en modo máster, cuando dentro del clúster haya disponibilidad en recursos para asumir nuevas tareas y en la base de datos "genesis" hayan solicitudes nuevas. Este componente responde al requisito funcional 1-b) del MPPP para la plataforma.

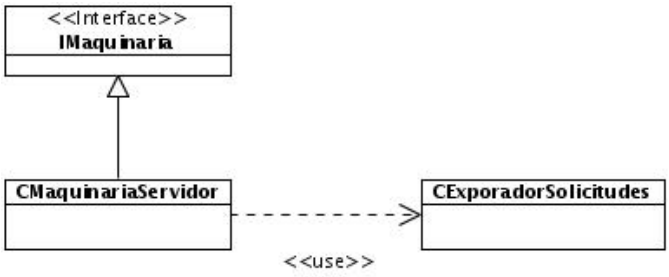

Figura 2.1: Diagrama de clases del Explorador de solicitudes.

En la *figura 2.1* se representa el diagrama de clases del "Explorador de solicitudes", puede observarse como este solo es utilizado por la maquinaria servidor, lo cual indica que solo es utilizado en el modo máster. A continuación quedan descritos cada uno de los atributos y métodos que contiene el Explorador:

Nombre de la clase: CExploradorSolicitudes	
Descripción: Clase que encapsula el comportamiento del Explorador de solicitudes	
Atributos	Tipo
ExplorerPtr	*static CExploradorSolicitudes**
Responsabilidades de la clase	
Nombre:	*CExploradorSolicitudes()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CExploradorSolicitudes()*
Descripción:	Destructor de la clase.
Nombre:	*static CExploradorSolicitudes* GetExploradorSolicitudes()*
Descripción:	Método estático, patrón Singleton, devuelve la única instancia.
Nombre:	*CSolicitud* GetSolicitudesNuevas(LLI64 pComplejidadPermitida)*
Descripción:	Devuelve una solicitud si existe con complejidad menor que pComplejidadPermitida.
Nombre:	*void CambiarEstadoSolicitud(LI32 pIdSolicitud, LI32 pIdEstado)*
Descripción:	Cambia el estado de la solicitud "pIdSolicitud" a estado "pIdEstado".
Nombre:	*LLI64 ObtenerComplejidadTotalEnProc()*
Descripción:	Obtiene la complejidad total de las solicitudes en procesamiento.

Cuadro 2.1: Atributos y responsabilidades del Explorador de solicitudes.

Como elementos a resaltar, podemos decir que a esta clase se le aplicó el patrón Singleton, de ella solo puede crearse una sola instancia, la cual puede ser accesible mediante el método "*static CExploradorSolicitudes* GetExploradorSolicitudes()*". El método más importante contenido es "*CSolicitud* GetSolicitudesNuevas(LLI64 pComplejidadPermitida)*", este contiene la funcionalidad principal del Explorador: buscar en la base de datos una solicitud cuya complejidad sea menor que "*pComplejidadPermitida*". La complejidad permitida es calculada por el motor en modo máster a partir de la carga que puede asumir el clúster tomando como base la carga contenida en ese instante por cada nodo. Si este método retorna una excepción indicando que no pudo obtener una solicitud, el motor volverá a realizar la petición en otro momento, en el cual la carga sea menor o existan solicitudes de menor complejidad.

2.3.2. Controlador de carga

El "Controlador de carga" es el componente encargado de realizar balance en la asignación de subtareas de cómputo a los nodos. Cuando una nueva solicitud es encontrada mediante el "Explorador de solicitudes" esta debe ser planificada como tarea máster en uno de los nodos del clúster, para realizar esta tarea el controlador de carga examina la carga de todos los nodos e intenta planificar esta subtarea

máster en el nodo menos cargado, si no ha sido posible realizarla en este nodo, la intenta planificar en el siguiente menos cargado y así sucesivamente. Si la tarea no pudo ser planificada en ningún nodo se queda como pendiente para otro momento en el cual el clúster esté menos cargado. Las tareas de control de carga son encapsuladas en el "Controlador de carga". Este componente es utilizado por el motor en modo máster. Este componente responde al requisito funcional 1-d) de la plataforma.

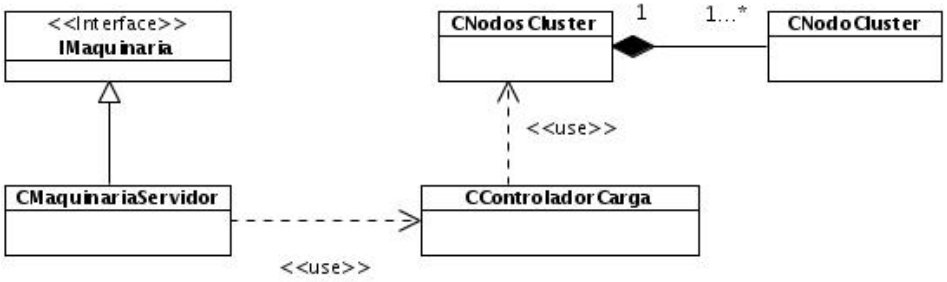

Figura 2.2: Diagrama de clases del Controlador de carga.

Como puede apreciarse en la *figura 2.2,* el controlador de carga utiliza una colección de instancias de tipo "CNodo", cada una de las cuales contiene información de la cantidad de subtareas creadas en cada uno de los nodos y del rendimiento de cada uno. A partir de la información contenida en "CNodosCluster" en un Árbol Binario Balanceado de Búsqueda, puede obtenerse el nodo menos cargado en el extremo izquierdo (nodo con menos peso) en orden constante. El peso de cada nodo es calculado de acuerdo al rendimiento del mismo y de la cantidad de subtareas que este posee.

Nombre de la clase: CControladorCarga	
Descripción: Clase que encapsula el comportamiento del Controlador de carga	
Atributos	Tipo
ControladorCargaPtr	*static CControladorCarga**
vSemControladorCargal	*static pthread_ mutex_ t*
Responsabilidades de la clase	
Nombre:	*CControladorCarga()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CControladorCarga()*
Descripción:	Destructor de la clase.
Nombre:	*CNodoCluster NodoMenosCargado()*
Descripción:	Devuelve el nodo menos cargado del clúster.
Nombre:	*void MarcarNoUtilizar(STR pNodo)*
Descripción:	Marca un nodo para no utilizar dentro del balance.
Nombre:	*void DesmarcarTodos()*
Descripción:	Desmarca todos los nodos que hayan sido marcados.
Nombre:	*static CControladorCarga* GetControladorCarga()*
Descripción:	Devuelve la única instancia de la clase, patrón Singleton.

Cuadro 2.2: Atributos y responsabilidades del Controlador de carga.

Dentro del "Controlador de carga" el método más importante es "*CNodoCluster NodoMenosCargado()*" que devuelve el nodo con menos carga dentro del clúster que pueda ser utilizado. En la implementación de este método se utiliza un Árbol Binario Balanceado de Búsqueda donde se almacena un nodo por cada nodo del clúster. Este método es de orden contante O(1) pues siempre se tendrá en el árbol binario como hijo más izquierdo al nodo menos cargado. A esta clase se le aplicó el patrón Singleton con el fin de garantizar que de esta sea creada una sola instancia.

2.3.3. Controlador de subtareas

El "Controlador de subtareas" encapsula el comportamiento de la creación y procesamiento de las subtareas, las cuales a su vez contienen algoritmos paralelos. Como funciones principales de este componente se destaca la creación de subtareas máster o esclavas, ordenadas por el nodo máster y la transmisión de datos a las mismas, enviados por otras subtareas locales o remotas necesarios para los algoritmos. Este componente responde a los requisitos funcionales 3, 4 y 6 del motor.

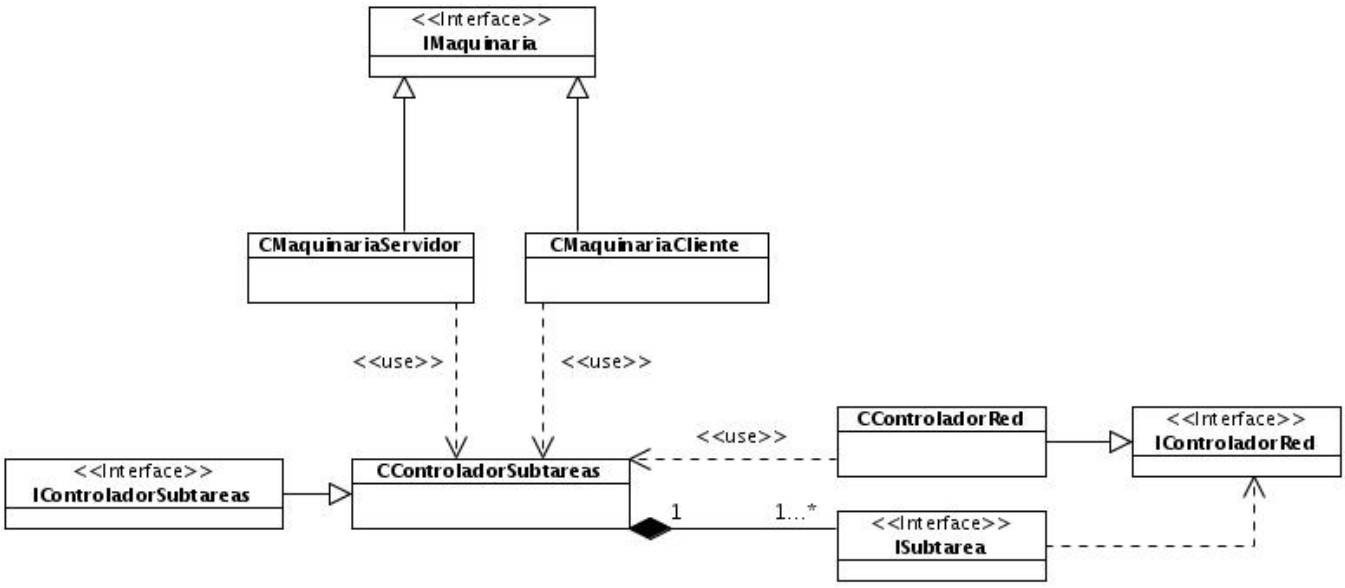

Figura 2.3: Diagrama de clases del Controlador de subtareas.

En la *figura 2.3* puede observarse que el controlador de subtareas contiene una colección de elementos "ISubtarea", estos elementos encapsulan una subtarea y son almacenados en un Árbol Binario Balanceado de Búsqueda para acceder a ellos en orden logarítmico. El "Controlador de red" utiliza al "Controlador de subtareas" ya que a través de la red llegan mensajes enviados de otras subtareas para las subtareas locales. Utilizado tanto en modo máster como en modo esclavo, encapsula todo el comportamiento de las subtareas. Es considerado por consiguiente en componente más importante. Controla la creación, sincronización, procesamiento y terminación de subtareas en un nodo, ya sea en modo máster o en modo esclavo. A continuación se realiza una descripción de los atributos y responsabilidades de la clase controladora "CControladorSubtareas":

Nombre de la clase: CControladorSubtareas	
Descripción: Clase que encapsula el comportamiento del Controlador de subtareas	
Atributos	Tipo
CantRealSubtareas	*ULI32*
ControladorSubtareas	*static CControladorSubtareas**
Subtareas	*set<CNodoSubtarea>*
SemControladorSubtareas	*static pthread_ mutex_ t*
IDHiloControladorSubtareas	*static pthread_ t*
Responsabilidades de la clase	
Nombre:	*CControladorSubtareas()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CControladorSubtareas()*
Descripción:	Destructor de la clase.
Nombre:	*ULI32 InstanciarSubtareaEsclavo(STR pAliaSubtarea)*
Descripción:	Intenta instanciar una subtarea esclavo de tipo "pAliaSubtarea"
Nombre:	*ULI32 InstanciarSubtareaMaster(STR pAliaSubtarea)*
Descripción:	Intenta instanciar una subtarea máster de tipo "pAliaSubtarea"
Nombre:	*static CControladorSubtareas* GetControladorSubtareas()*
Descripción:	Devuelve la única instancia de la clase, patrón Singleton.
Nombre:	*static void *HiloInstanciaSubtareaMaster(void *arg)*
Descripción:	Método de llamada de hilos de ejecución de subtareas máster.
Nombre:	*static void *HiloInstanciaSubtareaEsclavo(void *arg)*
Descripción:	Método de llamada de hilos de ejecución de subtareas esclavo.
Nombre:	*void EliminarSubtarea(ULI32 pIDST)*
Descripción:	Elimina la subtarea con identificador "pIDST" del nodo.
Nombre:	*void RecibirMensajeC8()*
Descripción:	Recibe un mensaje de tipo C8 para una subtarea local.
Nombre:	*void RecibirMensajeUC8()*
Descripción:	Recibe un mensaje de tipo UC8 para una subtarea local.

Cuadro 2.3: Atributos y responsabilidades del Controlador de subtareas.

Las funciones más importantes de esta clase la conforman los métodos "*ULI32 InstanciarSubtareaEsclavo(STR pAliaSubtarea)*", que intenta crear una subtarea esclavo, "*ULI32 InstanciarSubtareaMaster(STR pAliaSubtarea)*", que intenta crear subtareas máster y los métodos de recepción de mensajes principalmente utilizados por el "Controlador de red" como el "*void RecibirMensajeC8()*" o el "*void RecibirMensajeUC8()*" que reciben un mensaje de tipo caracter de 8 bits o enteros sin signo de 8 bits (0-255) respectivamente. En el caso de los métodos de recepción están soportados todos los tipos de datos

primitivos restantes (requisitos 3-a) al 3-n), los cuales fueron omitidos en la tabla.

2.3.4. Buffer de mensajes

El "Buffer de mensajes" es el componente que permite evitar desbordamiento de memoria por envío no controlado de mensajes entre los nodos. Este caso puede darse si un algoritmo paralelo mal diseñado es puesto en ejecución. Supongamos, por ejemplo, que el algoritmo de una subtarea comienza a enviar de manera desproporcionada mensajes a otra subtarea local o remota, entonces estos mensajes son acumulados en la memoria RAM hasta colapsar. El "Buffer" tiene como objetivo resolver este problema mediante el *paginado* que no es más que eliminar los mensajes *sobrantes* de la memoria RAM y enviarlos a la base de datos hasta que estos necesiten ser consultados. Muchas veces en el diseño de un algoritmo paralelo no puede ser evitada la acumulación de mensajes, sea ese el caso o no, el motor debe estar preparado para enfrentarlo.

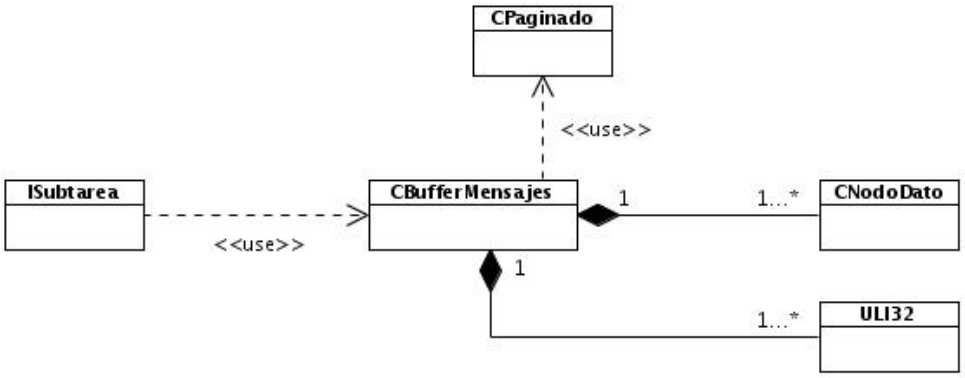

Figura 2.4: Diagrama de clases del Buffer de mensajes.

