
CINVESTAV-IPN
Biblioteca de Ingeniería Eléctrica

FB0000011712

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

SECCIÓN DE COMPUTACIÓN

CINVESTAV
IPN
ADQUISICION
DE LIBROS

C++ Distribuido

Tesis que presenta el:

Ing. José Mitre Silva

Para obtener el grado de:

Maestro en Ciencias en la Especialidad de Ingeniería Eléctrica

Director de Tesis: Dr. Héctor Ruiz Barradas

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

México, D.F., Julio 1998

XM
CLASIF.: 98.14
ADQUIS.: B1-15361
FECHA: 21-X-1995
PROCED.: Tesis-1998
$

A mi Padres y a mi Esposa con todo mi amor

Agradecimientos

A mis asesores: Dr. Héctor Ruíz Barradas y al M. en C. José Oscar Olmedo por su paciencia, sus valiosas aportaciones a través de sus comentarios y sugerencias, a quienes debo mi interés por este trabajo.

Al Dr. Feliú Davino Sagols Troncoso y al M. En C. Uriel Tirado Ríos por sus comentarios y por aceptar ser el jurado para la evaluación de este trabajo.

A todos mis profesores por sus experiencias, conocimientos y enseñanzas.

A mis padres por su comprensión, cariño y ejemplo; y por impulsarme y motivarme con sus consejos.

A Lulú por todo su amor y apoyo, ya que sin el no hubiera podido lograrlo.

A mis compañeros Gonzalo, Heberto, Francisco, César, Carlos, Carlos Alberto, Jesús, Nury, Eduardo, Manuel, Enrique, Samuel y Ricardo por su amistad y compañía.

A Sofía, Anabel y Flor por su ayuda y paciencia.

Al CONACyT por el apoyo económico proporcionado para la realización de este trabajo.

Y a todos los que de alguna manera me apoyaron durante todo este tiempo.

Muchas gracias.

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

Haz que cada meta lograda sea para ti sólo un nuevo punto de partida.

Alfredo Cuellar G.

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

CONTENIDO

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

1. Introducción

Los sistemas computacionales actuales buscan, entre otras cualidades, mayor rapidez en el procesamiento de datos, transparencia en el acceso a los recursos, seguridad en los datos, interfaces más gráficas, compartición de datos y dispositivos. Dejar el trabajo a un solo proceso o procesador no es suficiente para las exigencias actuales; la alternativa está en dividir el trabajo entre varios procesos.

Actualmente es común descomponer una aplicación compleja, en una serie de tareas más sencillas que se ejecuten en paralelo. Por otro lado, la multiplicación de redes y de estaciones de trabajo ha permitido que dichas tareas se ejecuten en distintas computadoras. Esto ha dado lugar a sistemas complejos conocidos como Sistemas Distribuidos. Los sistemas distribuidos son, por naturaleza, complejas aplicaciones que requieren de la interacción de varios componentes, en ocasiones físicamente dispersos.

Las claras ventajas que reporta el uso de redes de computadoras, en particular de sistemas distribuidos, ha impuesto a los programadores el reto de crear sistemas para ser utilizados en este entorno. Sin embargo, es necesario contar con lenguajes de programación que permitan al diseñador de sistemas programar de una manera sencilla, pero efectiva.

En la actualidad, existen muy pocos lenguajes diseñados expresamente para la programación de sistemas distribuidos, entre los más conocidos se encuentran OCAM y JAVA. Sin embargo, estos lenguajes son poco conocidos por la mayoría de los programadores comunes. Para resolver este problema, a lenguajes imperativos más conocidos, como PASCAL o C, se les han agregado herramientas que facilitan la programación de sistemas distribuidos, inclusive se han creado ambientes de programación paralela y/o distribuida utilizando estos lenguajes, como es el caso de PVM (Parallel Virtual Machine) [10].

Estas herramientas ayudan al programador a realizar aplicaciones distribuidas, sin embargo no son sencillas de entender ni de utilizar; no permiten al programador concentrarse en el desarrollo de la aplicación distribuida, ya que los problemas a los que se enfrenta éste son, primero, entender los mecanismos que utilizará para la creación de su programa y luego dedicarse a la programación de su sistema en sí, lo cual complica la programación.

Ante esta problemática, se pensó en crear herramientas adicionales para algún lenguaje X, que fueran fácilmente entendibles, que permitieran la disminución de código en las aplicaciones y que facilitaran la programación de sistemas distribuidos.

Para realizar lo anterior, el primer paso fue seleccionar el lenguaje al que se le iban a hacer las extensiones. Este lenguaje debería ser conocido por la mayoría de los programadores y utilizado en muchas de las aplicaciones actuales, con el fin de lograr un amplio entorno de uso. Asimismo, este lenguaje debería permitir mayor modularidad en las aplicaciones y

mayor abstracción en las funciones, características que en la actualidad son indispensables en los sistemas. Pensamos que el lenguaje C cumple con estas características, ya que actualmente es uno de los lenguajes de programación mas difundidos y utilizados en el ámbito mundial. Sin embargo, debido a la necesidad de modularidad, abstracción, disminución de código y la creciente popularidad de la programación orientada a objetos durante los últimos años, el lenguaje seleccionado fue C++. Este lenguaje soporta la abstracción y el encapsulamiento de datos de manera adecuada, por lo que resulta atractivo para el desarrollo de sistemas de software grandes y complejos. Además, los programadores expertos en C pueden dominar C++ rápidamente por ser éste último una extensión del primero.

Una vez seleccionado el lenguaje, el siguiente paso fue crear las herramientas diseñadas para facilitar la programación de sistemas distribuidos. Estas herramientas consisten en funciones, nuevas sentencias y construcciones que permiten al programador la creación, la comunicación y el control de procesos en el acceso a variables compartidas, de una manera sencilla. Se pretende que estas extensiones sean fáciles de entender y que tengan la abstracción necesaria para que el programador pueda concentrarse en el sistema y no en la implantación y/o utilización de estos elementos.

El resultado de este trabajo fue el lenguaje de programación que hemos llamado ***C++ Distribuido***. Su diseño e implementación son reportados en los siguientes capítulos, cuya estructura se explica a continuación.

El capítulo dos comienza introduciendo la terminología básica de la programación orientada a objetos, de la programación distribuida orientada a objetos, pasando por una breve presentación de algunos sistemas distribuidos que ayudan a entender el desarrollo de la tesis. El capítulo concluye con una propuesta que pretende resolver algunos problemas detectados en los lenguajes actuales de programación distribuidos orientados a objetos.

En el capítulo tres se describen las características del lenguaje ***C++ Distribuido***; las extensiones realizadas al C++ y la forma en que estas funcionan.

El capítulo cuatro esta dedicado a describir el diseño e implantación de la clase ***process***, cuyo objetivo es crear objetos en C++, los cuales se comportan como procesos. Este capítulo describe con detalle como se realizó el diseño e implantación de esta clase, así como los problemas encontrados durante su implantación.

El capítulo cinco explica cómo se implantó y cómo se usa la instrucción ***select-when*** para permitir la concurrencia de procesos y la exclusión mutua.

En el capítulo seis se muestra como se implantó la comunicación entre procesos, las herramientas utilizadas y los problemas encontrados.

El capítulo siete está dedicado a la presentación de una aplicación escrita usando el ***C++ Distribuido*** que muestra el uso de las extensiones descritas en este trabajo.

Por último presentamos las conclusiones y algunas sugerencias para trabajos futuros en esta dirección.

2. Lenguajes y Sistemas de Programación Distribuidos Orientados a Objetos

Durante los últimos años los sistemas computacionales han evolucionado enormemente. Día con día aumenta la necesidad de tener sistemas más confiables y rápidos, pero también aumenta la necesidad de que estos sistemas se ejecuten sobre redes de computadoras, de tal forma que la información pueda ser procesada en forma distribuida.

Para el desarrollo de estos sistemas es necesario contar con lenguajes que ofrezcan medios para la abstracción de datos, para el encapsulamiento, para la modularidad, para el reuso de código, para la comunicación entre procesos y para el paso de mensajes, entre otras características; ya que esto facilita la programación.

La programación orientada a objetos ofrece muchos de estos beneficios, pero carece de soporte para la programación distribuida, que también es altamente demandada en la mayoría de los sistemas actuales. Es por eso que surgió lo que se conoce como *Sistemas de Programación Distribuida Orientada a Objetos*, que unifica los conceptos de programación de sistemas distribuidos y la programación orientadas a objetos.

A continuación se presentarán algunos de los conceptos básicos de la programación distribuida y la programación orientada a objetos, y se dará una breve lista con algunos de los sistemas distribuidos orientados a objetos existentes actualmente. Más adelante se presentarán algunos problemas detectados en los sistemas y lenguajes de programación distribuida orientada a objetos, terminaremos con una propuesta para la solución de algunos estos problemas.

2.1. Programación Orientada a Objetos

La programación orientada a objetos se ha vuelto muy popular en los últimos años. Los programadores de compiladores y otros fabricantes de software se están apresurando a poner en venta versiones de sus productos orientadas a objetos. Sobre el tema han aparecido innumerables libros y ediciones especiales de revistas tanto académicas como comerciales. A juzgar por tanta actividad, la programación orientada a objetos está recibiendo la atención que en un tiempo se destino a ideas como la programación estructurada o los sistemas expertos.

Es probable que la programación orientada a objetos provenga en parte de la esperanza de que esta nueva técnica, como paso con muchas innovaciones previas en el desarrollo de software, sea la clave para incrementar la productividad, mejorar la confiablilidad y reducir los tiempos de programación. Aunque es verdad que se obtienen muchos beneficios al usar técnicas de la programación orientada a objetos, también es cierto que programar es todavía una tarea muy difícil; llegar a ser experto en programación requiere talento.

La programación orientada a objetos es más que una simple colección de lenguajes de programación nuevos. La programación orientada a objetos es una nueva forma de pensar acerca de lo que significa programar, acerca de cómo podemos estructurar la información dentro de una computadora.

Con frecuencia se alude a la programación orientada a objetos como un nuevo paradigma de programación. Otros paradigmas mencionados a veces incluyen el de la programación imperativa (lenguajes como Pascal o C), el de la programación lógica (Prolog) y el de la programación funcional (FP o ML)[28].

La programación orientada a objetos no consiste simplemente de unas cuantas características añadidas a los lenguajes de programación. Mas bien es una nueva forma de pensar acerca del proceso de descomposición de problemas y de desarrollo de soluciones de programación.

Los lenguajes de programación orientada a objetos permiten crear y diseñar programas que consisten de un conjunto de componentes autónomos, llamados *objetos*. Los objetos son entidades que agrupan información o datos y un conjunto de operaciones asociadas o procedimientos que manipulan sus datos, son responsables de tareas específicas e interaccionan entre sí bajo un esquema de colaboración. Los objetos son *ejemplares* o *instancias* de una *clase*. Todas las instancias (objetos) de una clase funcionarán de una forma similar. Una clase es un tipo de *dato abstracto*. Un tipo de dato abstracto es aquel definido por el programador que puede ser manipulado de una manera similar a los definidos por el sistema. Al igual que estos últimos, un tipo de datos abstracto corresponde a un conjunto de valores lícitos y de un número de operaciones primitivas que pueden ejecutarse sobre ellos. Un grupo de objetos que tienen el mismo conjunto de operaciones y la misma representación de estado se consideran instancias de la misma clase.

La característica principal de la programación orientada a objetos es que permite *abstracción*, *modularidad* y *reuso* de código: La abstracción es la capacidad de encapsular y aislar la información de diseño y la ejecución de programas, como se hace con los procedimientos y funciones en la programación imperativa y funcional, o con los módulos en la programación orientada a objetos. Un módulo pueden considerarse simplemente como un bloque de código que esta compuesto por dos partes: una pública, accesible desde fuera del módulo y una privada, que sólo es accesible desde dentro del módulo. El reuso de código es otra de las características importantes de la programación orientada a objetos. En el pasado, la reutilización del software era una meta muy buscada y pocas veces alcanzada. Una razón muy importante para ello es la estrecha interconexión de la mayoría de los programas que se crearon en forma convencional. Tales interdependencias pueden ser el resultado de definiciones de datos o pueden ser dependencias funcionales.
Los lenguajes de programación orientada a objetos proveen un mecanismo para separar con limpieza la información esencial de la información de poca importancia. De esta manera, usando técnicas orientadas a objetos, podemos construir grandes componentes de software reutilizable.

Dentro de los lenguajes de programación orientada a objetos, uno de los más conocidos y utilizados en la actualidad, es el C++ [2], el cual es un lenguaje derivado del lenguaje C, desarrollado por Bjarne Stroustrup en los laboratorios Bell de AT&T.

El C++ incorpora conceptos nuevos a los ya existentes en el lenguaje C, como son: clases, objetos, funciones miembro, constructores, destructores, herencia, etc. A continuación se da una breve explicación de los mismos:

- Una ***clase*** es una estructura de datos que contiene un conjunto de funciones, llamadas funciones miembro, que manejan a esa estructura. Una clase consta de:
  - Una representación concreta de los objetos y
  - Un conjunto de funciones miembros u operaciones para manejo de los objetos.

- Una ***función miembro*** es una función de una clase que determina los métodos o comportamientos que puede tener un objeto.

- Un ***objeto*** es una entidad o una instancia de una clase, que encapsula información o datos y un conjunto de operaciones asociadas o procedimientos que manipulan esos datos. El estado de un objeto se caracteriza por una colección de propiedades y se puede modificar mediante la ejecución de las funciones miembro de la clase a la que pertenece. Dichas operaciones constituyen el único medio de interacción del objeto con el exterior.

- El ***constructor*** es una función miembro de una clase y es la característica más importante que C++ añadió a C. Proporciona al programador una herramienta sencilla, pero poderosa, para definir nuevos tipos de datos o clases. Esta función miembro tiene la característica de tener siempre el mismo nombre de la clase y generalmente se utiliza para inicializar las variables de la clase. Toda clase debe tener forzosamente un constructor.

- El ***destructor*** de clases es una función miembro que sirve para borrar un objeto cuando su uso deja de tener sentido, tiene el mismo nombre que la clase pero antecedido de un símbolo "~".

- La ***herencia*** es una de las características importantes que permite la reutilización de código. No es raro que existan clases con algo en común. Al definirlas por separado, el código se duplica de manera considerable, lo que es un desperdicio de memoria y esfuerzo. El lenguaje C++ tiene la capacidad de obtener una nueva clase a partir de una ya existente, evitando con esto la duplicidad de código.

En la siguiente parte se explica qué es la programación distribuida y cuáles son algunos de los lenguajes más representativos que se encontraron durante la investigación realizada.

2.2 La Programación Distribuida

Un punto importante de la reciente investigación y desarrollo de la computación ha sido el desarrollo de los sistemas distribuidos. Con el surgimiento de redes de área local de alta velocidad y de la alta disponibilidad de computadoras personales, estaciones de trabajo y servidores, se ha acelerado el desarrollo de la investigación en esta área.

Los sistemas distribuidos pueden ser usados como sistemas computacionales interactivos de propósito general para soportar un amplio rango de aplicaciones comerciales e industriales. Se aplican en áreas tales como servicios de información en red y aplicaciones multimedia, donde la comunicación es uno requerimientos básicos.

Según varios autores, la *programación distribuida* se puede definir como aquella que nos permite crear sistemas (distribuidos) que se ejecuten sobre una colección de computadoras autónomas, físicamente dispersas e interconectadas a través de una red [16,29].

Un *sistema distribuido* se define como un conjunto de computadoras autónomas ligadas por una red y equipadas con software de sistemas distribuidos. Este software permite a las computadoras coordinar sus actividades y compartir los recursos del sistema: hardware, software y datos.

Los usuarios de un sistema distribuido bien diseñado deben percibir una sola computadora integrada, aunque esta puede estar construida por muchas computadoras ubicadas físicamente en distintos lugares.

Los sistemas creados con *lenguajes de programación distribuida* pueden ser divididos en dos grandes grupos: Los *sistemas distribuidos de propósito general* y los *sistemas operativos distribuidos*. Dentro de los sistemas de propósito general tenemos a aquellos sistemas para grupos de usuarios como son los sistemas bancarios, sistemas de comunicación multimedia, sistemas de reservaciones en líneas aéreas y los sistemas para cajeros automáticos de los bancos, entre otros. Uno de los requerimientos más importantes de tales aplicaciones es contar con un alto nivel de seguridad y privacidad de la información que el sistema mantiene.

Por otro lado están los sistemas operativos distribuidos, que son distinguidos de los sistemas operativos convencionales como UNIX, principalmente en que los sistemas operativos distribuidos permiten que un conjunto de computadoras sean tratadas como una sola entidad.

Según varios autores, las características más importantes de los sistemas distribuidos son: Permiten compartir los recursos, permiten la escalabilidad, son tolerantes a fallas, son transparentes y seguros, entre otras.

Como ya se mencionó anteriormente, existen pocos lenguajes creados expresamente para la programación distribuida, sin embargo a lenguajes de programación como ADA [11,12,13] o C y a sistemas como PVM se les han agregado mecanismos que nos ayudan a realizar aplicaciones bajo ambiente distribuido, pero estas son un poco difíciles de entender y utilizar. Además, la mayoría de estos lenguajes son poco conocidos. Quizás de todos ellos el más conocido y utilizado actualmente es C++.

En los últimos años se han tratado de establecer estándares para la programación de sistemas distribuidos, entre los que se sobresalen CORBA/OpenDoc de OMG (Object Managament Group) y COM/OLE de Microsoft. Estos estándares presentan un nuevo paradigma conocido como *objeto distribuido*, que permitirá crear aplicaciones cliente/servidor para las nuevas tecnologías de hardware. En general, un objeto distribuido es un componente que tiene "inteligencia" y la capacidad de empaquetamiento. Estas características permiten subdividir aplicaciones en componentes autocontrolados que pueden interactuar a través de una red. En este trabajo no se tratan éstos estándares; si se desea más información al respecto se puede consultar Orfail [32].

2.3 Programación Distribuida Orientada a Objetos

El modelo de programación distribuida orientada a objetos enfatiza la organización de un programa en objetos relativamente autónomos e interactuantes llamados procesos y combina dos áreas:

- Programación Orientada a objetos
- Programación concurrente y distribuida

La unión de estos dos conceptos ha resultado en lenguajes que se conocen como ***lenguajes de programación distribuida orientado a objetos***.

Los elementos principales de estos lenguajes de programación son:

- Atomicidad
- No-determinismo
- exclusión mutua
- sincronización de condición
- objetos y clases
- Herencia

Para el desarrollo del presente trabajo se estudiaron algunos de sistemas programados con lenguajes de programación distribuidos orientados a objetos, con el fin de conocer sus principales características, ventajas y deficiencias.

Los sistemas distribuidos orientados a objetos hacen posible un ambiente en el cual los programas consisten en un conjunto de módulos interactivos u objetos, que se pueden ejecutar concurrentemente sobre una colección de procesadores acoplados.

Los Sistemas de Programación Distribuida Orientados a Objetos más difundidos han sido reportados por Chin [9], estos son:

AMOEBA, ARGUS, CHORUS, CLOUDS, EDEN, EMERAL y TABS/Camelot

El estudio preliminar de estos sistemas permitió conocer que características eran necesarias agregar al lenguaje C++, para facilitar la programación de sistemas similares a los mencionados.

En la investigación se encontraron características generales y comunes en casi todos los sistemas estudiados. La implantación de tales características, varía dependiendo del sistema, pero el objetivo entre ellas es muy similar [9]:

- Facilitan la administración de los objetos para identificarlos y usarlos eficientemente en forma cooperativa.

- Combinan estaciones de trabajo independientes y posiblemente heterogéneas para ofrecer un ambiente de procesamiento distribuido.

- Ofrecen alta disponibilidad de los objetos en las aplicaciones.

- Garantizan la recuperación automática de los objetos cuando falla una estación de trabajo.

- Permiten al propietario del objeto especificar los clientes que puede atender.

- Ofrecen la posibilidad de asignar a los objetos de un programa varios procesadores para que puedan ejecutarse concurrentemente.

- Ocultan detalles irrelevantes para las aplicaciones ofreciendo un acceso uniforme a todos los objetos locales o remotos.

- Garantizan la integridad de un objeto asegurando que siempre se encuentre en un estado válido.