El "Buffer" utiliza la clase "CPaginado" la cual constituye la interfaz entre el motor y la base de datos para paginar aquellos mensajes *"sobrantes"*. En la figura puede apreciarse la relación que se establece con la clase "CNodoDato" conteniendo una colección de instancias de esta clase en un Árbol Binario Balanceado de Búsqueda que permite inserción de elementos repetidos (multiset). Esto permite introducir datos iguales en la estructura, ya que dos o más mensajes enviados entre subtareas no tienen por que ser necesariamente diferentes siempre. Aún cuando hay elementos repetidos, esta estructura garantiza el acceso a los elementos en orden logarítmico. A continuación se describen los principales atributos y responsabilidades de la clase controladora "CBufferMensajes":

Nombre de la clase: CBufferMensajes	
Descripción: Clase que encapsula el comportamiento del Buffer de mensajes	
Atributos	Tipo
MaxEspacioRAM	*ULI32*
EspacioRealRAM	*ULI32*
CantRealMensajes	*ULI32*
DatosPtr	*multiset<CNodoDato>**
EtiquetasEnPaginadoPtr	*set<ULI32>**
Responsabilidades de la clase	
Nombre:	*void ActualizarPaginado(CNodoDato pNodoDato)*
Descripción:	Actualiza el nodo que debe ser paginado si es necesario.
Nombre:	*void AdicionarDato(void* pDatoPtr)*
Descripción:	Adiciona un dato genérico al buffer de mensajes.
Nombre:	*void* ExtraerDato()*
Descripción:	Extrae un dato genérico del buffer de mensajes.
Nombre:	*CBufferMensajes()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CBuferMensajes()*
Descripción:	Destructor de la clase
Nombre:	*ULI32 GetEspacioRealRAM()*
Descripción:	Devuelve el valor total ocupado en RAM por el buffer.
Nombre:	*ULI32 GetCantRealMensajes()*
Descripción:	Devuelve la cantidad real de mensajes almacenados en el buffer.
Nombre:	*void AdicionarC8(C8 pDato)*
Descripción:	Adiciona un dato de tipo C8 (caracter de 8 bits) al buffer.
Nombre:	*void AdicionarLI32(LI32 pDato)*
Descripción:	Adiciona un dato de tipo LI32 (entero de 32 bits) al buffer.
Nombre:	*LI32 ExtraerLI32()*
Descripción:	Extrae un dato de tipo LI32 del buffer.

Cuadro 2.4: Atributos y responsabilidades del Buffer de mensajes.

Dentro de las responsabilidades más importantes del "Buffer de mensajes" podemos incluir todos los "Adicionar" y "Extraer" que constituyen el principio del componente, estos incorporan y extraen los datos haciendo transparente el la inserción y la eliminación de la base de datos de los mensajes cuando es necesario paginar o recuperar un dato. Se encuentran soportados, en estas operaciones, todos los tipos de datos primitivos. Si durante la recuperación de un mensaje paginado hay problemas con la base de datos, el "Buffer" lo notificará mediante una excepción para que este mensaje sea consultado

luego cuando el problema sea resuelto.

2.3.5. Generador de identificadores de mensajes

El generador de identificadores de mensajes tiene el objetivo de generar identificadores universales de 128 bits únicos. Para lograr que la manipulación de los mensajes dentro del clúster se realice de manera correcta es necesario para un conjunto de mensajes que estos sean "marcados" con un identificador que no se repita con el paso del tiempo. Este identificador es construido con 128 bits lo cual garantiza mediante un algoritmo que utiliza varios elementos de la computadora como la hora, dirección MAC, instante de tiempo, entre otros, que no se repitan los identificadores incluso si son generados de manera paralela en un mismo nodo. A continuación se describe el diagrama de clases que representa este componente:

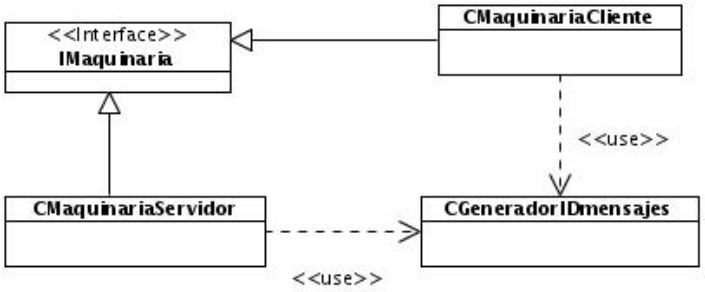

Figura 2.5: Diagrama de clases del Generador de identificadores de mensajes.

La "Maquinaria Servidor" y la "Maquinaria Cliente" utilizan el "Generador de identificadores de mensajes" al construir mensajes que pudieran ser enviados por la red o paginados. Para la construcción de los identificadores se reutilizó la biblioteca de clases "uuid" bajo licencia de software libre. Una descripción de los atributos y las responsabilidades de la clase controladora "CGeneradorIDmensajes" queda ilustrada en el siguiente cuadro:

Nombre de la clase: CGeneradorIDmensajes	
Descripción: Clase que encapsula el comportamiento del Generador de identificadores de mensajes	
Atributos	Tipo
GeneradorIDmensajesPtr	*static CGeneradorIDmensajes**
vSGeneradorIDmensajesg	*static pthread_ mutex_ t*
Responsabilidades de la clase	
Nombre:	*CGeneradorIDmensajes()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CGeneradorIDmensajes()*
Descripción:	Destructor de la clase.
Nombre:	*static CGeneradorIDmensajes* GetGeneradorIDmensajes()*
Descripción:	Devuelve la única instancia de la clase, patrón Singleton.
Nombre:	*UC8* GenerarID()*
Descripción:	Devuelve identificadores universales únicos de 128 bits.
Nombre:	*B8 Comparar(UC8* pID1, UC8* pID2)*
Descripción:	Compara identificadores universales únicos bit a bit.

Cuadro 2.5: Atributos y responsabilidades del Generador de identificadores de mensajes.

La función más importante de la clase controladora "CGeneradorIDmensajes" es "*UC8* GenerarID()*" que devuelve una secuencia de 16 elementos de tipo UC8 (entero sin signo de 8 bits) en cada llamada. Esta secuencia combinada constituye un identificador universal único. Estas funcionalidades se corresponden con el requisito funcional 5) del motor.

2.3.6. Generador de identificadores de subtareas

Al ordenar la creación de subtareas de cómputo se hace necesario, para el desarrollo de algoritmos paralelos, que estas tareas posean un identificador. Cuando una subtarea de cómputo es creada, las funciones de creación retornan un identificador el cual debe persistir con carácter único, durante la existencia de esta subtarea. Este identificador es un entero sin signo de 32 bits, único en cada instante de tiempo en el clúster. La generación de un identificador nuevo implica que este no esté siendo utilizado por ningún nodo. Cuando una subtarea es terminada, de manera correcta o incorrecta, el identificador de la misma podrá ser reutilizado al replanificar esta o al crear nuevas subtareas. La responsabilidad de que estos identificadores sean generados de manera correcta y actualizados cuando sea necesario le corresponde al "Generador de identificadores de subtareas". Este componente se corresponde con el requisito funcional 1-g) del motor. En el siguiente diagrama de clases se ilustra la relación que guarda

este componente con los demás componentes del motor:

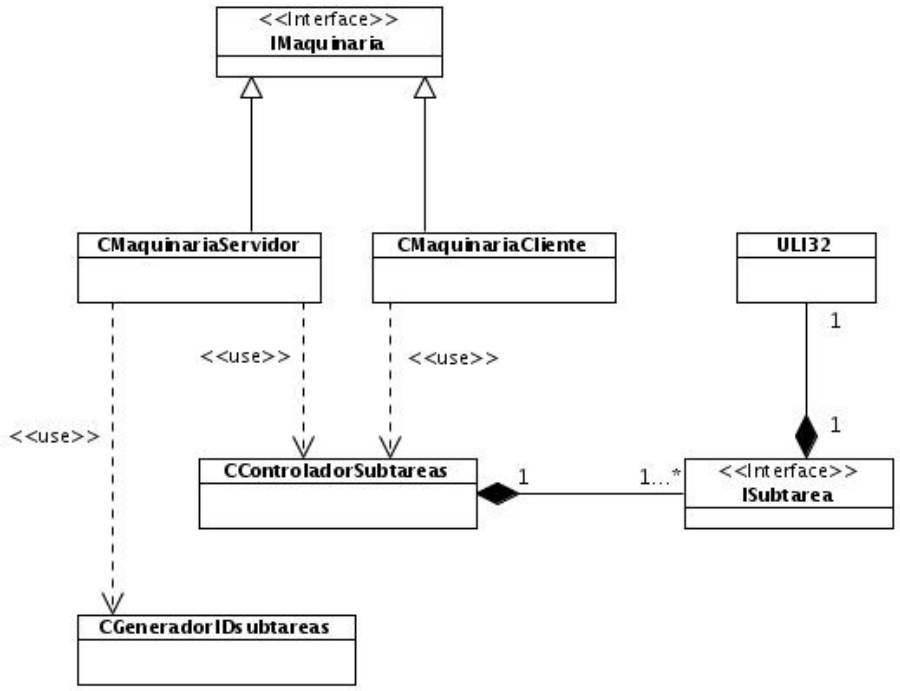

Figura 2.6: Diagrama de clases del Generador de identificadores de subtareas.

Como se ilustra en la *figura 2.6* el motor en modo esclavo (Maquinaria Cliente) o en modo máster (Maquinaria Servidor), utilizan al controlador de subtareas el cual a su vez crea y contiene una colección de subtareas, cualquiera que sea el nodo. Solo el motor máster puede generar un identificador de una subtarea y este es el responsable de hacerlo llegar al motor esclavo que lo solicite. A continuación quedan descritos, en el siguiente cuadro, los principales atributos y responsabilidades de la clase controladora "CGeneradorIDsubtareas":

Nombre de la clase: CGeneradorIDsubtareas	
Descripción: Clase que encapsula el comportamiento del Generador de identificadores de subtareas	
Atributos	Tipo
GeneradorIDSubtareasPtr	*static CGeneradorIDSubtareas**
UltimoIDAsignado	*ULI32*
Responsabilidades de la clase	
Nombre:	*CGeneradorIDSubtareas()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CGeneradorIDSubtareas()*
Descripción:	Destructor de la clase.
Nombre:	*static CGeneradorIDSubtareas* GetGeneradorIDSubtareas()*
Descripción:	Devuelve la instancia única de la clase, patrón Singleton
Nombre:	*ULI32 ObtenerIDSubtarea()*
Descripción:	Obtiene el próximo identificador disponible.
Nombre:	*ULI32 ObtenerUltimoIDAsignado()*
Descripción:	Obtiene el último identificador asignado.
Nombre:	*ULI32 DecrementarUltimoAsignado(ULI32 pUltimoAsignado)*
Descripción:	Decrementa y devuelve el último asignado dado los disponibles.
Nombre:	void ActualizarID(B8 pEstado, ULI32 pID)
Descripción:	Actualiza un identificador con un estado de disponibilidad

Cuadro 2.6: Atributos y responsabilidades del Generador de identificadores de subtareas.

La función más importante de esta clase es "*ULI32 ObtenerIDSubtarea()*", la cual a partir de todos los identificadores utilizados y los disponibles, devuelve el identificador más pequeño no utilizado. Esta función, en el peor de los casos, puede tener complejidad temporal O(n) siendo $\boldsymbol{n}$ la cantidad de identificadores utilizados.

2.3.7. Localizador de subtareas

El "Localizador de subtareas" es el encargado de controlar la identificación y localización de todas las subtareas del clúster. Dentro de la concepción de un algoritmo paralelo se incluye que las subtareas de cómputo puedan interactuar entre sí a través del paso de mensajes. Para lograr este objetivo cada nodo, máster o esclavo, debe poseer la suficiente información actualizada del lugar (nodo) donde se encuentra cada subtarea. Cuando un nodo solicita la creación de una o varias subtareas producto de la ejecución de un algoritmo, esta solicitud es atendida por el nodo máster, los identificadores son generados y las subtareas correspondientes son creadas en un conjunto de nodos de acuerdo a la carga de los mismos.

Antes de enviar la respuesta al nodo que ha solicitado la creación de subtareas, es replicada a cada nodo la ubicación de las subtareas creadas. Por último es enviada al nodo que realizó la petición, la información que contiene los identificadores de las subtareas creadas. Este mecanismo presenta este rigor con el objetivo de lograr integridad en la ejecución y la sincronización de los algoritmos paralelos. Sin los identificadores de las subtareas creadas, un algoritmo paralelo no puede funcionar, por tanto, un algoritmo no recibirá los identificadores solicitados hasta tanto no se hayan creado correctamente las subtareas correspondientes y esta información no esté consistente en todos los nodos del clúster. El siguiente diagrama de clases ilustra como se encuentra relacionado el "Localizador de subtareas" con los demás componentes del motor:

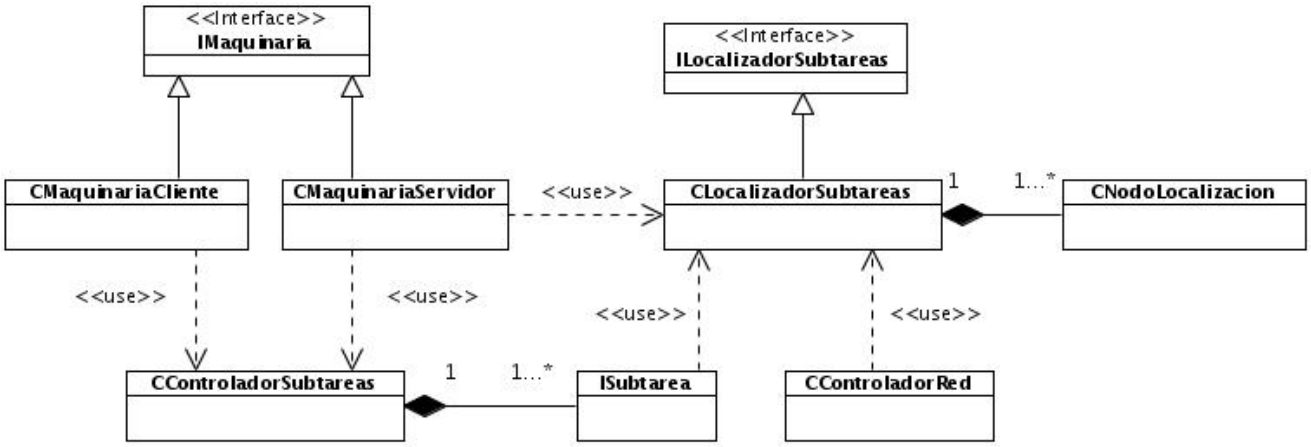

Figura 2.7: Diagrama de clases del Localizador de subtareas.

Como elemento importante a resaltar en el diagrama es la utilización del "Localizador de subtareas" por el "Controlador de red", esto es así porque en un nodo pueden coexistir subtareas que se intercambien mensajes. Antes de enviar un mensaje por un servicio de red, es comprobado que la subtarea destino no esté planificada localmente, aunque funcionaría igualmente a través de los servicios de red, se vería afectado el rendimiento al no entregar el mensaje directamente por la misma aplicación sin que intervengan los mismos. Localizar una subtarea de cómputo es tarea primordial de nodos máster y esclavos. A continuación se describen los principales atributos y responsabilidades de la clase controladora "CLocalizadorSubtareas":

Nombre de la clase: CLocalizadorSubtareas	
Descripción: Clase que encapsula el comportamiento del Localizador de subtareas	
Atributos	Tipo
NombrePClocal	*STR*
Localizador	*static CLocalizadorSubtareas**
NodosTareas	*set<CNodoLocalizacion>**
vSLocalizadorSubtareasg	*static pthread_ mutex_ t*
Responsabilidades de la clase	
Nombre:	*CLocalizadorSubtareas()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CLocalizadorSubtareas()*
Descripción:	Destructor de la clase.
Nombre:	*static CLocalizadorSubtareas* GetLocalizador()*
Descripción:	Devuelve la única instancia de la clase, patrón Singleton.
Nombre:	*void EliminarSubtarea(ULI32 pIDST)*
Descripción:	Elimina la subtarea "pIDST" del localizador.
Nombre:	*B8 EsSubtareaLocal(ULI32 pIDST)*
Descripción:	Retorna verdadero si "pIDST" es local.
Nombre:	*B8 EsSubtareaRemota(ULI32 pIDST)*
Descripción:	Retorna verdadero si "pIDST" es remota.
Nombre:	*void AgregarSubtarea(ULI32 pIDST, STR pNombreNodo)*
Descripción:	Añade un subtarea nueva al localizador.
Nombre:	*STR NombreNodoPorIDST(ULI32 pIDST)*
Descripción:	Devuelve la ubicación de una subtarea en el clúster.

Cuadro 2.7: Atributos y responsabilidades del Localizador de subtareas.

Dentro de las funciones más importantes se encuentra "*STR NombreNodoPorIDST(ULI32 pIDST)*", que realiza una búsqueda logarítmica dentro de la colección de identificadores de subtareas que posee y devuelve el nombre del nodo donde se encuentra. Dado que esta función es utilizada constantemente tanto en inserción, búsqueda y eliminación, estas tareas son ejecutadas todas con complejidad temporal logarítmica.

2.3.8. Controlador de red

El "Controlador de red" es uno de los componentes más importantes en la eficiencia del motor. Dentro de sus responsabilidades se encuentra la de hacer llegar de la manera más rápida los datos

(mensajes) a los diferentes nodos y que los mismos sean entregados de forma íntegra luego de pasar a través de la capa de red. Dentro de este componente se desarrollaron tres servicios servidores y tres servicios clientes, utilizándose cada uno de ellos en dependencia de las necesidades del motor. Así queda definido un servicio servidor "Máster-Esclavos" cuya implementación es utilizada por el motor en modo esclavo, a su vez existe un servicio para el flujo contrario: servicio cliente "Máster-Esclavos" cuya implementación es utilizada por el motor en modo máster. Cuando el máster emite una orden a un esclavo está asumiendo el rol de cliente, consumiendo el servicio servidor publicado en el esclavo a quien le emite la orden. De igual forma existe un servicio cliente y otro servidor "Esclavos-Máster" y un servicio servidor y cliente de envío de datos entre nodos ya sea entre máster y esclavos o entre esclavos.

El siguiente diagrama de clases ilustra la relación que guarda el "Controlador de red" con los restantes componentes del motor y los diferentes servicios:

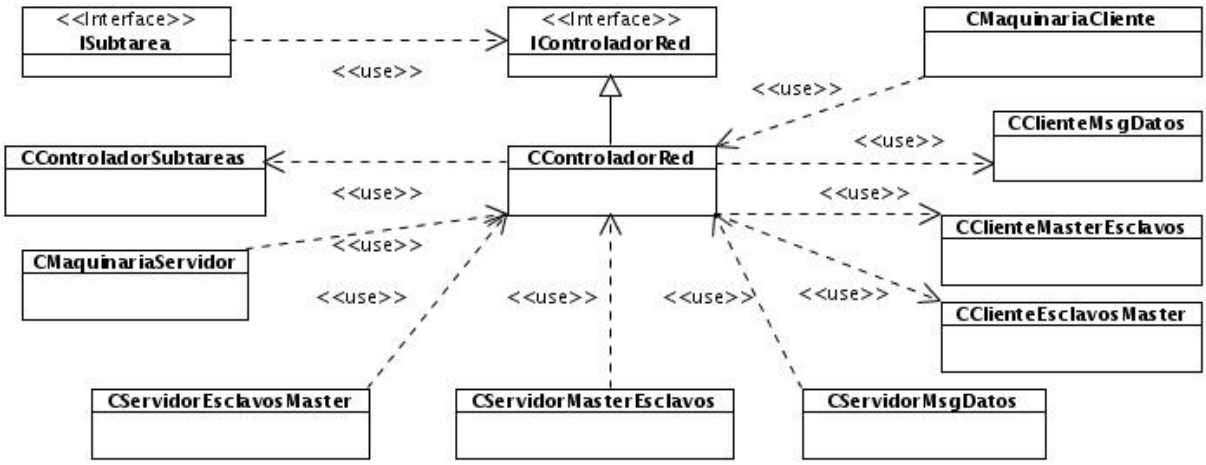

Figura 2.8: Diagrama de clases del Controlador de red.

En la *figura 2.8* se ilustra como el "Controlador de red" utiliza los servicios clientes para comunicarse con servicios servidores publicados en los nodos y es utilizado por los servicios servidores cuando otros nodos envían información. Estos servicios están desarrollados sobre la arquitectura CORBA, implementación OmniORB[11].

Cuando los controladores de red de diferentes motores entran en interacción a través de la red, pueden intercambiar información de manera paralela. No existe un flujo lineal para el intercambio de mensajes, esta implementación está desarrollada mediante hilos de ejecución lo cual garantiza el intercambio paralelo. Los controladores son capaces de sincronizarse para evitar condiciones de bloqueo, que en un clúster pueden tener no solo carácter local sino distribuido. Así una condición de "bloqueo

distribuido" es resuelta mediante una "sincronización distribuida" con sincronización para múltiples hilos de ejecución distribuidos.

A continuación quedan descritos los principales atributos y responsabilidades del "Controlador de red":

Nombre de la clase: CControladorRed	
Descripción: Clase que encapsula el comportamiento del Controlador de red	
Atributos	Tipo
Responsabilidades de la clase	
Nombre:	*CControladorRed()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CControladorRed()*
Descripción:	Destructor de la clase.
Nombre:	*static void RecibirLI32(LI32 pDato)*
Descripción:	Recibe un dato de tipo entero con signo de 32 bits a través de la red.
Nombre:	*static void OrdenarActualizarCreacionSubtarea()*
Descripción:	Ordena la actualización de creación de una subtarea en un nodo.
Nombre:	*static void OrdenarActualizarTerminacionSubtarea()*
Descripción:	Ordena la actualización de terminación de una subtarea en un nodo.
Nombre:	*static ULI32 OrdenarCrearSubtareaMaster()*
Descripción:	Ordena crear una subtarea máster en un nodo.
Nombre:	*static void EliminarSubtarea(ULI32 pIDST)*
Descripción:	Recibe orden de eliminación de una subtarea local.
Nombre:	*static ULI32 CrearSubtareas()*
Descripción:	Recibe orden de creación de subtareas de cómputo.

Cuadro 2.8: Atributos y responsabilidades del Controlador de red.

En el *cuadro 2.8* quedan descritas las principales funcionalidades del "Controlador de red", es de destacar que su flujo de datos es bidireccional: es el intermediario entre la capa de red y la capa de negocio y asume flujos de datos en ambos sentidos. Sus funciones van desde el intercambio de mensajes de datos entre subtareas de cómputo hasta el envío de órdenes del máster a los esclavos. Este componente es utilizado en cualquier modo (máster o esclavo). Responde a casi todos los requisitos funcionales del motor, solo basta para ello que implique para su desarrollo la utilización de la red.

2.3.9. Lector de parámetros de configuración

El "Lector de parámetros de configuración" es el componente encargado de cargar y verificar la configuración del motor y brindar elementos de configuración en cada momento a cualquier componente que lo solicite. Dentro de los elementos de configuración se incluyen diferentes secciones agrupadas por responsabilidades. Este componente fue desarrollado reutilizando la biblioteca de clases "libconfuse" que posee un conjunto de funcionalidades y macros que permiten definir un lenguaje para controlar la estructura de un fichero de configuración.

En el siguiente diagrama de clases se ilustra como se relaciona el "Lector de parámetros de configuración" con los demás componentes:

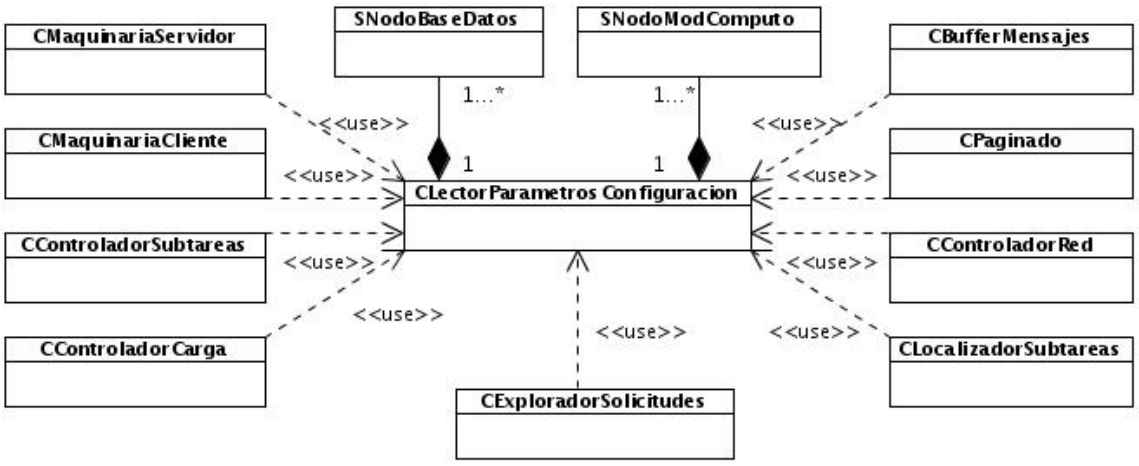

Figura 2.9: Diagrama de clases del Lector de parámetros de configuración.

La mayoría de los componentes utilizan el "Lector de parámetros de configuración". Los parámetros que este componente maneja son variables y constituyen elementos de configuración. Para determinados escenarios, estas variables pueden ser cambiadas para adecuar el motor al entorno. Si el "Lector de parámetros de configuración" encuentra un error grave en la configuración lo notificará y no permitirá que el motor sea iniciado. Por otro lado si un elemento de configuración es omitido se asumirá una configuración por defecto. Existen, como excepción, variables de configuración cuya omisión es considerada un error grave, en este caso siempre será realizada una notificación cuando sean omitidas.

A continuación se describen los principales atributos y responsabilidades de la clase controladora "CLectorParametrosConfiguracion":

Nombre de la clase: CLectorParametrosConfiguracion	
Descripción: Clase que encapsula el comportamiento del Lector de parámetros de configuración	
Atributos	Tipo
Lector	*static CLectorParametrosConfiguracion **
Modo	*STR*
NombrePcLocal	*STR*
TamanoBufer	*ULI32*
CantMaxSubtareas	*USI16*
Plataforma	*STR*
AutoProcesar	*STR*
RaizGenesis	*STR*
NombreNodoMaster	*STR*
PermitirSubtareasLocales	*STR*
BaseDatos	*set<SNodoBaseDatos>**
NodoServidorNombres	*STR*
Modulos	*set<SNodoModuloComputo>**
Responsabilidades de la clase	
Nombre:	*CLectorParametrosConfiguracion()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CLectorParametrosConfiguracion()*
Descripción:	Destructor de la clase.
Nombre:	*STR NombreDriverBDPorAlias(STR pAlias)*
Descripción:	Devuelve el nombre del driver de base de datos por un alias.
Nombre:	*void CargarConfiguracion()*
Descripción:	Carga la configuración de las variables desde fichero.
Nombre:	*IBaseDatos* GetBaseDatos(STR pDriver)*
Descripción:	Devuelve una base de datos por el nombre del driver.
Nombre:	*LI32 GetTamanoBuferMensajes()*
Descripción:	Devuelve el tamaño del buffer de mensajes.
Nombre:	*STR GetNombrePClocal()*
Descripción:	Devuelve el nombre del nodo que invoca la función.
Nombre:	*STR GetAutoProcesar()*
Descripción:	Devuelve si el motor está en procesamiento automático.

Cuadro 2.9: Atributos y responsabilidades del Lector de parámetros de configuración.

La función más importante de la clase "CLectorParametrosConfiguracion" es "*void CargarConfiguracion()*", que realiza un análisis del fichero de configuración e identifica, mediante el lenguaje definido, si existen errores en el mismo. De no encontrase errores, los valores de las variables son identificados y

convertidos a los tipos adecuados para que luego puedan ser consultados por los componentes del motor. La identificación de errores se realiza mediante el análisis del lenguaje definido para la estructura del fichero, si un error es encontrado se notifica el lugar del fichero donde ocurrió y una descripción del mismo.

2.3.10. Maquinaria

La maquinaria controla el comportamiento genérico del motor. Se tiene que el motor puede funcionar en modo máster o modo esclavo, para cada uno de estos modos existe una maquinaria. La "Maquinaria servidor" define el comportamiento para el nodo máster y la "Maquinaria cliente" define el comportamiento para los nodos esclavos. En el caso del nodo máster, la maquinaria tiene la responsabilidad de iniciar los servicios de red y demás componentes, ordenar la consulta de la base de datos en busca de solicitudes a procesar, utilizar el balance de carga para planificar subtareas, procesar solicitudes de los nodos esclavos y responder correctamente a estas, entre otras funciones de tipo máster.

La "Maquinaria cliente",por su parte, tiene la responsabilidad de iniciar los servicios de red y demás componentes, intentar unirse a un nodo máster para que este le asigne tareas, responder de manera "inteligente" ante fallos en los servicios de red y de base de datos o errores de credenciales al comunicarse con el nodo máster.