- Aseguran que se anulen todos los cambios hechos al estado del objeto cuando una operación no termine exitosamente.

- Ofrecen continuidad de procesamiento a pesar de las fallas en una estación de trabajo o en un objeto

- Permiten que cada objeto sea capaz de atender concurrentemente las operaciones solicitadas por otros, de modo que no interfieren entre sí.

- Ofrecen la posibilidad de que un programa bien diseñado corra más rápido que en un sistema convencional.

Actualmente son pocos los lenguajes de programación que permiten desarrollar sistemas con las características anteriores. La mayoría de estos lenguajes son poco conocidos y utilizados, por lo que varios investigadores han desarrollado lenguajes de programación basados en lenguajes conocidos como el C, haciendo extensiones que facilitan la programación de sistemas distribuidos y la programación de sistemas distribuidos orientados a objetos.

A continuación se muestra una breve descripción de los lenguajes basados en C, que se encontraron durante la investigación realizada para el desarrollo de esta tesis.

Dentro de estos trabajos tenemos el C Concurrente [1,15], que esta basado en el modelo de Procesos Secuenciales Comunicantes (CSP) de Hoare y que utiliza la instrucción SELECT [11,12,13,14] y la sentencia asíncrona *send*, el C++ Concurrente [2], que utiliza la clase *process* para la creación de procesos, la sentencia *Select-When*, y la comunicación bidireccional entre procesos utilizando variables compartidas; y el C++ Distribuido [3,4] que es un lenguaje para escribir aplicaciones paralelas en C++ sobre sistemas distribuidos.

2.4 Propuesta

Como se vio en el punto anterior, existen muy pocos lenguajes de programación distribuida orientada a objetos que, además, no son muy conocidos por la mayoría de los programadores comunes.

La propuesta que se hace en esta tesis es la de agregar las características necesarias para facilitar la programación distribuida a un lenguaje conocido y que además sea muy utilizado actualmente en un amplio rango de sistemas. El lenguaje seleccionado fue el C++, ya que es más conocido que lenguajes como ADA, OCAM o JAVA y, además maneja los conceptos de la programación orientada a objetos que, como ya se mencionó, facilitan la programación estructurada porque descompone al sistema en objetos con interfaces bien definidas.

Siguiendo como base algunos de los trabajos presentados anteriormente y principalmente, el lenguaje ***C++ Concurrente***[2]; al lenguaje C++ se le agregaron ciertas características encontradas en lenguajes como ADA, C++ Distribuido, C++ Concurrente y en sistemas como PVM, AMOEBA, MACH, entre otros, para facilitar al programador de sistemas la programación de sistemas distribuidos orientados a objetos. A diferencia del Lenguaje C++ Concurrente [2],que fue desarrollado sobre el sistema operativo DOS, el ***C++ Distribuido*** fue desarrrollado sobre UNIX, lo cual permite que el control y comunicación de los procesos sea más natural que el que realiza el C++ Concurrente[2].

Las características que se agregaron al lenguaje C++, consisten en clases y construcciones sintácticas adicionales para introducir procesos que se comportan como objetos, guardias, no-determinismo y paso de mensajes.

El resultado de este trabajo es un lenguaje de programación que llamamos ***C++ Distribuido.***

El ***C++ Distribuido*** es un lenguaje basado en C++, lo cual facilita su uso a los programadores que conocen el C++ por que no tienen la necesidad de aprender un nuevo lenguaje. Además cuenta con todos los beneficios de la programación orientada a objetos.

Las ventajas que tiene el lenguaje ***C++ Distribuido*** con respecto al lenguaje C++ común, son que el primero presenta mecanismos para el control de concurrencia y exclusión mutua, comunicación entre procesos y creación de los mismos de una forma más abstracta que el segundo. De esta forma la programación se vuelve más modular, sencilla y rápida.

El lenguaje ***C++ Distribuido*** tiene como principal base al ***C++ Concurrente*** [2]. Este último es un lenguaje que cuenta con similares extensiones a las presentadas en este trabajo, sin embargo, presenta los siguientes problemas: corre sólo sobre una computadora, el control de procesos lo realiza con un planificador propio de este lenguaje, el cual utiliza cierto tiempo para controlar los procesos, esto hace más lentas las aplicaciones. El lenguaje ***C++ Distribuido*** elimina estas limitaciones, ya que se pueden crear aplicaciones que se ejecuten en varias máquinas que tengan UNIX. Además la gestión de procesos es dejada al

sistema operativo, lo cual hace que esta actividad se realice más rápido que en el ***C++ Concurrente*** [2].

En el siguiente capítulo, se explican ampliamente las extensiones realizadas al C++ que conforman este nuevo lenguaje, el ***C++ Distribuido***.

3. El C++ Distribuido

3.1 Generalidades

Muchas de las aplicaciones complejas, tales como aplicaciones multimedia, aplicaciones en tiempo real y aplicaciones en red, pueden obtener beneficios de las características de la programación orientada a objetos, como son: Abstracción, modularidad, extensibilidad y reuso de código. Sin embargo, los lenguajes orientados a objetos no proporcionan soporte para la computación distribuida. Esto hace que la programación orientada a objetos en sistemas distribuidos tenga muchas dificultades.

Como se explicó en los capítulos anteriores, muchos han sido los investigadores que han intentado desarrollar lenguajes que permitan combinar los conceptos de programación orientada a objetos y la programación distribuida. Sin embargo, muchos de estos lenguajes han sido poco difundidos entre los programadores de sistemas.

El ***C++ Distribuido*** es una extensión al lenguaje C++ que pretende facilitar al programador la creación de sistemas distribuidos orientados a objetos. Con esta extensión se conjuntan los dos paradigmas más importantes de la programación actual, en cuanto a su capacidad para tratar problemas complejos: La programación orientada a objetos y la programación distribuida.

Una meta importante del diseño del ***C++ Distribuido*** es crear en el programador la ilusión de que programa en un lenguaje que posee nuevas construcciones que son consistentes con el lenguaje C++. Esto hace que los programas escritos en ***C++ Distribuido*** sean más fáciles de entender para aquellos programadores que conocen el lenguaje C ó C++.

Las extensiones hechas al lenguaje C++, descritas en este trabajo, consisten en clases y construcciones sintácticas adicionales para introducir procesos, guardias, no-determinismo y paso de mensajes.

Las características principales de este lenguaje ***C++ Distribuido*** son las siguientes:

1. Abstracción de datos mediante clases.
2. Creación de procesos a través de una clase.
3. Ejecución no determinista de programas.
4. Comunicación síncrona mediante mensajes.
5. Control de concurrencia.

3.2 Palabras Reservadas

El lenguaje ***C++ Distribuido*** añade las siguientes palabras reservadas a la lista del lenguaje C++:

1. *Process*
2. *cobegin*
3. *select*
4. *when*
5. *channel*
6. *cpp_send*
7. *cpp_receive*

Process es la nueva clase introducida al C++, que nos permite crear procesos. La palabra reservada *cobegin* nos permite iniciar la ejecución simultánea de todos los procesos. Las palabras reservadas *select* y *when* se utilizan como una instrucción no determinista adicional al C++ que permiten sincronizar procesos y será explicada con detalle más adelante, y por último; las funciones *cpp_send* y *cpp_receive* permiten la comunicación entre procesos a través de paso de mensajes. Estas funciones miembro pertenecen a la clase *channel.*

En las siguientes secciones se describirán a detalle cada una de estas nuevas instrucciones, así como el uso de ellas.

3.3 La Clase Process

En la investigación realizada sobre los lenguajes distribuidos orientados a objetos más conocidos, encontramos que la mayoría de ellos cuentan con algún mecanismo de creación de procesos, en donde estos se comportan como objetos. Es importante contar con un mecanismo similar, porque la definición, creación y control de procesos se vuelve más fácil y rápida que en lenguajes que no disponen de tal mecanismo. Es por ello que se pensó en la creación de una clase que permitiera esto, a la cual llamamos la clase *process.*

La clase *process* unifica los conceptos de objeto y proceso, al mismo tiempo que mantiene sus diferencias. De acuerdo con su comportamiento los objetos se pueden distinguir en pasivos o activos. Los objetos pasivos corresponden a la noción usual de colección de atributos y operaciones como en el C++ común y corriente. Los objetos activos extienden esta noción hasta considerarlos auténticos programas secuenciales con su propio control de ejecución, como los procesos que usa este lenguaje.

En lo sucesivo llamaremos ***objetos-proceso*** a los procesos creados utilizando la clase *process* del lenguaje C++ Distribuido.

3.3.1 Especificación de Procesos

La especificación de una clase de procesos establece la estructura interna y el comportamiento de un proceso mediante la declaración de variables y de funciones. Un ***objeto-proceso*** es un programa secuencial que hereda de la clase *process* la capacidad de comportarse como un proceso común de UNIX.

La especificación de cualquier clase de procesos debe incluir al menos un constructor, como se muestra en el siguiente esquema:

```
class X :: process
  {
      variable 1;
          :
      variable n;
      X(); //Constructor de la clase X
      funcion_miembro_1();
          :
      funcion_miembro_N();
  };
```

La especificación puede o no tener funciones miembro además del constructor o constructores. La especificación de un proceso introduce un tipo definido por el programador que es consistente con los tipos provistos por el lenguaje (como los tipos básicos). Por ejemplo, la clase *Writer*, que especifica un proceso escritor, se especificaría de la siguiente forma:

Ejemplo:

```
class Writer : process{
                int c ;
                public :
                Writer(char* ,int) ;        // Constructor
                };
```

en donde, *c* es una variable propia e independiente de cada proceso. Esta especificación permite declarar variables de tipo *Writer*, así como apuntadores, arreglos y referencias a ellas.

3.3.2 Definición de Procesos

La definición de una clase de procesos se establece en el cuerpo de la función constructora. El comportamiento de ***objeto-proceso*** lo establece el texto de esta función. Diferentes funciones constructoras inducen comportamientos diferentes en procesos de la misma clase o de clases distintas. Por ejemplo, si se tienen dos funciones constructoras en una clase llamada "Lectores" y la primera de ellas indica que va a leer un archivo y la otra indica que va a leer de un arreglo de caracteres en memoria, el ***objeto-proceso*** creado con el primer constructor leerá de un archivo y el otro de un arreglo de caracteres en memoria, respectivamente. Lo mismo sucede con ***objetos-proceso*** creados por distintas clases, por ejemplo dos clases llamadas "Lector" y "Escritor", deberán tener cada una al menos un constructor, llamados "Lector" y "Escritor", respectivamente. El constructor "Lector" lee de un archivo y el constructor "Escritor" escribe en un archivo y estos comportamientos serán los que tengan el ***objeto-proceso*** "Lector" y "Escritor", respectivamente.