En el siguiente diagrama de clases queda ilustrada la relación de las maquinarias cliente y servidor con los demás componentes del motor:

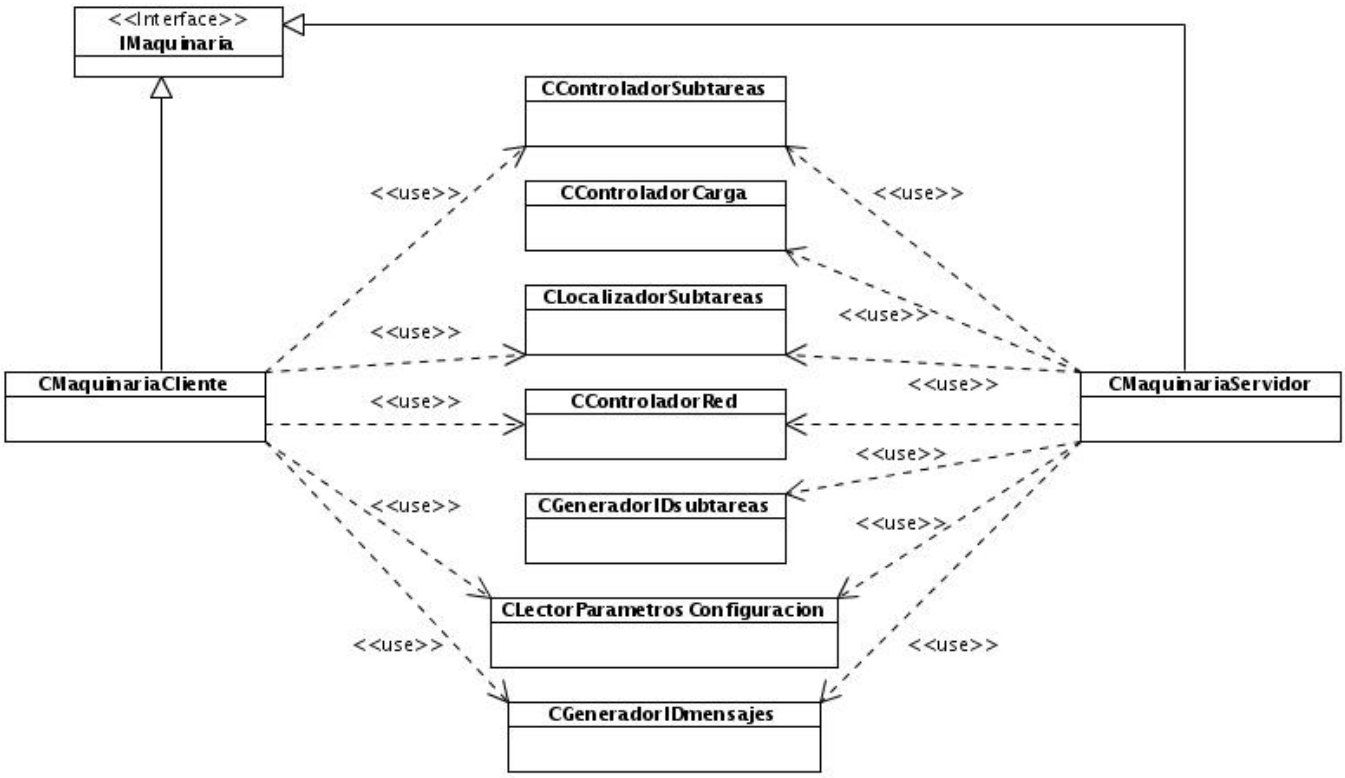

Figura 2.10: Diagrama de clases de las maquinarias Cliente y Servidor.

La maquinaria constituye la "inteligencia" del motor y define su modo de funcionamiento. Por esta razón guarda relación con todos los componentes, ya sea en modo máster o en modo esclavo.

En los siguientes cuadros se describen los principales atributos y responsabilidades de la clase controladora "CMaquinariaServidor":

Nombre de la clase: CMaquinariaServidor	
Descripción: Clase que encapsula el comportamiento de la Maquinaria servidor.	
Atributos	Tipo
MaquinariaServidorPtr	*static CMaquinariaServidor**
HaySolicitudes	*static B8*
CargaMaxima	*static LI32*
TerminarSolicitudes	*static B8*
TerminarChequeo	*static B8*
TerminarAdicionNodos	*static B8*
TerminarConsSolicitudes	*static B8*
NombrePClocal	*static STR*
NodosAIncorporar	*static queue<CNodoCluster> **
SubtareasACrear	*static queue<CNodoCreacionSubtareas>**
SolicitudesAprocesar	*static queue<CSolicitud>**
NodosCluster	*static set<CNodoCluster>**
vSNodosAIncorporarl	*static pthread_mutex_t*
vSSubtareasACrearg	*static pthread_mutex_t*
vSNodosClusterg	*static pthread_mutex_t*
vSCargaMaximag	*static pthread_mutex_t*
vSConsultarSolicitudesg	*static pthread_mutex_t*
vCIncorporarNodosClusterg	*static pthread_cond_t*
vCConsultarSolicitudesg	*static pthread_cond_t*
vCProcesarSolicitudesg	*static pthread_cond_t*
vCCambioCargaMaximag	*static pthread_cond_t*

Cuadro 2.10: Atributos de la Maquinaria servidor.

Nombre de la clase: CMaquinariaServidor	
Descripción: Clase que encapsula el comportamiento de la Maquinaria servidor	
Responsabilidades de la clase	
Nombre:	*CMaquinariaServidor()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CMaquinariaServidor()*
Descripción:	Destructor de la clase.
Nombre:	*static CMaquinariaServidor* GetMaquinariaServidor()*
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	*void AdicionarCreacionSubtareas()*
Descripción:	Adiciona una petición de creación de subtareas.
Nombre:	*void AdicionarNodoCluster()*
Descripción:	Intenta adicionar un nuevo nodo al clúster.
Nombre:	*void IniciarProcesamientoSolicitudes()*
Descripción:	Inicia el procesamiento de solicitudes mediante un hilo de ejecución.
Nombre:	*void IniciarProcesamientoNodos()*
Descripción:	Inicia el procesamiento de nodos mediante un hilo de ejecución.
Nombre:	*void IniciarChequeoNodos()*
Descripción:	Inicia el chequeo de vida de nodos mediante un hilo de ejecución.
Nombre:	*void IniciarConsultarSolicitudes()*
Descripción:	Inicia la consulta de solicitudes mediante un hilo de ejecución.
Nombre:	*static void* IncorporarNodosCluster(void *arg)*
Descripción:	Función principal para hilo de ejecución de incorporación de nodos.
Nombre:	*static void* ProcesarSolicitudes(void *arg)*
Descripción:	Función principal para hilo de ejecución de procesamiento de solicitudes.
Nombre:	*static void* ChequearNodosCluster(void *arg)*
Descripción:	Función principal para hilo de ejecución de chequeo de nodos.
Nombre:	*static void *ConsultarSolicitudes(void * pArg)*
Descripción:	Función principal para hilo de ejecución consulta de solicitudes.
Nombre:	*static void ReplicarCreacionSubtarea(ULI32 pIDST, STR pNodo)*
Descripción:	Función de replicación de subtareas en el clúster.
Nombre:	void Ejecutar()
Descripción:	Comienza la ejecución de la Maquinaria servidor.

Cuadro 2.11: Responsabilidades de la Maquinaria servidor.

En el siguiente cuadro se describen los principales atributos y responsabilidades de la clase controladora “CMaquinariaCliente”:

Nombre de la clase: CMaquinariaCliente	
Descripción: Clase que encapsula el comportamiento de la Maquinaria cliente.	
Atributos	Tipo
MaquinariaCliente	*static CMaquinariaCliente **
SemMaquinariaCliente	*static pthread_ mutex_ t*
Condicion	*static pthread_ cond_ t*
Responsabilidades de la clase	
Nombre:	*CMaquinariaCliente()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CMaquinariaCliente()*
Descripción:	Destructor de la clase.
Nombre:	*static CMaquinariaCliente * GetMaquinariaCliente()*
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	*void Ejecutar()*
Descripción:	Comienza la ejecución de la Maquinaria cliente.

Cuadro 2.12: Atributos y responsabilidades de la Maquinaria cliente.

Es importante señalar que el desarrollo de todos los servicios servidores y clientes fue realizado con soporte para múltiples hilos. Esta característica permite aprovechar el procesamiento paralelo en nodos que tengan más de un elemento de proceso.

2.3.11. Procesador

De todos los componentes, el procesador está localizado en el más alto nivel. Ya sea en modo máster o esclavo, el procesador solo contendrá una maquinaria, la cual a su vez puede ser cliente o servidor. Para el procesador, el tipo de maquinaria es un elemento abstracto, él solo emite órdenes a la maquinaria, en dependencia del tipo de maquinaria, esta se comporta de una forma o de otra. Estas formas de comportamiento definen la "inteligencia" del motor para un modo u otro.

En el siguiente diagrama de clases podemos observar la relación que se establece entre el procesador y las maquinarias:

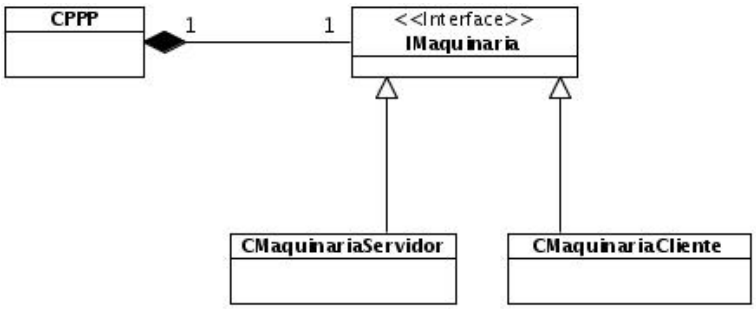

Figura 2.11: Diagrama de clases del Procesador de Peticiones Paralelas.

Dentro de las responsabilidades del procesador se encuentra la de comenzar la ejecución del motor, pausar esta o finalizarla. Estas son funciones genéricas que implican en su mayoría a todos los nodos del clúster si la maquinaria es de tipo servidor (nodo máster).

En el siguiente cuadro quedan descritos los principales atributos y responsabilidades de la clase controladora "CPPP", la cual define al procesador:

Nombre de la clase: CPPP	
Descripción: Clase que encapsula el comportamiento del procesador.	
Atributos	Tipo
ObjCPPP	static CPPP*
Maquinaria	IMaquinaria*
EstaProcesando	B8
HiloProcesador	static pthread_t
	Responsabilidades de la clase
Nombre:	CPPP()
Descripción:	Constructor de la clase.
Nombre:	virtual ~CPPP()
Descripción:	Destructor de la clase.
Nombre:	static CPPP* GetObjPPP()
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	void Procesar()
Descripción:	Comienza el procesamiento del procesador.
Nombre:	static void *HiloProcesarMaq(void *arg)
Descripción:	Función principal para hilo de procesamiento.
Nombre:	*void Ejecutar()*
Descripción:	Comienza la ejecución de la Maquinaria cliente.

Cuadro 2.13: Atributos y responsabilidades del procesador.

2.4. Manejo de la concurrencia

El motor se desarrolló para un óptimo aprovechamiento de los recursos de los nodos. Esto implicó trabajar constantemente con hilos de ejecución para entornos con múltiples elementos de proceso. La biblioteca de clases utilizada para la creación y manipulación de hilos fue "Posix Threads"[22] (pthread) de GNU.

Cuando varios hilos de ejecución son creados, estos pueden compartir las mismas direcciones de memoria, a diferencia de cuando son creados varios procesos. Esto, si bien constituye una ventaja considerable, introduce nuevos elementos a tener en cuenta durante el desarrollo, como también introduce nuevos tipos de errores. La complejidad reside en que para lograr que las variables compartidas sean accedidas y modificadas de manera correcta por varios hilos de ejecución, es necesario sincronizar a estos. La sincronización, por otra parte, es un proceso que no debe ser utilizado fuera de lo necesario pues nuestros programas pueden terminar siendo secuenciales.

Dentro del repertorio de primitivas de "pthread" encontramos funciones para la creación, sincronización y destrucción de hilos de ejecución. La sincronización se realiza principalmente a través de semáforos y esperas condicionales. Estas primitivas nos permiten diseñar algoritmos paralelos muy eficientes con hilos de ejecución en oposición a la utilización de procesos.

2.5. Conclusiones

En la primera parte de este capítulo se abordaron los principales requisitos funcionales del MPPP. En una segunda parte se analizan cada uno de los componentes del motor que se corresponden con las funcionalidades esperadas del mismo. Refleja vital importancia en el desarrollo del MPPP la utilización de hilos de ejecución en la manipulación de los servicios de red para garantizar la paralelización en el procesamiento y mejorar la eficiencia. Al finalizar este capítulo quedan descritas las responsabilidades de cada una de las partes de MPPP y su integración. Quedan cubiertos los objetivos trazados para la implementación de los principales componentes del MPPP para la Plataforma Génesis.

Capítulo 3

Comunicación cliente-servidor

3.1. Introducción

El presente capítulo profundiza en el subsistema de comunicación del Motor de Procesamiento de Peticiones Paralelas como base fundamental para la transmisión eficiente de los datos. Se describe la arquitectura CORBA utilizada en el desarrollo de los servicios de red, detallando la utilización del lenguaje descriptivo IDL para el desarrollo de aplicaciones distribuidas. Por otra parte son descritos todos los servicios de red que implementa el MPPP conjuntamente con la utilización del Servicio de Nombres "NameService" de CORBA para la utilización de objetos distribuidos.

3.2. CORBA como arquitectura para la comunicación

3.2.1. Arquitectura CORBA

La arquitectura CORBA define una infraestructura para la invocación de métodos remotos en entornos heterogéneos. Realiza una abstracción de la red en la invocación de los métodos de un objeto que se encuentra en una localización geográfica diferente como si este fuese local.

La portabilidad de CORBA es una de las características esenciales, así aplicaciones CORBA pueden comunicarse sin importar plataforma, sistema operativo o lenguaje en que hayan sido implementadas. Para la modelación de las interfaces de comunicación, utiliza el lenguaje descriptivo IDL, el cual constituye un estándar: no importa para que lenguaje sea compilado después; para el mismo servicio de red las interfaces modeladas serán siempre iguales. Las interfaces de los objetos están completamente

separadas de la implementación de los mismos.

El hecho de que esta arquitectura este pensada para entornos heterogéneos ofrece claras ventajas. No importa por ejemplo el protocolo de red subyacente, una aplicación CORBA será portable para la mayoría de estos (TCP/IP, IPX etc.), lo cual demuestra su robustez y eficiencia.

Para la utilización de CORBA se definen dos tipos de objetos: objetos clientes y objetos servidores, los cuales indican quien consume el servicio y quién lo brinda respectivamente. Aunque esta definición está en principio separada, CORBA abstrae los métodos de la red. Así, cuando es invocado un método local en un objeto cliente, este método es realizado en el objeto servidor aunque este esté localizado geográficamente distante.

3.2.2. Lenguaje IDL

El lenguaje IDL (Interface Definition Language) es un lenguaje puramente descriptivo, con este se definen las interfaces de los objetos compartidos, luego estas interfaces son compiladas al lenguaje que se desea y los compiladores IDL generan los archivos que son utilizados para implementar las responsabilidades de las clases generadas a partir de las interfaces, definiendo de esta manera el protocolo de comunicación.

Como características soportadas por lenguaje podemos citar las siguientes:

1. Declaración de módulos.
2. Declaración de interfaces.
    a) Declaración de atributos y funciones.
    b) Soporte para herencia.
3. Declaración de excepciones para las funciones.
4. Declaración de tipos de datos.
5. Declaración de constantes.
6. Preproceso para C++.
7. Identificadores *Case sensitive*.

8. Todos los tipos primitivos estándares.

9. Tipos construidos.

Declaración de módulos:

Los módulos permiten agrupar funcionalidades comunes, para declarar un módulo basta con encerrar las interfaces entre llaves precedido de la palabra reservada *module* y su identificador:

```
module enviodatos
{
}
```

al compilarse el archivo los módulos son traducidos a *namespaces* de C++.

Declaración de interfaces:

Las interfaces a su vez son declaradas dentro de los módulos y al compilarse se traducen en declaraciones de clases, a continuación se describen dos interfaces dentro del módulo anterior:

```
module enviodatos
{
    interface MasterEsclavos{};
    interface EsclavosMaster{};
}
```

estas interfaces pueden contener funciones y excepciones como puede verse a continuación:

```
module enviodatos
{
    exception CredencialesIncorrectas {};
    interface MasterEsclavos
   {
       void FuncionAutenticar(in short pIdentificador)
           raises (CredencialesIncorrectas);
   }
}
```

El modificador ***"in"*** dentro de la función "FuncionAutenticar" indica que en el flujo de comunicación este parámetro tendrá un solo sentido, esto quiere decir que será traducido a un parámetro constante

de C++ y no se podrá obtener datos de respuesta a través de él por referencia. Existen otros dos modificadores: ***"out"*** e ***"inout"*** los cuales son utilizados para parámetros que solo devuelven valores por referencia o para otros que realicen transmisión de datos en ambos sentidos respectivamente.

Por otro lado la función "***raises***" recibe como parámetros el identificador de una excepción lo cual indica que la función que le precede (FuncionAutenticar) puede lanzar una excepción de tipo "CredencialesIncorrectas" la cual también se encuentra declarada antes.

Tipos primitivos:

Dentro de los tipos primitivos se encuentran soportados los siguientes:

Tipo	Descripción
void	Tipo sin retorno
boolean	Verdadero o falso (true, false)
char	Caracter almacenado en 8 bits
wchar	Caracter almacenado en 16 bits
short	Entero con signo de 16 bits
unsigned short	Entero sin signo de 16 bits
long	Entero con signo de 32 bits
unsigned long	Entero sin signo de 32 bits
long long	Entero con signo de 64 bits
unsigned long long	Entero sin signo de 64 bits
float	Flotante de precisión simple (estandarizado por IEEE)
double	Flotante de precisión doble (estandarizado por IEEE)
octet	Paquete de 8 bits
any	Cualquier tipo

Cuadro 3.1: Tipos primitivos soportados por IDL.

Tipos construidos:

Los tipos construidos contribuyen a que nuestros programas estén mejor estructurados. En el lenguaje descriptivo IDL podemos declararlos de las siguientes formas:

enum mes {enero, febrero, marzo, abril, mayo, junio, julio};

Como en el lenguaje C++ "mes" constituye luego de la declaración anterior un tipo para declarar nuevas variables, las cuales pueden contener valores "enero", "febrero"..."julio". En la definición de las interfaces no se realizan inicializaciones con estos valores, siendo un lenguaje descriptivo solo se realizan

declaraciones, estas declaraciones son traducidas al lenguaje que se desee mediante un compilador IDL. Otra forma es la construcción de tipos mediante estructuras (en semejanza nuevamente a C++):

```
struct persona
{
    string Nombre;
    long Edad;
};
```

3.2.3. Compilación de las interfaces

Cuando los módulos y las interfaces son declarados, pueden ser compilados para el lenguaje que se desee. El archivo donde se declaran las interfaces debe tener extensión ".idl".

Supongamos que tenemos, en un archivo llamado "ProtocoloRed.idl", la siguiente interfaz:

```
interface FuncionesMasterEsclavos
{
    /*Definición de las excepciones*/
    exception CredencialesIncorrectas {};
    /*Definición de las funciones*/
    void PeticionCrearSubtareas( unsigned long pCantidadSubtareas);
    void PeticionUnionCluster()
      raises (CredencialesIncorrectas);
    void PeticionDesunionCluster(in string pNombreNodo);
};
```

para generar a partir de este archivo el código CORBA para C++, se utiliza cualquier compilador IDL disponible. En el desarrollo del motor se utilizó "omniidl4" que se encuentra disponible bajo licencia de software libre. Para compilar el archivo del ejemplo anterior para C++ basta con hacer en una consola:

eipad3:~# omniidl -bcxx ProtocoloRed.idl

luego de esto son generados dos archivos (ProtocoloRed.hh y ProtocoloRedSK.cc), los cuales contienen todas las especificaciones (código C++) para que a partir de ellos podamos construir nuestro protocolo de comunicación.

En el archivo "ProtocoloRed.hh", podemos localizar, entre otras cosas, una clase que representa al

patrón Proxy, generada a partir de la interfaz IDL. Para implementar el servicio servidor "FuncionesEsclavosMaster" debemos heredar de esta clase. Para el ejemplo de interfaz anterior, la clase Proxy es traducida como "POA_FuncionesMasterEsclavos" y contiene un método virtual puro por cada función declarada en la interfaz los cuales deben ser redefinidos en la clase que hemos construido derivada de "FuncionesMasterEsclavos". El concepto que debemos tener claro es que cuando un cliente consuma el servicio servidor y ejecute las funciones de un objeto de nuestra clase en una localización remota, será ejecutada la implementación que le hayamos dado a las funciones de nuestra clase para el servidor y no donde fue inicialmente realizada la llamada.

Por otro lado el archivo "ProtocoloRedSK.cc" contiene toda la implementación que ofrece la arquitectura CORBA para que la comunicación sea posible, segura y eficiente. Ninguno de los dos archivos deben ser modificados.

3.2.4. Servicio de nombres

CORBA no solo ofrece una metodología para desarrollar la comunicación, además ofrece un conjunto de servicios para que esta sea más eficiente, cómoda y segura. El servicio de nombres (NameService) ofrece la posibilidad de centralizar la información sobre la localización de los objetos distribuidos, lo que en pocas palabras significa centralizar la manera en que podemos utilizar los servicios servidores.

El servicio de nombres es instalado y configurado en un nodo. Cuando el servicio de nombres es arrancado, las aplicaciones CORBA residentes en los nodos pueden registrar sus servicios en este nodo. Cuando un nodo necesita comunicarse con otro, solo necesita consultar el nodo que posee el servicio de nombres en busca del servicio registrado del nodo destino. Con el nombre del servicio a consultar (pares llaves-tipo) se obtiene la localización del objeto distribuido y se consumen sus métodos.

Analizando este mecanismo puede comprobarse la efectividad en la comunicación a través de CORBA quién abstrae al desarrollador de aplicación de la complejidad de la red. Al desarrollar una aplicación CORBA es necesario iniciar los servicios adicionales que serán utilizados durante la ejecución. De esta forma se desarrolló el MPPP quién inicia los servicios de nombres de manera automática al ejecutarse la aplicación en todos los nodos ya sea en modo máster o esclavo. El siguiente fragmento de programa ilustra como es iniciado CORBA con el servicio de nombres localizado en el nodo "nodoservnombres.uci.cu":

1. **C8** * vArgvl[3];

2. **STR** vServiciol = "NameService=corbaname::nodoservnombres.uci.cu";

3. vArgvl[0]= **new char**[5];**strcpy**(vArgvl[0],"MPPP");

4. vArgvl[1]= **new char**[12];**strcpy**(vArgvl[1],"-ORBInitRef");

5. vArgvl[2]=**new char**[vServiciol.length()+1];

6. **strcpy**(vArgvl[2],vServiciol.c_str());

7. //Inicializar corba.

8. **int** vCantArgl = 3;

9. ORB = CORBA::**ORB_init**(vCantArgl, vArgvl);

La función "CORBA::**ORB_init**()" es l encargada de iniciar CORBA. Esta función recibe dos parámetros, el primero es la cantidad de argumentos (filas) que serán pasados en el segundo argumento que es una matriz de caracteres. El primer argumento es el nombre de la aplicación que en nuestro caso es "MPPP", el segundo argumento indica que tipo de servicio vamos a iniciar identificado por "-ORBInitRef" y el tercer argumento el Identificador Universal de Recursos del servicio: "NameService=corbaname::nodoservnombres.uci.cu". Cuando nuestra aplicación hace referencia al servicio de nombres en busca de la localización de un objeto distribuido, sabe que el mismo está localizado en "nodoservnombres.uci.cu".

3.3. Servicio de envío y recepción de datos

El servicio de envío y recepción de datos es utilizado por el motor, ya sea en modo máster o modo esclavo, en el paso de mensajes, como petición de los algoritmos en las subtareas de cómputo. Por la necesidad de que este sea un servicio rápido, es implementado por separado de los demás servicios. Para lograr que la comunicación sea bidireccional entre los nodos en la transmisión de mensajes de datos, tanto esclavos como máster deben tener implementados el cliente y el servidor de este servicio. Lo que indica que un máster o un esclavo asuma el rol de cliente o servidor CORBA es el flujo de la información: quién desea enviar un dato a otro nodo asume el rol de cliente CORBA y quién lo recibe el rol de servidor. En el siguiente diagrama de clases se ilustra como está relacionado el Controlador de red con el Servicio de envío y recepción de datos:

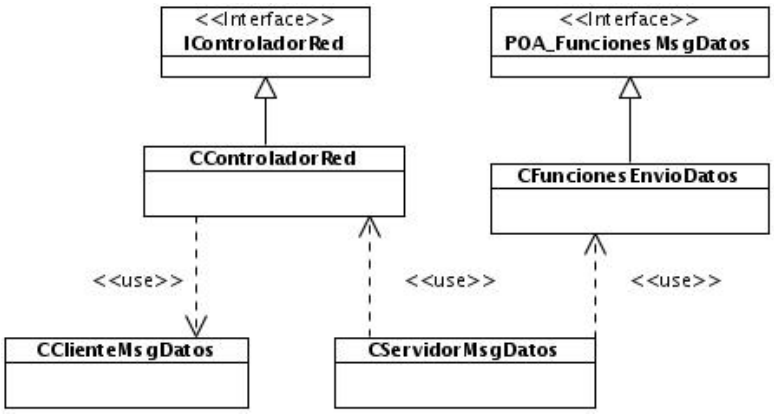

Figura 3.1: Diagrama de clases del Servicio de envío y recepción de datos.

En la *figura 3.1* puede observarse como, en el flujo de comunicación, el cliente "CClienteMsgDatos" es utilizado por el "Controlador de red" para enviar datos a otros nodos, mientras que el servicio servidor de envío de datos "CServidorMsgDatos" utiliza al controlador de red durante la recepción de mensajes a través de la red y esto constituye la capa de más bajo nivel en la manipulación de los mensajes, que es implementada por el desarrollador de aplicación.

A continuación queda descrita la clase controladora "CClienteMsgDatos" la cual constituye en si misma el servicio cliente para la transmisión de datos:

Nombre de la clase: CClienteMsgDatos	
Descripción: Clase que encapsula el comportamiento de servicio Cliente de envío de datos.	
Atributos	Tipo
ClienteMsgDatos	*static CClienteMsgDatos**
ContextoRaiz	*CosNaming::NamingContext_ var*
Responsabilidades de la clase	
Nombre:	*CClienteMsgDatos()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CClienteMsgDatos()*
Descripción:	Destructor de la clase.
Nombre:	*FuncionesEnvioMensajesDatos_ var ReferenciaAObjetoFunciones()*
Descripción:	Devuleve una referencia al objeto distribuido a partir del servicio de nombres.
Nombre:	*void IniciarServicio()*
Descripción:	Inicia los servicios clientes de envío de datos.
Nombre:	*static CClienteMsgDatos* GetClienteMsgDatos()*
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	*void EnviarC8(C8 pDato)*
Descripción:	Envía un caracter de 8 bits a un nodo remoto.
Nombre:	*void EnviarLI32(LI32 pDato)*
Descripción:	Envía un entero de 32 bits a un nodo remoto.
Nombre:	*void EnviarF32(F32 pDato)*
Descripción:	Envía un flotante de precisión simple a un nodo remoto.
Nombre:	*void EnviarSTR(STR pDato)*
Descripción:	Envía una cadena de caracteres a un nodo remoto.

Cuadro 3.2: Atributos y responsabilidades del Cliente de envío de datos.

Dentro de las funciones más importantes podemos destacar "*FuncionesEnvioMensajesDatos_ var ReferenciaAObjetoFunciones()*" la cual consulta el nodo servidor de servicio de nombres en busca de la referencia del objeto distribuido que se desea utilizar y la devuelve. Esta referencia es utilizada luego para enviar los datos al nodo remoto que se desee. Las funciones de envío de datos (ej. "*void EnviarF32(F32 pDato)*") son las encargadas de hacer llegar el dato al nodo distribuido de manera correcta y realizar tratamiento de excepciones sobre cualquier problema ocurrido durante el envío. Todos los tipos de datos primitivos están soportados para el intercambio de datos.

Nombre de la clase: CServidorMsgDatos	
Descripción: Clase que encapsula el comportamiento del servicio Servidor de recepción de datos	
Atributos	Tipo
ServidorMsgDatos	static CServidorMsgDatos*
ORB	*CORBA::ORB_var*
Sa_obj	*CORBA::Object_var*
IDHilo	*static pthread_t*
Responsabilidades de la clase	
Nombre:	*CServidorMsgDatos()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CServidorMsgDatos()*
Descripción:	Destructor de la clase.
Nombre:	*static void * FuncionHiloServidorMsgDatos(void * pArg)*
Descripción:	Función principal para hilo de escucha del servidor.
Nombre:	*static CServidorMsgDatos* GetServidor()*
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	*void IniciarServicio()*
Descripción:	Inicia el servicio servidor.
Nombre:	*void LiberarServicio()*
Descripción:	Libera el servicio servidor.
Nombre:	*CORBA::ORB_var GetORB()*
Descripción:	Devuelve la instancia ORB del servicio servidor.
Nombre:	*CORBA::Object_var GetSa_obj()*
Descripción:	Devuleve un objeto genérico CORBA.

Cuadro 3.3: Atributos y responsabilidades del Servidor de recepción de datos.