Un ***objeto-proceso*** inicia su actividad ejecutando la primera instrucción del bloque de la función constructora y termina luego de ejecutar la última instrucción que aparece en ella. En otras palabras, la duración del proceso es la misma que la del bloque que lo define, es decir del constructor. Por ejemplo, el comportamiento de la clase de procesos *Writer* puede ser de la siguiente forma:

Ejemplo:

```
// Constructor de la clase Writer
Writer : : Writer(char * s, int n)
        {
                cout << "Proceso " << s << "inicia su actividad \n" ;
                for (c = 0 ; c<n ; c++)
                    {
                        cout << s << c ;
                    }
                cout << "Proceso "<< s << " termina su actividad \n";
        }
```

3.3.3 Declaración de Objetos-proceso

Un ***objeto-proceso*** es una instancia de alguna clase derivada de *process*. En un programa pueden coexistir diferentes instancias de la misma clase derivada. Por ejemplo, los ***objetos-proceso*** a, b y c, son instancias de la clase *Writer* que a su vez es derivada de la clase *process*, en el siguiente programa:

Ejemplo:

```
main()
  {
        Writer a("Escritor A :", 4) ;
        Writer b("Escritor B :", 3) ;
        Writer c("Escritor C :", 2) ;
        cobegin;
   }
```

Cuando se declara una variable de cierta clase de procesos, se crea automáticamente un ***objeto-proceso*** aunque esto no significa que su actividad comience de inmediato. El inicio de su actividad concurrente tiene lugar con la declaración de la instrucción *cobegin*, que será explicada más adelante Se considera que todo proceso que sea declarado antes de la instrucción de *cobegin*, es susceptible de ponerse en ejecución. En adelante, si algún proceso A en ejecución declara otro proceso B, entonces B está listo para iniciar su actividad en cualquier momento.

3.4 La Instrucción Select-When

Si se desea que varios procesos resuelvan cooperativamente una tarea, es necesario proveer mecanismos de concurrencia y sincronización. El diseño de estos mecanismos impacta en el desempeño del sistema, facilitando la programación y prueba de programas y determinando el nivel de competencia por los recursos en ambientes competitivos.

La concurrencia es importante por varias razones. Primero, muchos sistemas físicos son modelados más naturalmente por procesos concurrentes. Por ejemplo, consideremos un sistema bancario, o las funciones de varios departamentos en una organización o el sistema de reservaciones de una aerolínea. Todos estos claramente involucran actividades concurrentes. Segundo, la ejecución puede acelerar el tiempo de procesamiento requerido en alguna aplicación. Por ejemplo, puede resolver un gran sistema de ecuaciones, si el problema puede ser dividido y un algoritmo determina qué procesos pueden trabajar en porciones del problema, logrando la solución en menos tiempo que si se dejara este trabajo a un solo proceso.

Cuando se tienen muchos procesos ejecutándose en un sistema concurrente, es imposible predecir el orden de aparición de sus eventos, por ésta razón se debe contar con mecanismos eficientes para la sincronización de los procesos concurrentes.

Todos los lenguajes concurrentes y distribuidos, sin importar el modelo y el estilo que sigan deben incluir alguna forma de introducir sincronización de procesos, concurrencia y no-determinismo.

En este trabajo incluimos un mecanismo para el modelado del no-determinismo y la sincronización de procesos, muy poderoso y sencillo basado en una construcción introducida por Dijkstra. Esta construcción es conocida como "Comandos Guardados" o "Comandos con guardias" y ha sido la base para instrumentar mecanismos de control de concurrencia en muchos sistemas.

Los comandos con guardias tienen la forma G→C, donde G es un guardia y C una lista de instrucciones o comandos. Un guardia consiste de expresiones booleanas y declaraciones posiblemente sucedidas por un comando de entrada. Los guardias fallan si cualquiera de las expresiones booleanas es falsa. La lista de comandos es ejecutada si la expresión booleana es verdadera. Los comandos con guardias pueden combinarse dentro de un comando alternativo de la siguiente forma:

$$[\ G1 \rightarrow C1 \quad G2 \rightarrow C2 \quad \ldots \quad Gn \rightarrow Cn\ ]$$

la cual especifica la ejecución de exactamente uno de los comandos guardados. Consecuentemente, si todos los guardias fallan, el comando alternativo falla. Si más de un guardia es ejecutable, se selecciona uno arbitrariamente. El programador no tiene el control sobre los procesos seleccionados.

Las ***regiones protegidas con guardias*** son bloques de instrucciones precedidos por guardias. permiten a un proceso seleccionar una serie de acciones de entre varias conforme al estado interno del proceso. Si no es posible seleccionar alguna de ellas, se pospone la elección hasta que alguno de los guardias sea verdadero.

Los comandos con guardias se han utilizado como base en la implantación de un mecanismo para la sincronización de procesos y el modelado del no-determinismo en lenguajes de programación como ADA, pl/1, C Concurrente, C++ Concurrente, entre otros. Este mecanismo es conocido como *Select-When* o simplemente *SELECT*. Esta instrucción es la que se incorporó al lenguaje ***C++ Distribuido***. La instrucción consiste de un bloque de instrucciones que se ejecutan sin interrupción siempre que se verifique la condición que le precede.

La instrucción *select-when* contiene los mecanismos necesarios para suspender la ejecución del solicitante cuando ningún guardia es verdadero, y para reactivarlo mas tarde con el fin de intentar de nuevo. Eventualmente, cuando se verifica alguna condición, el control ejecuta una serie de acciones que acompañan a la condición y termina la instrucción. Por otra parte, cuando es posible elegir más de una alternativa, la elección se hace en forma no determinista.

Es importante mencionar que si se tiene un solo proceso en ejecución y todos los guardias son falsos, este proceso quedará bloqueado indefinidamente. En este caso no se necesita utilizar la instrucción *select-when*, ya que no se requieren mecanismos de sincronización porque solo hay un proceso trabajando.

En el lenguaje C++ Concurrente[2], las regiones protegidas con guardias sólo pueden emplearse en la definición de las funciones componentes de los procesos. La implantación del lenguaje ***C++ Distribuido*** elimina esta limitación, ya que las regiones protegidas pueden aparecer en cualquier lugar del programa. Cuando la instrucción *select-when* aparece en la función constructora, el propio proceso es el único que puede suspender su ejecución con dicha instrucción, por otra parte, cuando aparece en los servicios, los procesos que lo soliciten son los que se pueden suspender.

3.4.1 No-Determinismo

Como ya se ha explicado anteriormente, el no-determinismo es un elemento que se encuentra implícito en la mayoría de las aplicaciones reales como por ejemplo: los servicios de un banco, o las funciones de varios departamentos en una organización gubernamental, o un sistema de reservaciones de una aerolínea. Todos estos claramente involucran el no-determinismo.

EL no-determinismo sucede cuando de entre varios eventos sucede uno al azar. Los comandos con guardias y por lo tanto, la instrucción *select-when* incluyen el no-determinismo para probar los guardias y cuando en un programa existen dos o más guardias verdaderos y de ese conjunto se elige solo uno al azar para ejecutar las instrucciones que le siguen.

3.4.2 Sintaxis y Semántica

La instrucción *select-when* tiene la siguiente estructura sintáctica:

select-when → **select** *clausula_when* | **select** { *lista_clausulas_when* }

lista_clausulas_when→*clausula_when*| *lista_clausulas_when*; *lista_clausulas_when*

clausula_when → **when** (*expresion_booleana*) *instrucciones*

La instrucción comienza con la palabra reservada *select* seguida de una o varias cláusulas *when*. Una cláusula *when* consiste de una o varias instrucciones precedidas por un guardia. El guardia es una expresión lógica (condición) que no debe producir efectos colaterales.

Informalmente, su semántica se describe como sigue:

1. Un proceso puede ejecutar la instrucción *select-when* siempre que ningún otro la esté ejecutando. Si existen varios procesos tratando de ejecutar la instrucción, se escoge uno al azar.

2. Un proceso se ve obligado a posponer la ejecución de la instrucción cuando otro la esta ejecutando.

3. Cuando un proceso completa la instrucción *select-when* y el bloque de instrucciones que siguen a esta instrucción, se permite que otro que está en espera la ejecute.

4. Un proceso ejecuta la instrucción probando todos los guardias una sola vez, cada que ejecuta la instrucción *select*. La prueba de los guardias se realiza en forma no determinista, es decir, independientemente del orden en el que aparecen en el código.

5. Si se encuentra al menos un guardia cuya condición se cumpla, entonces el proceso ejecuta el bloque de instrucciones que le siguen sin interrupción. Cuando termina de ejecutarlas, termina de ejecutar la instrucción.

6. Si ningún guardia posee una condición que se cumpla, el proceso suspende la ejecución de la instrucción y el proceso se bloquea hasta que posteriormente le toque su turno de volver a probar los guardias. Enseguida se permite que otro proceso que desee ejecutar la instrucción *select-when*, lo haga.

7. Cuando existe mas de un proceso deseando entrar a la región protegida, la elección del proceso se hace al azar.

La implantación de esta instrucción debe asegurar la imparcialidad en la elección del proceso que va a ejecutar la instrucción.

Evidentemente esta solución evita la interferencia entre procesos, pero puede impedir el progreso en la ejecución de un proceso (dead-lock) cuando recurrentemente ninguno de los guardias se cumpla.

La instrucción *select-when* introduce acciones atómicas y no-determinismo durante la ejecución del programa concurrente o distribuido. Las acciones atómicas aseguran que un objeto transite únicamente entre estados validos visibles externamente.

La instrucción *select-when* produce el efecto de serializar el acceso de los procesos a datos compartidos o métodos públicos.

Aunque es posible resolver los problemas de sincronización de procesos usando semáforos, se prefirió resolverlo desde una perspectiva más abstracta usando la instrucción *select-when*.

A continuación se muestra un ejemplo de cómo se utiliza esta instrucción. Dos tipos de procesos, llamados lectores y escritores, comparten un recurso. Los lectores pueden usar el recurso simultáneamente, pero cada escritor debe tener acceso exclusivo a él. Los lectores y escritores se comportan como sigue:

Una variable S define el estado actual del recurso como uno de los siguientes:

S=0 1 escritor usa el recurso
S=1 0 procesos usan el recurso
S=2 1 lector usa el recurso
S=3 2 lectores usan el recurso
...

El proceso *Resource* regula el paso de los lectores *Reader* y de los escritores Writer mediante las operaciones *startread, endread, startwrite, endwrite*, respectivamente.

Este ejemplo utiliza dos de las herramientas del C++ Distribuido, que son la clase *process* y la instrucción *select-when*.

```
1    #include "cppd.h"
2    #define TIME 500
3
4    // Especificación de una clase de procesos Resource.
5    // Esta clase representa el recurso compartido por los lectores y los escritores.
6
7    class Resource : process
8    {
9    int s;
11   public:
12   Resource(void);
13   Void startread(void);
14   Void endread (void);
15   Void startwrite (void);
16   Void endwrite (void);
17   };
18
19   // Especificación de una clase de procesos Reader. Esta clase representa al lector.
20
21   class Reader : process
22           {
23   public:
24   Reader(int, Resource&);
25   };
26
```

```
27    // Especificación de una clase de procesos writer. Esta clase representa al Escritor.
28
29    class writer: process
30    {
31    public:
32    writer (int, Resource&);
33    };
34
35    // El constructor de la clase de procesos Resource únicamente inicializa la variable s.
36
37    Resource:: Resource(void):Process
38    {
39            s =1;
40    };
41
42    void Resource:: startread (void)
43    {
45    Select When (s >= 1)
46            s++;
47    EndSelect;
48    };
49
50    void Resource:: endread (void)
51    {
52       s - -;
53    };
54
55    void Resource:: startwrite (void)
56    {
57    Select When (s == 1)
58            s =0;
59    EndSelect;
60    };
61
62    void Resource:: endwrite (void)
63    {
64       s = 1;
65    };
66
```

```
67
68      Reader:: Reader(char* id, Resource& book) [
69        for(int i=0; i < 10; i ++)
70          {
71                cout << "Lector %s solicita recurso   " << id;
72                book.startread();
73                cout << "Lector %s usando el recurso   " << id;
74                delay(TIME);
75                book.endread();
76                cout << "Lector %s libera el recurso   " << id;
77           }
78      }
79
80      Writer:: Writer (char* id, Resource& book)
81      {
82      for(int i=0; i < 10; i ++)
83          {
84                cout << "Escritor  %s solicita recurso   " << id;
85                book.startwrite();
86                cout << "Escritor %s usando el recurso   " << id;
87                delay(TIME);
88                book.endwrite();
89                cout << "Escritor %s libera el recurso   " << id;
90          }
91      }
92
93      main()
94      {
95       Resource book;
96      Reader (1, book); Reader (2, book); Reader (3, book);
97      Writer (1, book); Writer (2, book);Writer (3, book);
98      Cobegin;
99      }
```

Figura 3.1 Programa de Lectores- Escritores usando la instrucción select-when.

Se puede observar en el programa anterior que la programación de la exclusión mutua para accesar a la variable compartida s, es muy sencilla y clara; basta con incluir la instrucción *select-when* entre las instrucciones que se desea sean accesadas en forma exclusiva por un proceso.

Como ya se mencionó anteriormente, es posible utilizar semáforos para controlar al acceso exclusivo a una región crítica, en lugar de utilizar la instrucción *select-when*. Sin embargo, la utilización de semáforos no es tan sencilla como la instrucción *select-when* y es más

susceptible de errores por parte del usuario. Obsérvese detenidamente el orden de las operaciones sobre un semáforo para el control de acceso a una región crítica en el siguiente código:

```
semaforo.sem_num = MUTEX;
semaforo.sem_op = 0;        // Operación Down
semaforo.sem_flg = 0;
semop(semid,&semaforo,1);

        REGION CRITICA.....

semaforo.sem_num = MUTEX;
semaforo.sem_op = 1;        // Operación Signal
semaforo.sem_flg = 0;CC
semop(semid,&semaforo,1);
```

Supóngase que el valor de la primera instrucción *semaforo.sem_num* sea igual a uno (*signal*) en lugar de cero(*down*) esto provocaría que dos o más procesos entraran a la región crítica. O supóngase que el valor del semáforo es cero antes de que un proceso ejecute la instrucción *semaforo.sem_num*. Cuando el proceso llega a esta instrucción, se bloquea indefinidamente, provocando un candado mortal.

Estos problemas se destacan para mostrar lo cuidadoso que se debe ser cuando se utilizan semáforos. Un error sutil puede ocasionar un problema grave en el sistema.

Para facilitar la escritura de programas correctos es que se propone la utilización de una primitiva de sincronización de más alto nivel como la instrucción *select-when*.

A continuación se explicará como se realizó la implementación de la comunicación entre procesos para el lenguaje C++ Distribuido.

3.5 Comunicación entre Procesos

En los sistemas distribuidos es de vital importancia contar con mecanismos de transmisión de información. En el caso de que estos sistemas sean orientados a objetos es necesario disponer de herramientas capaces de transmitir mensajes entre objetos. Este es el punto en que radican muchas dificultades para la construcción de sistemas distribuidos orientados a objetos.

Los sistemas distribuidos son, por naturaleza, complejas aplicaciones que requieren de la interacción de varios componentes, físicamente dispersas. Los mecanismos para intercambiar información entre nodos distantes son una de las principales fuentes de complejidad.

Los sistemas y lenguajes distribuidos orientados a objetos cuentan con modelos básicos de comunicación entre procesos por medio de mensajes. Aunque algunos de los sistemas manejan mecanismos de comunicación más complejos, éstos están basados generalmente en las funciones de comunicación entre procesos usando un esquema básico de envío y recepción de mensajes.

A partir de estos modelos de comunicación, surge la idea de desarrollar un sistema básico de comunicación entre procesos, a partir del cual se pudieran crear sistemas más complejos, pero que por si solo nos permitiera la comunicación y que además fuera simple y sencillo de utilizar.

Como nuestro lenguaje es orientado a objetos, es necesario tener una clase que permita la comunicación. La clase propuesta recibe el nombre de *channel* (canal). La clase *channel* tiene dos funciones miembro que permiten la comunicación bidireccional entre procesos. Estas funciones son *cpp_send()* y *cpp_receive()*.

Las funciones se usan de la siguiente forma: No hay mas que declarar un objeto del tipo *channel* y utilizar las funciones *cpp_send()* y *cpp_receive()*, para enviar y recibir mensajes; a través de un puerto virtual.

El papel de la función *cpp_receive()* es pasivo en el establecimiento de la comunicación, es decir que el proceso que ejecuta esta función, llamémosle receptor, se pone a la espera de mensajes que provengan del proceso o procesos que emite el o los mensajes, llamémosles emisores.

Un proceso emisor es la entidad que toma la iniciativa de la demanda de conexión a un receptor. Esta demanda se realiza por medio de la primitiva *cpp_send()*.

La función *cpp_send()* tiene los siguientes parámetros:

```
int cpp_send() (dir_serv, puerto, point_datos)
        char *dir_serv;
        int puerto;
        void *point_datos;
```

en donde:

dir_serv.- Es una cadena de caracteres que indican la dirección de la máquina con la que se desea conectar.

puerto.- Es un número entero que indica el número de puerto virtual en donde se depositarán los datos.

point_datos.- Es un apuntador a los datos que se desean transmitir.

Y la función *cpp_receive()* tiene los siguientes parámetros:

```
int cpp_receive(dir_serv, puerto, point_datos)
        char *dir_serv;
        int puerto;
        void *point_datos;
```

en donde:

Dir_serv.- Es una cadena de caracteres que indican la dirección de la máquina con la que se desea conectar.

Puerto.- Es un número entero que indica el número de puerto virtual de donde se leerán los datos.

Point_datos.- Es un apuntador a los datos que se desean leer.

El valor devuelto por ambas funciones es un número entero que indica si se logró o no la comunicación. Si el valor devuelto es 1, entonces las operaciones de envío y recepción de mensajes fueron exitosa. Si el valor devuelto por alguna de las dos funciones (o por las dos) es igual a –1, esto quiere decir que existió algún error en la comunicación de los datos.

Es importante hacer notar que el número de puerto debe ser el mismo en los dos procesos que se desean comunicar. Es decir si el proceso emisor escribe al puerto 5000, el proceso receptor debe leer de ese mismo número de puerto. De no ser así, la comunicación entre los procesos no se puede lograr.

Cabe mencionar que la función *ccp_receive()* es bloqueante, es decir, el proceso que ejecuta esta función (proceso receptor) se bloquea hasta recibir un mensaje. Una vez que el mensaje es recibido, el proceso continúa con su actividad. Por el contrario, si ningún mensaje es recibido durante cierto tiempo, la función devuelve el control inmediatamente indicando que se ha producido un error de conexión.

3.5.1 Un ejemplo de comunicación entre procesos

A continuación se presentan dos programas muy sencillos que se comunican usando las funciones *cpp_send()* y *cpp_receive()*, respectivamente.

El proceso emisor envía una cadena de caracteres con la palabra "HOLA", mientras que el proceso receptor recibe un mensaje con esa palabra, la lee y la muestra en pantalla. El proceso emisor se ejecuta en la máquina conocida como "pc-12" que tiene la dirección 148.247.2.205 y el proceso receptor se ejecuta en la máquina llamada "alpha" con la

dirección 148.247.2.5. El número de puerto utilizado es el 5000 y los dos procesos utilizan cada uno un arreglo para almacenar la cadena de caracteres.

Programa Emisor:

```
1     #include "cppd.h"
2
3     main()
4     {
5     channel c2;
6     int x;
7     char datos[4];
8
9     datos[0] = 'H';
10    datos[1] = 'O';
11    datos[2] = 'L';
12    datos[3] = 'A';
13
14    c2.cpp_send("148.247.2.205",5000,datos);
15
16    if (x == -1)
17      {
18       printf("ERROR");
19       exit(0);
20       }
21    else
22       {
23            printf("Los datos se enviaron bien");
24      }
25    }
```

Figura 3.2 Programa Emisor de un mensaje usando la clase channel.

La función *cpp_send()* en la línea 14 de la figura 3.2 tiene como parámetros de entrada, la dirección de la máquina con la que se va a comunicar, el número de puerto y el arreglo donde están los datos.

Programa Receptor:

```
1     #include "cppd.h"
2     main()
3     {
4     channel c1;
5     int y;
6     char dat[5];
7
```

```
8       y = c1.cpp_receive("148.247.2.205",5000,dat);
9
10      if (y == -1)
11        {
12          printf("ERROR\n");
13          exit(0);
14        }
15      else
16       {
17        printf("\nLos datos llegaron bien\n\n");
18        printf("%c", dat[0]);
19        printf("%c", dat[1]);
20        printf("%c\n\n", dat[2]);
21       }
22      }
```

Figura 3.3 Programa receptor de un mensaje utilizando la clase channel

La función *cpp_receive()* tiene como parámetros la dirección de la maquina en donde se encuentra el proceso emisor, el número de puerto de donde leerá los datos y el nombre de la estructura en donde depositara los datos recibidos.

Como se puede ver en los programas anteriores, la comunicación se vuelve mucho muy sencilla usando estas funciones, comparándola con la forma tradicional de programar la comunicación con sockets.

A continuación se presenta otro ejemplo que muestra la ventaja de tener estas funciones dentro de una clase. Este ejemplo muestra un como realizar una multiemisión de mensajes (*multicast*) a varias máquinas en donde ese encuentran otros programas receptores de estos mensajes.

El primer programa es el que realiza la emisión de los mensajes:

```
1       #include "cppd.h"
2
3       class multicast : public channel
4       {
5          public:
6                multicast(){};
7                envia_multicast();
8                channel::cpp_send; //aquí se indica que la función cpp_send se va a utilizar.
9       };
10
```

```
11    multicast::envia_multicast()
12    {
13    char mensaje[4];
14
15    mensaje[0]='H';
16    mensaje[1]='O';
17    mensaje[2]='L';
18    mensaje[4]='A';
19
20    channel::cpp_send("148.247.2.205",5000,mensaje);
21    channel::cpp_send("148.247.2.205",5001,mensaje);
22    channel::cpp_send("148.247.2.205",5002,mensaje);
23    };
24
25    main()
26    {
27    multicast mc;
28    int x;
29
30    mc.envia_multicast();
31    if (x == -1)
32      {
33       printf("ERROR");
34       exit(0);
35      }
36    else
37       {
38             printf("Los datos se enviaron bien");
39       }
40    }
```

Figura 3.4 Programa Multicast que envía mensajes y que muestra la utilidad de la herencia usando la clase channel.

Este proceso envía tres mensajes a tres procesos receptores que, en este caso, están corriendo sobre la misma máquina (la que tiene el número de dirección 148.247.2.205), pero están recibiendo el mensaje en diferente número de puerto. El mensaje que se envía es "HOLA". Si este mensaje es enviado correctamente, el proceso imprimirá un mensaje en la pantalla indicando que los datos se enviaron correctamente. En otro caso se indicara que ha ocurrido un error.

El siguiente programa muestra el código de uno de los receptores de uno de los tres mensajes, los otros dos son similares lo único que varía es el número de puerto donde leerán el mensaje y la dirección de la máquina donde corre el proceso con el que se van a comunicar, que en este caso también esta corriendo sobre la máquina con dirección 148.247.2.205:

```
1       #include "cmsg.h"
2
3       main()
4       {
5       channel c1;
6       int y;
7       char dat[5];
8
9       y = c1.cpp_receive("148.247.2.205",5000,dat);
10
11      if (y == -1)
12         { printf("ERROR\n");
13            exit(0);
14         }
15      else
16         {
17           printf("\nLos datos llegaron bien, Soy tres \n\n");
18           printf("%c", dat[0]);   printf("%c", dat[1]);  printf("%c\n\n", dat[2]); printf("%c\n\n", dat[4]);
21         }
22      }
```

Figura 3.5 Programa Receptor de uno de los mensajes enviados por el programa multicast

Para ejecutar estos programas, primero se deben ejecutar los programas que recibirán los mensajes y al último el proceso que los envía.

Si los datos llegan correctamente, los procesos receptores imprimirán un mensaje en la pantalla indicándolo, en otro caso se indicará que ha ocurrido un error.

Como se puede observar, la programación de la comunicación entre procesos en este ejemplo, se vuelve mucho más sencilla utilizando la clase *channel* y la herencia, que es uno de los beneficios que nos ofrece la programación orientada a objetos, ya que podemos hacer uso de código y programas que ya esta fueron hechos con anterioridad y no volverlos a hacer.

En los siguientes tres capítulos se describirá como se realizó la implementación de las extensiones hechas al C++, que conforman el nuevo lenguaje C++ Distribuido.

El capítulo cuatro describe como se realizó la implantación de la clase process, el capítulo cinco la de la instrucción *select-when* y el capítulo seis la de la clase *channel* y sus funciones de transmisión y recepción de mensajes.

4. La Clase Process

La clase *process* es una estructura que se agregó al lenguaje C++, la cual permite crear procesos como instancias de esta clase; es decir, crear objetos de tipo "proceso" (*objetos-proceso*), los cuales puedan ser manejados por el núcleo de la misma forma que un proceso común creado con la función *fork* en UNIX.

A continuación se describirá la implantación de esta clase. Primeramente se dará una descripción del comportamiento interno de un programa escrito en C++ para facilitar la explicación de esta implementación; y enseguida se describirá como se modificó el comportamiento normal de dicho programa para lograr la implementación deseada.

4.1 Implantación de la clase Process

Para entender mejor la forma en que se implantó esta clase, se dará una breve explicación de la estructura y el comportamiento de un programa ejecutable en UNIX.
La estructura de todo archivo ejecutable creado por los compiladores del C y C++ viene impuesta por el sistema. A grosso modo, un archivo ejecutable consta de las siguientes partes:

1. Un conjunto de cabeceras que describen atributos del archivo.

2. Un bloque donde se encuentran las instrucciones en lenguaje máquina del programa. Este bloque se conoce en UNIX como el texto del programa.

3. Un bloque dedicado a la representación en lenguaje máquina de los datos que deben ser inicializados cuando arranca la ejecución del programa. Aquí está incluida una indicación de cuanto espacio de memoria debe reservar el núcleo para estos datos. Tradicionalmente, este bloque se conoce como *bss* (pseudo operador del ensamblador del IBM 7090 que significa *block started by symbol*). El núcleo inicializa en tiempo de ejecución esta zona a valor 0.

4. Otras secciones, tales como tablas de símbolos.

Cuando un programa es leído del disco por el núcleo y es cargado en memoria para ejecutarse, se convierte en un proceso.

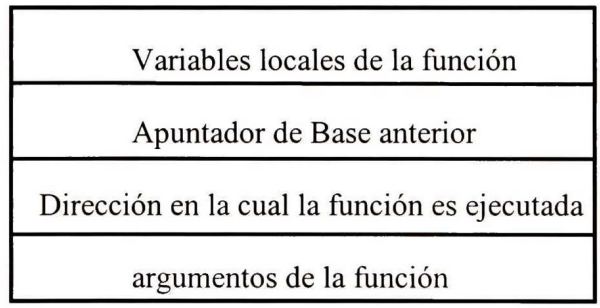
Un proceso se compone de tres bloques fundamentales que se conocen como segmentos. Estos bloques son:

1. El segmento de texto. Contiene las instrucciones que entiende la CPU de nuestra máquina. Este bloque es una copia del bloque de texto del programa.

2. El segmento de datos. Contiene los datos que deben ser inicializados al arrancar el proceso. Si el programa ha sido generado por un compilador de C, en este bloque estarán las variables globales y las estáticas. Este segmento corresponde al bloque *bss* del archivo que contiene el programa.

3. El segmento de pila (stack). Este segmento es creado por el núcleo al arrancar el proceso y su tamaño es gestionado dinámicamente por el núcleo. La pila se compone de una serie de bloques lógicos, llamados ***stack frames***, que son introducidos en la pila cuando se llama a una función y son sacados cuando de ella cuando se vuelve de la función. Un *stack frame* se compone de los parámetros actuales de la función invocada, la dirección de regreso a la instrucción siguiente a la llamada de la función, la dirección del Apuntador de Base BP (*Base Pointer*) anterior a la llamada de la función y las variables locales de la función, como se ve en la figura 4.1. El BP es un registro que sirve para hacer referencia a las variables locales y a los parámetros.

   En los programas fuente no se incluye código para administrar la pila (a menos que estén escritos en ensamblador); es el compilador quien incluye el código necesario para controlarla.



Stack Frame

Figura 4.1. Contenido de un Stack Frame.

Debido a que los procesos se pueden ejecutar en dos modos: usuario y núcleo(o supervisor); el sistema maneja dos pilas por separado. La pila de usuario y la pila del núcleo. La pila de usuario contiene los argumentos, variables locales y los otros datos relativos a funciones que se ejecutan en modo usuario.

La pila del núcleo contiene los *stack frames* de las funciones que se ejecutan en modo supervisor (estas funciones son las llamadas al sistema).

El ***planificador*** (*scheduler*) es la parte del núcleo encargada de administrar la CPU y determinar qué proceso pasa a ocupar tiempo de CPU en un determinado instante.

El contexto de un proceso refleja su estado, definido por su código, los valores de sus variables de usuario globales y sus estructuras de datos, el valor de los registros de la CPU, los valores almacenados en su entrada de la tabla de procesos y en su área de usuario, y el valor de sus pilas de usuario y núcleo.

Cuando se ejecuta un proceso, se dice que el sistema se está ejecutando en el contexto de un proceso. Cuando el núcleo decide que debe ejecutar otro proceso, realiza un cambio de contexto, lo que da lugar a que el núcleo guarde la información necesaria para poder continuar con la ejecución del proceso interrumpido en el mismo punto donde la dejó. De igual manera, cuando el proceso cambia del modo usuario al modo núcleo, se guarda la información para cuando el proceso tenga que regresar al modo usuario. Sin embargo, el cambio de modo usuario a núcleo y viceversa, no se contempla como un cambio de contexto.

Hechas las precisiones anteriores, ahora explicaremos como se implantó la clase *process*. El objetivo principal es tener una clase que permita la creación de procesos UNIX con sólo declarar objetos de la misma forma que se declaran objetos en C++. Para entender mejor como se desarrolló esta implantación, primeramente se explica el primer intento realizado en la creación de la clase *process*. Posteriormente se comentan los problemas encontrados y las soluciones dadas. Por último, se muestra la implantación definitiva de esta clase con las correcciones hechas. Es importante hacer notar que la idea básica de esta implantación se basó en la clase *process* del lenguaje C++ Concurrente [2].

A continuación se presenta el primer programa de la clase *process* en lenguaje C++ y tiene la siguiente estructura:

Ejemplo:

```
1    // Definición de la clase process.
2    class process
3    {
4    int x;
5    public;
6    process();    // Constructor de la clase process.
7    };
8
```

```
9     // Constructor de la clase process.
10    process::process()
11    {
12      int y;
13      if (getppid() != 1)
14          // Se crea un proceso...
15          y = fork();
16    }
17
18    // Definición de una clase  X derivada de la clase process.
19    class X : process
20    {
21    public:
22              X();  // Constructor
23    };
24
25    X :: X()
26    {
27    printf("HOLA  .....\n");
28    };
29
30    main()
31    {
32       X objeto1;  // objeto1 es una instancia de la clase X
33       :
34    }
```

Figura 4.2. Código del primer intento de la clase process.

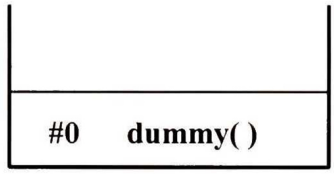
El flujo normal de este programa sería el siguiente: Inicialmente, el programa principal ejecutaría la primera instrucción de la función principal (línea 32 de la figura 4.2): *X objeto1*; que declara un objeto "*objeto1*" instanciado de la clase *X*, que a su vez es una clase derivada de la superclase *process*.

Después de ejecutar esta declaración, el apuntador de instrucciones ejecuta las instrucciones del constructor de la clase *X* (línea 25, figura 4.2), no sin antes meter en la pila el contexto de la función *main*. Pero, como la clase *X* es una clase derivada de *process*, la siguiente instrucción a ejecutar es la primera del constructor de la clase *process* (línea 12, figura 4.2), lo que provoca que ahora se introduzca en la pila el *stack frame* de la clase *X*. Dentro del constructor de la clase process se crea un proceso utilizando la función *fork()* en la línea 15 de la figura 4.2. Este último proceso creado es el que se desea ejecute las instrucciones del constructor de la clase *X*. Una vez que el proceso principal ha ejecutado las instrucciones del constructor de la clase *process*, el control regresa al constructor de la clase hija, es decir el constructor de la clase *X* (línea 27, figura 4.2). La dirección de regreso se encuentra en la pila. En este momento se ejecuta el constructor de la clase *X* y al terminar, el control pasa

de nuevo a la función *main*, obteniendo la dirección de regreso a ella de la pila. La siguiente instrucción que se ejecuta después de lo anterior, es la que se encuentra inmediatamente después de la declaración del objeto "*objeto1*" de la línea 32 de la figura 4.2 y si esta instrucción es una declaración de otro objeto, el flujo es el mismo descrito arriba; si no, entonces el programa continúa con su flujo normal.

En los siguientes esquemas se puede ver cómo se almacenan los *stacks frames* en la pila de acuerdo a las instrucciones que se ejecutan en el programa anterior:

El estado inicial de la pila al comenzar el programa se da en la figura 4.3:



PILA

Figura 4.3. Estado inicial de la pila.

La figura 4.3 muestra la pila con el *stack frame* de la función ***dummy ( )*** del núcleo que inicialmente no tiene nada, pero que al iniciar el programa mantendrá información del proceso principal creado por el núcleo, necesaria para su manejo.

En el momento que se inicia el programa, el núcleo crea un proceso que ejecuta todas las instrucciones de este programa y que llamamos **A**.

Como ya se mencionó anteriormente, la primera instrucción que se ejecuta es la declaración de un objeto llamado "*objeto1*" en la función *main* (Línea 32 de la figura 4.2). En este momento, el núcleo guarda en la pila el *stack frame* de la función *dummy()* y los datos de la función *main()*, ya que la declaración de cualquier objeto provoca que el control se mande al constructor de la clase de la cual el objeto es instanciado. La figura 4.4 muestra la pila después de la llamada al constructor:

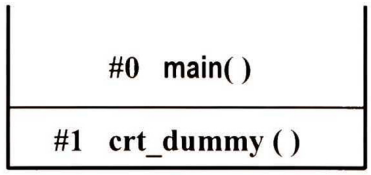


Figura 4.4. Estado de la pila al momento ejecutar la declaración del objeto.

Una vez declarado un objeto, el proceso **A** llama a la función constructora de la clase *X*, provocando que en la pila se inserte el *stack frame* de la función *main()* y los datos del

constructor de la clase *X*. En la figura 4.5 se puede observar el estado de la pila al momento de pasar el control al constructor de la clase *X*.

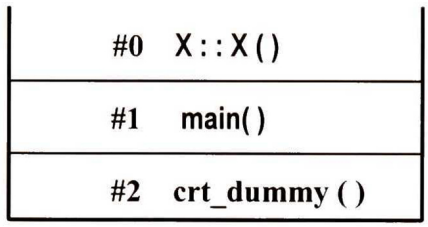


Figura 4.5 Estado de la pila al pasar el constructor *writer*.

Como la clase *X* es una clase derivada de la clase *process*, el núcleo pasa el control del proceso **A** al constructor de la clase *process y* se inserta en la pila el *stack frame* de la función constructora de la clase *writer* y los datos del constructor de *process* como se muestra en la figura 4.6.

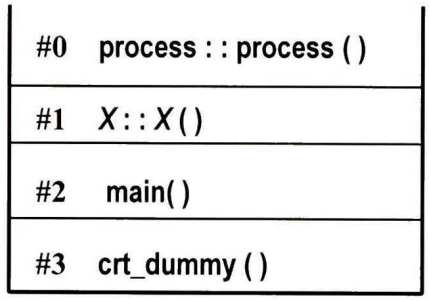


Figura 4.6. Estado de la pila al pasar el constructor *process*.

El núcleo ejecuta ahora todas las instrucciones que se encuentran en el constructor de la clase *process* incluyendo la función *fork()*, que crea un proceso hijo, al que denominamos **B**. Cuando el proceso **A** termina de ejecutar todas las instrucciones del constructor de la clase *process*, pasa a ejecutar ahora las instrucciones del constructor de la clase hija es decir de la clase *X*. El proceso **B** una vez creado, ejecutará las instrucciones que siguen de la llamada el sistema *fork()* que lo creo. Por lo tanto el proceso **A** y el proceso **B** ejecutan el constructor de la clase *X*.

Notemos que al momento de la creación del proceso **B** también se crea una pila igual a la del proceso **A**, por lo que de aquí en adelante se tienen dos procesos ejecutando las mismas instrucciones y por lo tanto existen dos pilas.

En este punto se saca de las dos pilas el *stack frame* de la función constructora de la clase *process*, dejando la pila como se muestra en la figura 4.7.

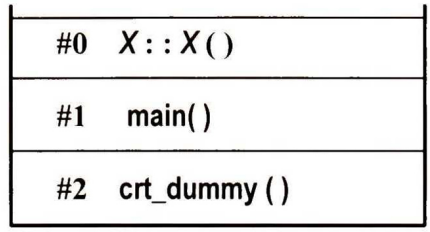


Figura 4.7. Estado de las dos pilas (del proceso hijo y el proceso padre) al sacar el *stack frame* de *process*.

Cuando los dos procesos terminan de ejecutar el constructor de la clase *X*, el control se regresa nuevamente a la función *main* para continuar con las instrucciones que siguen a la declaración del objeto "*objeto1*"(línea 33, figura 4.2). Si en este punto existiera otra declaración de un objeto de este tipo, el seguimiento sería el mismo.

Al pasar el control a la función *main()*, a las pilas de los dos procesos **A** y **B** se les quita el *stack frame* de la clase *writer* como se muestra en la siguiente figura:

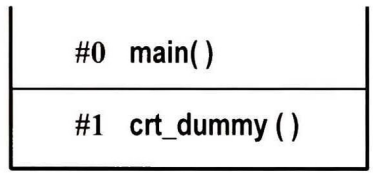


Figura 4.8. Estado de las dos pilas (del proceso hijo y el proceso padre) al sacar el *stack frame* de *writer*.

Este sería el flujo normal del programa, pero como se puede observar tiene los siguientes problemas:

1) Al inicio de la ejecución de este programa se tiene un solo proceso padre llamado **A**; cuando este proceso llega al constructor de la clase *process* crea otro proceso que llamamos **B** (Hijo de **A**), que es una copia de **A**. Por lo tanto, de aquí en adelante, los dos procesos ejecutan las funciones siguientes que son el constructor de la clase *X*() y la función *main*().

Esto es un problema, ya que lo que nosotros deseamos es que el proceso **A** sólo se encargue de la creación de los objetos-proceso, en este caso el *objeto-proceso* **B** y que continúe creando mas *objeto-proceso* si existen más declaraciones de objetos o termine su actividad inmediatamente después de esto, mientras que el *objeto-proceso* hijo **B** se encarga de ejecutar las instrucciones del constructor de la clase *X* que define su comportamiento. Una vez que el proceso **B** termina de ejecutar las instrucciones del constructor de la clase *X* también debe terminar su actividad, ya que como mencionamos anteriormente el comportamiento del *objeto-proceso* lo

define el constructor de la clase a la que pertenece. Es decir, si deseamos que el *objeto-proceso* se comporte como un programa que escribe en un archivo, ese comportamiento se programará en el constructor de la clase que lo define.

2) El otro problema que se detectó es que el *objeto-proceso* hijo **B** inicia su actividad inmediatamente después de que es creado; cuando debería iniciarla hasta que todos los *objetos-proceso* sean creados, esto con el fin de que trabajen concurrentemente.

Para ilustrar lo anterior, supongamos que tenemos un programa como el de la figura 4.9, que utiliza la clase *process* en una clase *writer* y se declaran tres *objetos-proceso* de esa clase.

Ejemplo:

```
main()
 {
        writer W1 ;
        writer W2 ;
        writer W3 ;
 }
```

Figura 4.9. Declaración de tres objetos-proceso del tipo writer

Al ejecutar este programa el núcleo crea primero al proceso principal que denominamos **A**. Al detectar la declaración del objeto *W1* se manda llamar al constructor de la clase *writer* e inmediatamente después al de la clase *process*, como ya se dijo; aquí se crea un proceso hijo que llamaremos también *W1* y que se ejecuta inmediatamente. Posteriormente, el proceso **A** llega a la declaración del objeto *W2* y enseguida se crea el proceso *W2* y se ejecuta, y por ultimo el objeto *W3* y el proceso *W3* se ejecuta. Como podemos ver, la ejecución de estos tres procesos es secuencial y no concurrente como se desea.

A continuación se describe como se solucionaron estos problemas y se muestra como se modifico el código de la figura 4.2, para obtener la clase *process* definitiva.

4.1.1 Solución a la creación de procesos.

El primer problema que se presentó con el programa es que el proceso principal **A** ejecuta todas las instrucciones del programa, cuando sólo debería encargarse de la creación de los *objetos-proceso* y de las instrucciones dentro de la función *main()*, si es que existen. Inmediatamente después de esto debe de terminar su actividad. Este problema se resolvió de la siguiente manera:

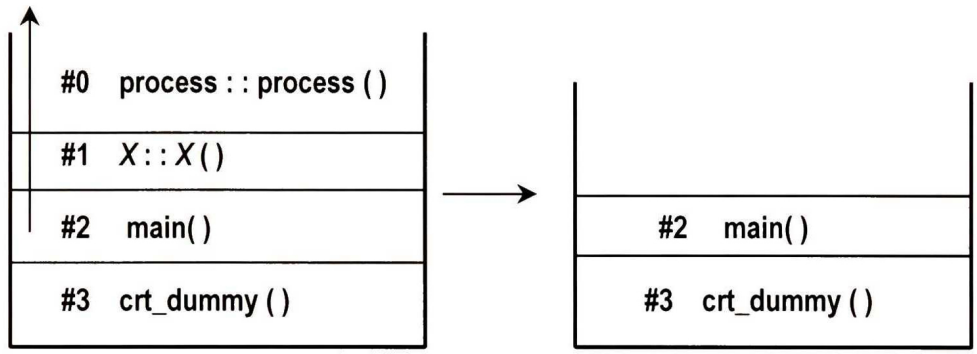
El flujo del programa es el mismo hasta la parte en que se llega al constructor de la clase *process*. Una vez que se crea el proceso con la llamada al sistema *fork()*, en el constructor de esta clase, se saca de la pila la dirección de regreso a la función *main()* y ésta se asigna al

apuntador conocido como *Base Pointer* (BP), lo cual provoca que el control del programa pase a ejecutar la siguiente declaración que aparezca en *main()*. Es importante decir que esta es la única forma y más eficiente, en que se puede mandar el control del programa a *main()*. Antes de hacerlo así, se pensó en otras formas, sin embargo ninguna de ellas funcionó correctamente.

En la figura 4.10 se muestra como se modifica la pila del proceso principal para obtener la dirección de regreso a la función main().

Se saca el *stack frame* #1 y obtenemos del *stack frame* #2 la dirección de regreso a la función *main()*.

Stack Pointer = Dirección de retorno a main()



Estado de la pila cuando el control Se encuentra en la clase *process*.

Estado de la pila después de sacar el *stack frame* de la clase *X*.

Figura 4.10. Estados de la pila antes y después de ejecutar las instrucciones del constructor *process*.

El programa que realiza el cambio del control de flujo normal a la forma en que se desea se muestra en la figura 4.11.

```
1    #include <stdio.h>
2    #include <iostream.h>
3    #include <signal.h>          // Para el tratamiento del signal SIGSEGV
4    #include <stdlib.h>
5    #include <sys/types.h>       // Para el fork
6    #include <asm/segment.h>     // Para el código en ensamblador
7    #include <sys/sem.h>         // Para el semáforo
8
9    #define SEMAFORO 0
10
```

```
11      struct sembuf cmm;              // Definición de la estructura
12      key_t ll=ftok("SEM",'X');
13      int sid = semget(ll, 1, IPC_CREAT | 0600);
14
15      class process
16        {
17              int x;
18              public:
19              process();
20        };
21
22      // Constructor de la clase process.
23  process::process()
24  [
25      int y;
26
27        if (getppid() != 1)
28          // Se crea un proceso...
29           y = fork();
30        else
31          exit(0);
32
33        switch(y)
34        {
35          //Si el proceso es el hijo.
36          case 0:
37              cmm.sem_num = SEMAFORO;
38              cmm.sem_op = -1;
39              cmm.sem_flg = 0;
40              semop(sid, &cmm, 1);
41              break;
42          //Si el proceso es el padre
43          default:
44            __asm__("
45                        movl %bp,%sp    # Mueve el apuntador de pila a la dirección
46                                        # donde esta el apuntador de base
47                        popl %bp        # Desapila el apuntador de base
48                        movl %bp,%sp    # Repite la operación
49                        popl %bp
50                        ret             # Regresa a main()
51                    ");
52          };};
```

Figura 4.12. Código de la clase *process*.

Como se puede ver en la figura anterior, se agregó un bloque de instrucciones en lenguaje ensamblador que obliga a que el control regrese a la función *main()* y no al constructor de la clase *X* (como lo haría normalmente), después de haber creado al *objeto-proceso* correspondiente. Todo lo anterior se hace con el fin de que el proceso principal sólo se encargue de crear los *objetos-proceso* y darles la señal para que empiecen su ejecución concurrente ejecutando la instrucción *cobegin*, que será explicada más adelante.

Este bloque de instrucciones en ensamblador realiza las siguientes acciones: primero mueve el apuntador de pila **SP** a la dirección donde se encuentra el apuntador de base **BP** del *stack frame* de la clase *process* como se muestra en la figura 4.13a. Luego extrae de la pila el apuntador de base **BP** con la instrucción *popl %bp*, lo cual provoca que el apuntador de base apunte ahora el **BP** del *stack frame* de la función anterior (en este caso el constructor de la clase *X*), como se ve en la figura 4.13b. Estos dos pasos se repiten una vez mas, provocando que el **BP** apunte ahora al **BP** del *stack frame* de la función *main()* y el **SP** apunte a la dirección de retorno a la función *main()* (figura 4.13c), por lo cual al ejecutar la instrucción *ret*, el control de ejecución pasa a la función *main()*.

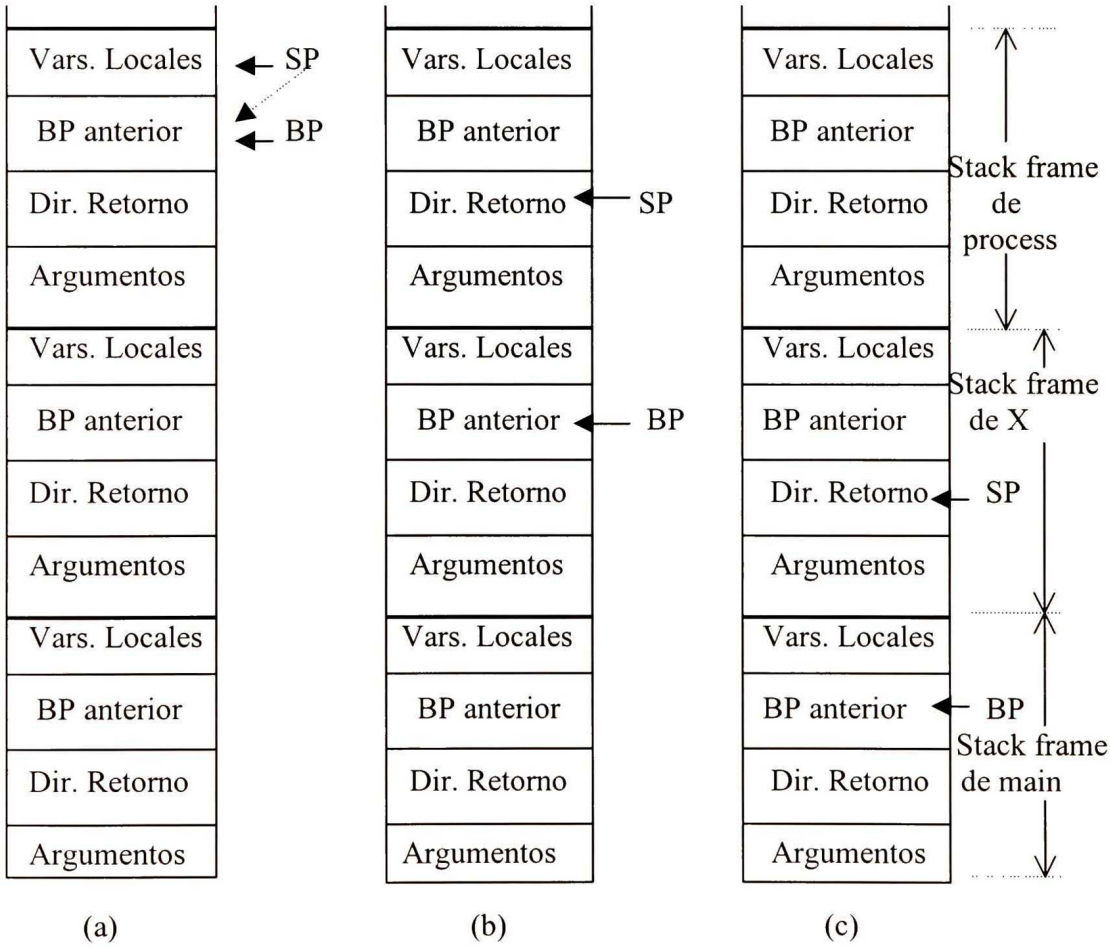


Figura 4.13. La pila en los tres puntos de la ejecución.

De esta forma el control de flujo del proceso principal se altera en la forma que muestra la figura 4.14.