3.4. Servicio de órdenes del nodo máster a los esclavos

Cuando el motor en modo esclavo es iniciado, la primera tarea luego de iniciar los servicios clientes y servidores es la de comenzar el diálogo de contacto con el nodo máster. Luego de comprobadas las credenciales, el nodo es autorizado y se le encomiendan subtareas de cómputo. Las órdenes dadas por el motor máster a los esclavos son realizadas a través del "Servicio de órdenes del nodo máster a los esclavos". En este servicio a diferencia del "Servicio de envío y recepción de datos", el cliente CORBA solo será utilizado por el máster y el servidor CORBA solo será utilizado por los esclavos. El flujo de comunicación en este caso es unidireccional: de máster a esclavos. En el siguiente diagrama de clases

se representa como está relacionado el "Servicio de órdenes del nodo máster a los esclavos" con el "Controlador de red":

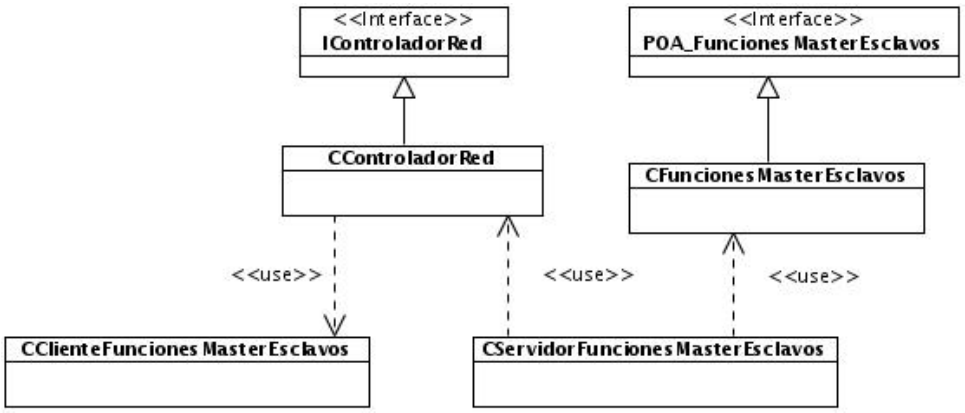

Figura 3.2: Diagrama de clases del Servicio de órdenes del nodo máster a los esclavos.

A continuación se describen los principales atributos y responsabilidades de las clases controladoras "CClienteFuncionesMasterEsclavos" y "CServidorFuncionesMasterEsclavos":

Nombre de la clase: CClienteFuncionesMasterEsclavos	
Descripción: Clase que encapsula el comportamiento del servicio Cliente máster-esclavos.	
Atributos	Tipo
ContextoRaiz	*CosNaming::NamingContext_var*
Responsabilidades de la clase	
Nombre:	*CClienteFuncionesMasterEsclavos()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CClienteFuncionesMasterEsclavos()*
Descripción:	Destructor de la clase.
Nombre:	*void IniciarServicio()*
Descripción:	Inicia el servicio cliente de órdenes del máster a los esclavos.
Nombre:	*static CClienteFuncionesMasterEsclavos* GetClienteFuncionesMasterEsclavos()*
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	*void ActualizarCreacionSubtarea()*
Descripción:	Ordena la actualización de creación de una subtarea en un nodo esclavo.
Nombre:	*void ActualizarTerminacionSubtarea()*
Descripción:	Ordena la actualización de terminación de una subtarea en un nodo esclavo.
Nombre:	*ULI32 CrearSubtareas()*
Descripción:	Ordena la creación de un conjunto de subtareas en un nodo esclavo.
Nombre:	*ULI32 CrearSubtareamaster()*
Descripción:	Ordena la creación de una subtarea máster en un nodo esclavo.
Nombre:	*void EliminarSubtarea()*
Descripción:	Ordena la eliminación de una subtarea en un nodo esclavo.
Nombre:	*void EstaNodo()*
Descripción:	Ordena el chequeo de vida de un nodo.
Nombre:	*void ConfirmarExitoCreacionSubtareas()*
Descripción:	Ordena la confirmación de creación de subtareas a un nodo.
Nombre:	*void ConfirmarErrorCreacionSubtareas()*
Descripción:	Ordena la confirmación de error de subtareas a un nodo esclavo.

Cuadro 3.4: Atributos y responsabilidades del Cliente de órdenes del máster a los esclavos.

Nombre de la clase: CServidorFuncionesMasterEsclavos	
Descripción: Clase que encapsula el comportamiento del servicio Servidor máster-esclavos.	
Atributos	Tipo
EstaHiloCreado	B8
ORB	*CORBA::ORB_var*
Sa_obj	*CORBA::Object_var*
IDHilo	*static pthread_t*
Responsabilidades de la clase	
Nombre:	*CServidorFuncionesMasterEsclavos()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CServidorFuncionesMasterEsclavos()*
Descripción:	Destructor de la clase.
Nombre:	*CORBA::ORB_var GetORB()*
Descripción:	Devuelve la instancia ORB del servicio servidor.
Nombre:	*static CServidorFuncionesMasterEsclavos* GetServidorFuncionesMasterEsclavos()*
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	*CORBA::Object_var GetSa_obj()*
Descripción:	Devuelve un objeto genérico CORBA.
Nombre:	*static void * FuncionHiloServidorFuncionesMasterEsclavos(void * pArg)*
Descripción:	Función principal para el hilo de escucha servidor.
Nombre:	*void IniciarServicio()*
Descripción:	Inicia el servicio servidor.
Nombre:	*void LiberarServicio()*
Descripción:	Libera el servicio servidor.

Cuadro 3.5: Atributos y responsabilidades del Servidor de órdenes del máster a los esclavos.

3.5. Servicio de solicitudes de los esclavos al nodo máster

Al igual que el "Servicio de órdenes del nodo máster a los esclavos", el "Servicio de solicitudes de los esclavos al nodo máster" constituye la vía de comunicación de los nodos esclavos al nodo máster. Mediante este servicio, implementado y utilizado como servidor en el motor en modo máster, los nodos esclavos realizan peticiones como la solicitud de unión al clúster y la solicitud de creación de subtareas. Constituye en sí un servicio independiente de los demás por el nivel de importancia que posee. En el siguiente diagrama de clases se representa la relación del "Servicio de solicitudes de los esclavos al nodo máster" con el "Controlador de red":

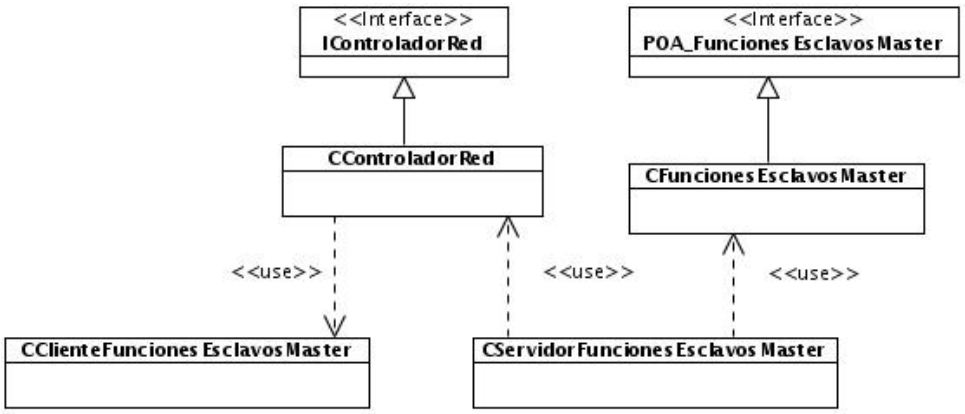

Figura 3.3: Diagrama de clases del Servicio de solicitudes de los esclavos al nodo máster.

En la *figura 3.3* puede observarse que el flujo de comunicación establecido tiene carácter unidireccional. El controlador de red utiliza al servicio cliente "CClienteFuncionesEsclavosMaster" para enviar datos al máster y a u vez este es utilizado en el máster cuando llega un dato por el servicio servidor "CServidorFuncionesEsclavosMaster". Para un mejor entendimiento de los servicios, quedan descritos a continuación los atributos y las responsabilidades de las clases controladoras "CClienteFuncionesEsclavosMaster" y "CServidorFuncionesEsclavosMaster" respectivamente:

Nombre de la clase: CClienteFuncionesEsclavosMaster	
Descripción: Clase que encapsula el comportamiento del servicio Cliente esclavos-máster.	
Atributos	Tipo
ContextoRaiz	*CosNaming::NamingContext_ var*
Responsabilidades de la clase	
Nombre:	*CClienteFuncionesEsclavosMaster()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CClienteFuncionesEsclavosMaster()*
Descripción:	Destructor de la clase.
Nombre:	*void IniciarServicio()*
Descripción:	Inicia el servicio cliente de órdenes del máster a los esclavos.
Nombre:	*static CClienteFuncionesEsclavosMaster* GetClienteFuncionesEsclavosMaster()*
Descripción:	Devuelve la instancia de la clase, patrón Singleton.
Nombre:	*void PeticionCrearSubtareas()*
Descripción:	Solicita crear una subtarea al nodo máster.
Nombre:	*void PeticionUnionCluster()*
Descripción:	Solicita unirse al clúster de alto rendimiento para recibir tareas.
Nombre:	*void PeticionDesunionCluster()*
Descripción:	Solicita separarse del clúster de alto rendimiento.

Cuadro 3.6: Atributos y responsabilidades del Cliente de solicitudes de los nodos esclavos al nodo máster.

Nombre de la clase: CServidorFuncionesEsclavosMaster	
Descripción: Clase que encapsula el comportamiento del servicio Servidor esclavos-máster.	
Atributos	Tipo
EstaHiloCreado	B8
ORB	*CORBA::ORB_var*
Sa_obj	*CORBA::Object_var*
IDHilo	*static pthread_t*
Responsabilidades de la clase	
Nombre:	*CServidorFuncionesEsclavosMaster()*
Descripción:	Constructor de la clase.
Nombre:	*virtual ~CServidorFuncionesEsclavosMaster()*
Descripción:	Destructor de la clase.
Nombre:	*CORBA::ORB_var GetORB()*
Descripción:	Devuelve la instancia ORB del servicio servidor.
Nombre:	*static CServidorFuncionesEsclavosMaster* GetServidorFuncionesEsclavosMaster()*
Descripción:	Devuelve la única instancia de la clase, patrón Singleton.
Nombre:	*CORBA::Object_var GetSa_obj()*
Descripción:	Devuelve un objeto genérico CORBA.
Nombre:	*static void * FuncionHiloServidorFuncionesEsclavosMaster(void * pArg)*
Descripción:	Función principal para el hilo de escucha servidor.
Nombre:	*void IniciarServicio()*
Descripción:	Inicia el servicio servidor.
Nombre:	*void LiberarServicio()*
Descripción:	Libera el servicio servidor.

Cuadro 3.7: Atributos y responsabilidades del Servidor de solicitudes de los nodos esclavos al nodo máster.

3.6. Conclusiones

Al comienzo de este capítulo se realizó una descripción de la arquitectura CORBA y sus ventajas a la hora de desarrollar aplicaciones distribuidas. En una segunda parte se profundizó en la metodología de desarrollo de aplicaciones CORBA haciendo especial énfasis en el lenguaje descriptivo IDL para el desarrollo de interfaces. Por otro lado se analizó la utilización del servicio de nombres "NameService" que ofrece CORBA para la centralización de la localización de objetos distribuidos y su importancia en la eficiencia de este tipo de aplicaciones. En la parte final del capítulo se describió como se desarrollaron los diferentes servicios para la comunicación a través de la red, que poseen especial importancia

en la eficiencia de las comunicaciones entre aplicaciones del MPPP. Las comunicaciones entre nodos constituye la base del intercambio de mensajes, paradigma utilizado en la plataforma Génesis para el cómputo paralelo de datos. La arquitectura CORBA para las comunicaciones disminuye el costo en tiempo, introducido en el intercambio de mensajes a través de la red frente a otras tecnologías. El desarrollo de los servicios de redes del MPPP sobre esta arquitectura permite flexibilidad, portabilidad, seguridad y eficiencia en el intercambio de datos en la plataforma.

Capítulo 4

Adición de módulos y configuración

4.1. Introducción

En el presente capítulo se describe la metodología para añadir módulos de cómputo a la plataforma el acoplamiento al Motor de Procesamiento de Peticiones paralelas (MPPP). Por otro lado se trata la manera en que se desarrollan los algoritmos paralelos para los módulos de cómputo describiendo los principales tipos de datos y funciones soportados para el cómputo paralelo y el paso de mensajes entre subtareas de cómputo. Comprobando las funcionalidades implementadas quedan ilustrados los resultados de las diferentes pruebas realizadas al MPPP y una evaluación del rendimiento demostrado. El capítulo finaliza con un epígrafe dedicado a la configuración del MPPP para optimizar el rendimiento y la funcionalidad del mismo sobre diferentes escenarios.

4.2. Módulos de cómputo

Los módulos de cómputo constituyen la realización de la plataforma y su objetivo final. Estos pueden ser añadidos de forma dinámica al MPPP sin que este deba ser compilado nuevamente, basta para ello que estén correctamente diseñados e implementados. De manera general los módulos de cómputo deben ser compilados en librerías dinámicas (Share Objects) las cuales contengan una función exportada que constituye el mecanismo de unión al motor. Los algoritmos para cómputo paralelo son programados dentro de estas librerías a partir de un conjunto de biblioteca de clases especiales para estos, también desarrolladas de manera simultánea al MPPP.

4.2.1. Interfaz de la librería dinámica

Para el desarrollo de librerías dinámicas pueden utilizarse la mayoría de los compiladores. Las librerías dinámicas son programas compilados que carecen de función principal (main), en su defecto exportan funciones que pueden ser utilizadas por programas ejecutables como si estas fueran de ellos. Esta potencialidad, unido a herramientas de la Programación Orientada a Objetos como la Herencia y el Polimorfismo, permite desarrollar programas modulares potenciando una arquitectura más flexible.

Para realizar un módulo de cómputo paralelo, la función exportada necesaria para que el motor pueda reconocer a la librería dinámica como un nuevo módulo es la siguiente:

ISubtarea GetSubtarea();*

donde el tipo "*ISubtarea*" está contenido dentro de la biblioteca de clases "libmodgenesis.a" cuyo archivo cabecera es "modgenesis.h" la cual fue desarrollada junto al motor.

Formalmente, la codificación en lenguaje C/C++ de una librería dinámica para el MPPP recomendada queda descrita como sigue:

```
//Declaración de la función a exportar.
extern "C"
{
    ISubtarea* GetSubtarea();
}
//Implementación de la función.
ISubtarea* GetSubtarea()
{
   //Algoritmo paralelo.
   return Subtarea. //Siendo Subtarea una instancia de un tipo heredado de ISubtarea.
}
```

La instancia de tipo "*ISubtarea*" debe ser de un tipo heredado de la misma. Cuando se hereda de esta clase son accesibles un conjunto de tipos de datos y funciones para el cómputo paralelo. Es de obligatoria implementación el método "*void EjecutarAlgoritmo()*" el cual contendrá el algoritmo para cómputo paralelo. Luego de compilada, la librería dinámica debe ser añadida en la sección "moduloscomputo" del fichero de configuración "genesis.conf" localizado en "/etc/genesis/genesis.conf".

4.2.2. Desarrollo de algoritmos paralelos

El Motor de Procesamiento de Peticiones Paralelas fue desarrollado para soportar el desarrollo de algoritmos para cómputo paralelo a través del paso de mensajes. Como en la mayoría de las herramientas para cómputo paralelo existentes soporta la utilización de tipos de datos primitivos, funciones para la creación de subtareas y funciones para el paso de mensajes con tipos primitivos.

4.2.2.1. Tipos de datos

A continuación se realiza una descripción de los diferentes tipos de datos soportados cuya definición está en el archivo cabecera "modgenesis.h":

Tipo	Descripción
C8	Caracter con signo de 8 bits.
UC8	Caracter sin signo de 8 bits.
I32	Entero con signo de 32 bits.
UI32	Entero sin signo de 32 bits.
LI32	Entero largo con signo de 32 bits.
ULI32	Entero largo sin signo de 32 bits.
LLI64	Entero con signo de 64 bits.
SI16	Entero corto con signo de 16 bits.
USI16	Entero corto sin signo de 16 bits.
F32	Flotante de precisión simple de 32 bits.
D64	Flotante de precisión doble de 64 bits.
LD96	Flotante de precisión doble de 96 bits.
STR	Cadena de caracteres.
B8	Booleano de 8 bits.

Cuadro 4.1: Descripción de los tipos primitivos soportados por el MPPP.

4.2.2.2. Funciones

A continuación quedan descritas las funciones heredadas de "*ISubtarea*" las cuales son utilizadas para el desarrollo del algoritmo paralelo en ""*void EjecutarAlgoritmo()*":

Función	Descripción
void EnviarC8()	Envía un dato de tipo C8 a una subtarea destino.
void EnviarUC8()	Envía un dato de tipo UC8 a una subtarea destino.
void EnviarI32()	Envía un dato de tipo I32 a una subtarea destino.
void EnviarUI32()	Envía un dato de tipo UI32 a una subtarea destino.
void EnviarLI32()	Envía un dato de tipo LI32 a una subtarea destino.
void EnviarULI32()	Envía un dato de tipo ULI32 a una subtarea destino.
void EnviarLLI64()	Envía un dato de tipo LLI64 a una subtarea destino.
void EnviarULLI64()	Envía un dato de tipo ULLI64 a una subtarea destino.
void EnviarSI16()	Envía un dato de tipo SI16 a una subtarea destino.
void EnviarUSI16()	Envía un dato de tipo USI16 a una subtarea destino.
void EnviarF32()	Envía un dato de tipo F32 a una subtarea destino.
void EnviarD64()	Envía un dato de tipo D64 a una subtarea destino.
void EnviarLD96()	Envía un dato de tipo LD96 a una subtarea destino.
void EnviarSTR()	Envía un dato de tipo STR a una subtarea destino.
void EnviarB8()	Envía un dato de tipo B8 a una subtarea destino.
C8 RecibirC8()	Recibe un dato de tipo C8 de una subtarea destino.
UC8 RecibirUC8()	Recibe un dato de tipo UC8 de una subtarea destino.
I32 RecibirI32()	Recibe un dato de tipo I32 de una subtarea destino.
UI32 RecibirUI32()	Recibe un dato de tipo UI32 de una subtarea destino.
LI32 RecibirLI32()	Recibe un dato de tipo LI32 de una subtarea destino.
ULI32 RecibirULI32()	Recibe un dato de tipo ULI32 de una subtarea destino.
LLI64 RecibirLLI64()	Recibe un dato de tipo LLI64 de una subtarea destino.
ULLI64 RecibirULLI64()	Recibe un dato de tipo ULLI64 de una subtarea destino.
SI16 RecibirSI16()	Recibe un dato de tipo SI16 de una subtarea destino.
USI16 RecibirUSI16()	Recibe un dato de tipo USI16 de una subtarea destino.
F32 RecibirF32()	Recibe un dato de tipo F32 de una subtarea destino.
D64 RecibirD64()	Recibe un dato de tipo D64 de una subtarea destino.
LD96 RecibirLD96()	Recibe un dato de tipo LD96 de una subtarea destino.
STR RecibirSTR()	Recibe un dato de tipo STR de una subtarea destino.
B8 RecibirB8()	Recibe un dato de tipo B8 de una subtarea destino.
ULI32* CrearSubtareas()	Solicita crear subtareas de cómputo de un tipo determinado.
ULI32 GetIDSTLocal()	Devuelve el identificador de la subtarea local.
ULI32 GetIDSTPadre()	Devuelve el identificador de la subtarea padre.
ULI32 GetIDSolicitud()	Devuelve el identificador de la solicitud que generó la subtarea.

Cuadro 4.2: Descripción de las funciones utilizadas para el desarrollo de algoritmos paralelos.

4.2.2.3. Ejemplo

En el siguiente ejemplo simple, se demuestra como se desarrolla un algoritmo paralelo a través de la plataforma:

En el algoritmo máster:

```
void CSubtareaPruebaMaster::EjecutarAlgoritmo()
{
   ULI32 Suma = 0;
   ULI32 *IDs = CrearSubtareas("SUMAEJEMPLO",100);
   for(USI16 vIl = 0; vIl<100;vIl++)
  {
      EnviarULI32(IDs[vIl],0,111);
  }
   for(USI16 vIl = 0; vIl<100;vIl++)
  {
      Suma += RecibirULI32(IDs[vIl],1);
  }
  cout<<"Resultado: "<<Suma<<endl;
}
```

En el algoritmo esclavo:

```
void CSubtareaPruebaEsclavo::EjecutarAlgoritmo()
{
   ULI32 Dato = RecibirULI32(GetIDSTPadre(),0);
   EnviarULI32(GetIDSTPadre(),1,Dato);
}
```

Mediante la función "CrearSubtareas" son solicitadas al nodo máster la creación de 100 subtareas de cómputo de tipo "SUMAEJEMPLO". Cuando el nodo máster recibe esta información planifica mediante balance de carga estas 100 subtareas de tipo esclavo en diferentes nodos del clúster. Cada vez que una subtarea es creada en un nodo , su localización es difundida para los demás nodos del clúster. Una vez terminado este proceso los 100 identificadores de las mismas son devueltos a la subtarea máster y almacenados el el arreglo "IDs". Para cada uno de las 100 subtareas es enviado mediante la función "EnviarULI32" el dato "111" con etiqueta "0". En este momento los esclavos están escuchando este dato del la subtarea padre mediante la función "RecibirULI32" con la misma etiqueta. Los esclavos reciben el

dato y lo reenvían al padre, el cual esta esperando por todos los resultados enviados para sumarlos. El resultado de la suma es "11100". Este algoritmo sencillo solo posee valor para las pruebas realizadas para determinar y corregir errores de funcionamiento, no está diseñado para resolver problemas paralelos. Con este y otros algoritmo fueron probados todos los tipos de datos primitivos y corregidos todos los errores de funcionamiento.

A continuación se describe un algoritmo paralelo para la multiplicación de matrices en paralelo almacenadas en bases de datos relacionales, cuyo resultado demuestra la viabilidad del cómputo paralelo frente a soluciones secuenciales:

En el algoritmo máster:

```
void CSubtareaPruebaMaster::EjecutarAlgoritmo()
{
        CMT_solicitud_matriz * vSoll = new CMT_solicitud_matriz();
        vSoll->Conectar(CLectorParametrosConfiguracion::GetLector()->GetBaseDatos("pgsql"));
        SolicitudMatriz vSolicitudl = vSoll->ObtenerSolicitudMatrices(GetIDSolicitud());
        ULI32 *IDs = CrearSubtareas("MULTMAT",vSolicitudl.CantidadFilasME1);
        for(USI16 vIl = 0; vIl<vSolicitudl.CantidadFilasME1;vIl++)
        {
                EnviarULI32(IDs[vIl],0,vIl);//Numero de fila que le toca procesar
                EnviarULI32(IDs[vIl],1,vSolicitudl.IDMatrizEntrada1);//ID de matriz de entrada 1
                EnviarULI32(IDs[vIl],2,vSolicitudl.IDMatrizEntrada2);//ID de matriz de entrada 2
                EnviarULI32(IDs[vIl],3,vSolicitudl.IDMatrizResultado);//ID de la matriz resultado
                EnviarULI32(IDs[vIl],4,vSolicitudl.CantidadColumnasME1);//Cantidad de columnas matriz de entrada 1
                EnviarULI32(IDs[vIl],5,vSolicitudl.CantidadColumnasME2);//Cantidad de columnas matriz de entrada 2
        }
        //cout<<"Envie todos los datos a mis esclavos"<<endl;
        for(USI16 vIl = 0; vIl<vSolicitudl.CantidadFilasME1;vIl++)
        {
                if(!RecibirB8(IDs[vIl],6))cout<<"SUBTAREA: "<<IDs[vIl]<<" no ha terminado correctamente"<<endl;
                else cout<<"SUBTAREA: "<<IDs[vIl]<<" ha terminado correctamente"<<endl;
        }
        cout<<"SUBTAREA MASTER: "<<GetIDSTLocal()<<" ha terminado correctamente"<<endl;
}
```

En el algoritmo esclavo:

```
void CSubtareaPruebaEsclavo::EjecutarAlgoritmo()
{

        CMT_valores_matriz * vValoresMatrizl = new CMT_valores_matriz();
        vValoresMatrizl->Conectar(CLectorParametrosConfiguracion::GetLector()->GetBaseDatos("pgsql"));
        ULI32 vNumFilal = RecibirULI32(GetIDSTPadre(),0);//Numero de fila que le toca procesar
        ULI32 vIDMatrizEntrada1l = RecibirULI32(GetIDSTPadre(),1);//ID de matriz de entrada 1
        ULI32 vIDMatrizEntrada2l= RecibirULI32(GetIDSTPadre(),2);//ID de matriz de entrada 2
        ULI32 vIDMatrizResultadol= RecibirULI32(GetIDSTPadre(),3);//ID de la matriz resultado
        ULI32 vCantidadColumnasME1l = RecibirULI32(GetIDSTPadre(),4);//Cantidad de columnas matriz de entrada 1
        ULI32 vCantidadColumnasME2l = RecibirULI32(GetIDSTPadre(),5);//Cantidad de columnas matriz de entrada 2
        ULI32 * vVectorFilal = new ULI32[vCantidadColumnasME1l];
        for(ULI32 vIl = 0;vIl<vCantidadColumnasME1l;vIl++)
        {
                vVectorFilal[vIl]= vValoresMatrizl->ObtenerValor(vNumFilal,vIl,vIDMatrizEntrada1l);
        }
        for(ULI32 vCantColum2l = 0;vCantColum2l < vCantidadColumnasME2l;vCantColum2l++)
        {
                D64 vElementol = 0.0;
                for(ULI32 vCantColum1l = 0;vCantColum1l<vCantidadColumnasME1l;vCantColum1l++)
                {
                    vElementol +=
                    vVectorFilal[vCantColum1l]*
                    vValoresMatrizl->ObtenerValor(vCantColum1l,vCantColum2l,vIDMatrizEntrada2l);
                }
                vValoresMatrizl->InsertarValor(vNumFilal,vCantColum2l,vIDMatrizResultadol,vElementol);
        }
        EnviarB8(GetIDSTPadre(),6,true);//Reportar al máster
}
```

Resultados:

Este algoritmo fue aplicado a una base de datos PostgreSQL obteniéndose los valores correctos para diferentes matrices multiplicando. Al intercambiar mensajes los datos no se corrompieron y llegaron a los destinos solicitados. En cuanto al rendimiento para un algoritmo secuencial sobre un gestor de bases de datos con 800000 consultas, el algoritmo demoró 1 hora 30 minutos en realizar el cómputo de las matrices. Para un solo gestor con el algoritmo paralelo en la plataforma el tiempo fue reducido a 1 hora mientras que al utilizar el gestor distribuido el tiempo fue reducido a 30 minutos solo con 6 nodos.

Estos resultados reflejan claramente la potencialidad que puede ser aprovechada al desarrollar algoritmos paralelos sobre bases de datos distribuidas para el procesamiento paralelo de datos.

4.3. Fichero de configuración

El fichero de configuración “genesis.conf” se localiza en la dirección “/etc/genesis/genesis.conf” y contiene un conjunto de variables de configuración para el Motor de Procesamiento de Peticiones

Paralelas. Estas variables permiten configurar el motor para diferentes escenarios. Añadir módulos de cómputo, drivers para soporte a bases de datos, configuración de parámetros de red, entre otras funciones, implican realizar cambios en el fichero de configuración. El fichero de configuración está dividido por secciones las cuales agrupan un conjunto de variables cada una. Muchas de estas variables pueden ser omitidas, otras no, en caso de una omisión indebida en MPPP emitirá un aviso para que el problema sea corregido.

4.3.1. Sección "local"

La sección "local" agrupa variables del nodo local, las cuales son comunes para cualquier nodo, ya sea el nodo máster o los nodos esclavos. A continuación queda descrita esta sección con las variables agrupadas por esta:

```
local
{
    modo = esclavo
    nombrepclocal = eipad3.uci.cu
    tamanobufer = 128
    cantmaxsubtareas = 300
    plataforma = linux
    autoprocesar = si
    raizgenesis = /Proyecto/genesis
}
```

En la siguiente tabla se describen cada una de las variables de la sección:

Variable	Descripción
modo	Establece el modo en que se ejecutará el motor (máster o esclavo)
nombrepclocal	Nombre del nodo en que está localizado el fichero.
tamanobufer	Tamaño del buffer de mensajes de las subtareas de cómputo.
cantmaxsubtareas	Cantidad máxima de subtareas que asumirá el nodo.
plataforma	Plataforma del nodo (Linux, Windows etc.).
autoprocesar	Determina si al ejecutarse, el motor asumirá tareas automáticamente (si o no).
raizgenesis	Carpeta raíz de los binarios del MPPP.

Cuadro 4.3: Descripción de las variables de configuración de la sección "local".

La variable "tamanobufer" indica la cantidad de mensajes que una subtarea puede almacenar en su buffer antes que los mismos comiencen a ser paginados por exceso. Por otra parte la variable "cantmaxsubtareas" indica la cantidad máxima de subtareas que puede asumir el nodo.

4.3.2. Sección "nodomaster"

La sección "nodomaster" está reservada para parámetros de configuración del nodo máster. A continuación queda descrita la sección:

```
nodomaster
{
    nombrenodomaster = eipad5.uci.cu
    permitirsubtareaslocales = no
}
```

En la siguiente tabla se describen cada una de las variables de la sección:

Variable	Descripción
nombrenodomaster	Nombre del nodo que cumplirá funciones de máster del clúster.
permitirsubtareaslocales	Indica si deben ser planificadas subtareas en el máster o no (si o no).

Cuadro 4.4: Descripción de las variables de configuración de la sección "nodomaster".

4.3.3. Sección "basedatos"

En la sección "basedatos" se configuran los parámetros de las bases de datos con que la plataforma interactúa. Elementos de configuración como soporte para nuevos drivers de bases de datos o accesos a nuevas bases de datos implican la modificación de esta sección. A continuación queda descrita esta sección con las variables y subsecciones agrupadas por esta:

```
basedatos
{
    idbd genesis
    {
        nombreservidorbd= eipad5.uci.cu
        puerto = 5432
        nombrebd= Genesis
```

```
        usuariobd= postgres
        password= postgres
        driver= libpgsql.so
    }
}
```

Cada vez que se desee añadir un nuevo acceso a bases de datos, debe incluirse una subsección "idbd". En este caso el alias de esta subsección es "genesis" y se incluyen dentro de esta los parámetros de configuración para el acceso los cuales quedan descritos en la siguiente tabla:

Parámetro	Descripción
nombreservidorbd	Nombre del servidor de bases de datos.
puerto	Puerto del servidor de bases de datos.
nombrebd	Nombre de la base de datos a la que se desea acceder.
usuariobd	Usuario para acceder a esta base de datos.
password	Contraseña para acceder a esta base de datos.
driver	Driver Génesis de soporte a este tipo de bases de datos.

Cuadro 4.5: Descripción de las variables de configuración de la sección "nodomaster".

4.3.4. Sección "red"

La sección "red" está dedicada a la configuración de los parámetros globales de la red. La arquitectura CORBA ofrece servicios como el servicio de nombres (NameService) que necesitan ser conocidos por los nodos, ya sea el máster o los esclavos. Un ejemplo de la configuración de esta sección queda descrito como sigue:

```
red
{
    nodoservidornombres = eipad3.uci.cu
}
```

La variable "nodoservidornombres" tiene como valor el nodo que contiene el servicio de nombres. Este servicio es utilizado por los nodos del clúster para localizar los objetos distribuidos u con esta información utilizar los diferentes servicios de red.

4.3.5. Sección "moduloscomputo"

Las sección "moduloscomputo" agrupa las variables de configuración de los módulos de cómputo añadidos a la plataforma. Para que el Motor de Procesamiento de Peticiones Paralelas reconozca los módulos, deben estar descritos en el fichero de configuración de la siguiente forma:

```
moduloscomputo
{
    modulo matrices {alias= multmatrices lib= multmatrices.so}
    modulo biotecnologia{alias= biotecnologicos lib= biotec.so}
    modulo prediccioneconomica{alias= economicos lib= eco.so}
    modulo tamanoparticula{alias= particulas lib= part.so}
}
```

donde la palabra que sigue a "modulo" constituye el nombre del módulo, el "alias" del módulo es utilizado para indicar el tipo de subtarea en los algoritmos paralelos y la variable "lib" indica la librería dinámica que posee la función "*ISubtarea* GetSubtarea()*" publicada.

4.4. Conclusiones

En este capítulo se analizó la metodología a seguir en el desarrollo de módulos de cómputo al Motor de Procesamiento de Peticiones Paralelas. En una segunda parte se describieron los tipos de datos primitivos y funciones que son utilizados en la conformación de un algoritmo paralelo y los principios básicos para la realización de estos. Al finalizar este capítulo se describieron las diferentes secciones del fichero de configuración "genesis.conf" las cuales permiten adecuar el motor a las características de diferentes escenarios para maximizar el rendimiento del mismo. Las resultados arrojados ante las pruebas demuestran la eficiencia que se obtiene al utilizar algoritmos paralelos frente a soluciones secuenciales en situaciones que así lo permiten. Para cada escenario, debe estudiarse y particularizarse si es viable o no la utilización de algoritmos paralelos y evaluarse la correcta configuración del Motor de Procesamiento de Peticiones Paralelas para entorno dónde sea utilizado.

Conclusiones

Al concluir el presente trabajo queda realizado un estudio sobre los sistemas para cómputo paralelo de datos a partir del cual se decidió realizar una implementación íntegra del Motor de Procesamiento de Peticiones Paralelas (MPPP) para la plataforma Génesis ya que las herramientas estudiadas no permitían la utilización de ninguna de estas de acuerdo a las necesidades planteadas. El proceso de modelación, diseño e implementación de la aplicación se desarrolló por la herramientas escogidas. Se desarrolló una primera versión funcional del MPPP el cual fue probado bajo diferentes escenarios arrojando pruebas satisfactorias. Durante el proceso de pruebas fueron detectados errores los cuales fueron corregidos y probados oportunamente. Como elemento añadido a los objetivos planteados, se profundizó en los elementos de configuración del MPPP para su utilización bajo entornos diferentes para maximizar su rendimiento. Para cada caso debe estudiarse la complejidad del problema tratado y los recursos disponibles para un mejor rendimiento. Por lo anteriormente expuesto se dan por cumplidos los objetivos propuestos al iniciar el trabajo, existiendo una correspondencia entre los objetivos planteados y los resultados obtenidos.

Recomendaciones

Luego de cumplirse los objetivos del trabajo, se recomienda:

Comenzar otro ciclo de desarrollo con el mismo para dar soporte a nuevas funcionalidades no incluidas en esta primera versión como la adición de primitivas de reducción, barreras de sincronización, soporte para grupos y mensajes por difusión, las cuales están presente en la mayoría de los sistemas para cómputo paralelo. Además de estas funcionalidades se recomienda incluir el soporte para el intercambio de mensajes complejos como vectores, matrices, listas y tipos de datos distribuidos. Desde otra perspectiva se recomienda potenciar el soporte para nuevos tipos de componentes distribuidos para el acceso, recuperación y tratamiento de datos a través del intercambio de mensajes mediante algoritmos paralelos.

Bibliografía

[1] Bull home page, 2009. [Disponible en: http://www.bull.com]

[2] Colaboradores Joanne Woodcock, Diccionario de Informática e Internet de MIcrosoft.[ET al.]._MAdrid: Mc Graw-Hill Interamericana de España, 2001. pp462.

[3] Colaboradores Joanne Woodcock, Diccionario de Informática e Internet de MIcrosoft.[ET al.]._MAdrid: Mc Graw-Hill Interamericana de España, 2001. pp140

[4] Debian home page, 2009. [Disponible en: http://www.debian.org/index.es.html]

[5] Duncan Grisby. Omniidl — The omniORB IDL Compiler. AT&T Laboratories Cambridge, 2000, 21pp

[6] EasyEclipse home page, 2009. [Disponible en: http://www.easyeclipse.org/site/home/]

[7] Eoin Carroll. The omniORB utilities. 2000, 2pp

[8] Fast Fourier Transform, 2009. [Disponible en: http://mathworld.wolfram.com/FastFourierTransform.html]

[9] Faucets: Shared Computing Power, 2009. [Disponible en: http://charm.cs.uiuc.edu/research/faucets]

[10] Flynn M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

[11] Free High Performance ORB, 2009. [Disponible en: http://omniorb.sourceforge.net]

[12] GNU home page, 2009. [Disponible en http://www.gnu.org]

[13] Horstmann Markus, Kirtland Mary. DCOM Architecture, 2009. [Disponible en: http://msdn.microsoft.com/en-us/library/ms809311.aspx]

[14] Javier Holguera Blanco, Daniel García García. CORBA. 2004, 24pp

[15] Jones, M. Tim. Linux and symmetric multiprocessing, 2007. [Disponible en: http://www.ibm.com/developerworks/library/l-linux-smp]

[16] MPI home page, 2009. [Disponible en: http://www-unix.mcs.anl.gov/mpi]

[17] Olson Mike, Ogbuji Uche. Messaging technologies compared, 2009. [Disponible en http://www.ibm.com/developerworks/webservices/library/ws-pyth9/#table1]

[18] Osiris So_a Albert, Isabel Casas Sandra. Survey de Tecnologías Grid, 2009. [Disponible en: http://www.cyta.com.ar/ta0704/v7n4a1.htm]

[19] OpenMosix home page, 2009. [Disponible en http://openmosix.sourceforge.net]

[20] PgAdmin home page, 2009. [Disponible en: http://www.pgadmin.org/]

[21] PgFoundry home page, 2009. [Disponible en: http://pgfoundry.org/projects/pgcluster/]

[22] Pthreads home page, 2009. [Disponible en: http://www.gnu.org/software/pth]

[23] PVM home page, 2009. [Disponible en: http://www.epm.ornl.gov/pvm]

[24] Remote Method Invocation Home Page, 2009. [Disponible en: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp]

[25] Rodríguez, Eduardo René. Cronología del procesamiento en paralelo,2009. [Disponible en: http://homepage.mac.com/eravila/histpara.html]

[26] Smerlin L., Tschanz R. Procesamiento Paralelo: qué tener en cuenta para aprovecharlo. Conceptos y alternativas en Linux. Universidad Tecnológica Nacional. Santa Fe Argentina, 2007, 29pp

[27] Subversion home page, 2009. [Disponible en: http://subversion.tigris.org/]

[28] Duncan Grisby. The omniORB version 4.1 User's Guide. AT&T Laboratories Cambridge, 2007, 131pp

[29] The OMNI Naming Service. AT&T Laboratories Cambridge, 2008, 3pp

[30] Timeline of Computing History, Computer, vol. 29, no. 10, pp. TL1-TL34, Oct. 1996

[31] Tristan Richardson. The OMNI Thread Abstraction. AT&T Laboratories Cambridge, 2001, 7pp

[32] Visual Paradigm home page, 2009. [Disponible en: `http://www.visual-paradigm.com/`]

[33] XML-RPC home page, 2009. [Disponible en `http://www.xmlrpc.com`]

[34] 7030 Data Processing System, 2009. [Disponible en: `http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP7030.html`]

[35] 704 Data Processing System, 2009. [Disponible en: `http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html`]

Anexo I Estándares de código C++

Nomenclatura de las clases

1. El nombre de la clase debe ser un nombre claro, fonéticamente agradable.
2. Debe cumplir con las reglas de C++ para la nomenclatura.
3. En cuanto a estructura debe comenzar inicialmente con la letra "C" en mayúscula.
4. La letra "C" debe estar seguida una palabra con letra inicial mayúscula.
5. Si el nombre es compuesto se aplica lo anterior y se le concatena la otra palabra comenzando esta con letra inicial mayúscula.
6. No se debe utilizar el carácter "_" para los nombres de clases.
7. En el caso de las interfaces el nombre comenzará con "I" seguido de otra palabra con letra inicial mayúscula.
8. Nuca salvo excepciones bien justificadas el nombre de la clase aparecerá en plural.
9. Las llaves de la clase estarán alineadas a 0 espacios horizontales coincidiendo a verticalmente con la palabra reservada ***class***. La segunda llave siempre irá seguida del caracter ";" que nunca debe faltar.

Nomenclatura de los métodos

1. Siempre se le declarará un método "Set" y un "Get" para cada atributo aunque se suponga a priori que no será utilizado.
2. El orden será primero todos los "Set" , luego todos los "Get" y luego las restantes funciones de la clase.
3. En caso de que existan funciones privadas estas se declararán debajo del último atributo declarado.
4. Debe ser fonéticamente agradable y expresar lo que hace de tan solo leerlo.
5. El nombre de un método siempre comenzará con una palabra con letra inicial mayúscula.

6. En el caso de nombres compuestos se le concatenará la siguiente o siguientes palabras comenzando cada una con letra inicial mayúscula.

7. Si la función es para cambiarle el valor a un atributo se le pondrá delante el prefijo “Set” seguido del nombre del atributo. Entre paréntesis se le colocará el tipo de dato del atributo y el mismo nombre del atributo con el prefijo "p". Esta función será de tipo void.

8. Si la función es para retornar el valor a un atributo se le pondrá delante el prefijo “Get” seguido del nombre del atributo. Entre paréntesis no se le pasará ningún parámetro. Esta función será del tipo del atributo.

9. No se utilizará el carácter “_” para la nomenclatura de funciones, este queda reservado para otras nomenclaturas.

10. El nombre de los parámetros de los métodos debe cumplir con las reglas de formación de los atributos a lo cual se le añade que comenzarán con la letra inicial minúscula "p".

11. Los métodos se declararán con un espacio Tab a partir de 0 espacios con respecto a su modificador de visibilidad superior.

Nomenclatura de los atributos

1. Deben comenzar con una palabra con letra inicial mayúscula.

2. Si el nombre es compuesto se aplica lo anterior y se le concatena la otra palabra comenzando esta con letra inicial mayúscula.

3. Si este atributo es un apuntador debe, concatenársele al final del nombre un "Ptr".

4. No se debe utilizar el carácter "_" para los nombres de los atributos, este queda reservado para otras nomenclaturas.

Métodos a implementar

1. Los métodos a implementar deben ser colocados en el .cpp salvo excepciones por ejemplo la declaración de plantillas donde los métodos serán declarados e implementados en el .h.

2. La implementación debe seguirse de acuerdo a las normas establecidas por el lenguaje.

Declaración y nomenclatura de constantes

1. Nunca se omitirá el tipo de dato en la declaración e inicialización y se declararán de acuerdo a las definiciones del lenguaje para constantes.
2. El nombre comenzará siempre con la letra "c" seguido de una palabra con letra inicial mayúscula.
3. Si el ámbito de la constante es global se le concatenará al final del nombre una "g" y si es local una "l".
4. No se utilizará el carácter "_" para la nomenclatura de las constantes.
5. En el caso que el nombre sea compuesto se concatenarán la o las palabras comenzando cada una con letra inicial mayúscula.

Declaración y nomenclatura de variables

1. Se declararán de acuerdo a las definiciones del lenguaje para variables.
2. El nombre comenzará siempre con la letra "v" seguido de una palabra con letra inicial mayúscula.
3. Si el ámbito de la variable es global se le concatenará al final del nombre una "g" y si es local una "l".
4. No se utilizará el carácter "_" para la nomenclatura de las variables, este queda reservado para otras nomenclaturas.
5. En el caso de que el nombre sea compuesto se concatenarán la o las palabras comenzando cada una con letra inicial mayúscula.
6. Siempre se inicializarán con un valor.
7. Los apuntadores se inicializarán por defecto con un "0" o pidiéndole memoria.

Creación y nomenclatura de instancias de clases

1. Las instancias de las clases se acogerán a las reglas de formación de los atributos y variables o constantes según sea el caso.

Condicionales

1. Las condicionales deben anidarse de acuerdo a las Normas de tabulación del código fuente.

2. Si se realiza una comparación con una constante debe colocarse primero la constante y luego la variable pues de no ser así y se confunde el signo "==" por "=" solo emite una advertencia y de la forma descrita aquí el compilador emite un error.

Ciclos

1. Con el objetivo de que nuestro código sea más portable las variables de control de los ciclos serán declaradas e inicializadas antes de declarar los ciclos, hay compiladores que no admiten la declaración en el mismo ciclo.

2. Las normas de tabulación se subordinan a las normas de Tabulación del código fuente.

Tabulación del código fuente

1. Para la tabulación del código fuente se colocarán las instrucciones al mismo nivel.

2. Si se anidan las condicionales o ciclos combinados con condicionales, se colocarán las instrucciones a un espacio Tab de su instrucción exterior.

3. Las llaves de ciclos, condicionales etc. se colocarán debajo de los mismos siendo buena práctica escribir ambas llaves en el momento que se debe colocar la primera.

4. Si la condicional o ciclo solo posee una sola instrucción podrán ser eliminadas las llaves opcionalmente, en cuyo caso es obligatorio colocar la instrucción al lado del ciclo o la condicional.

5. Opcionalmente se podrán declarar o no en la misma línea variables del mismo tipo de datos.

Glosario de Términos

CORBA Common Object Request Broker Architecture

DCOM Distributed Component Object Model

ERP Enterprise Resource Planning

IP Internet Protocol

MPI Message Passing Interface

MPPP Motor de Procesamiento de Peticiones Paralelas

PVM Parallel Virtual Machine

RMI Java Remote Method Invocation

SOAP Simple Object Access Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol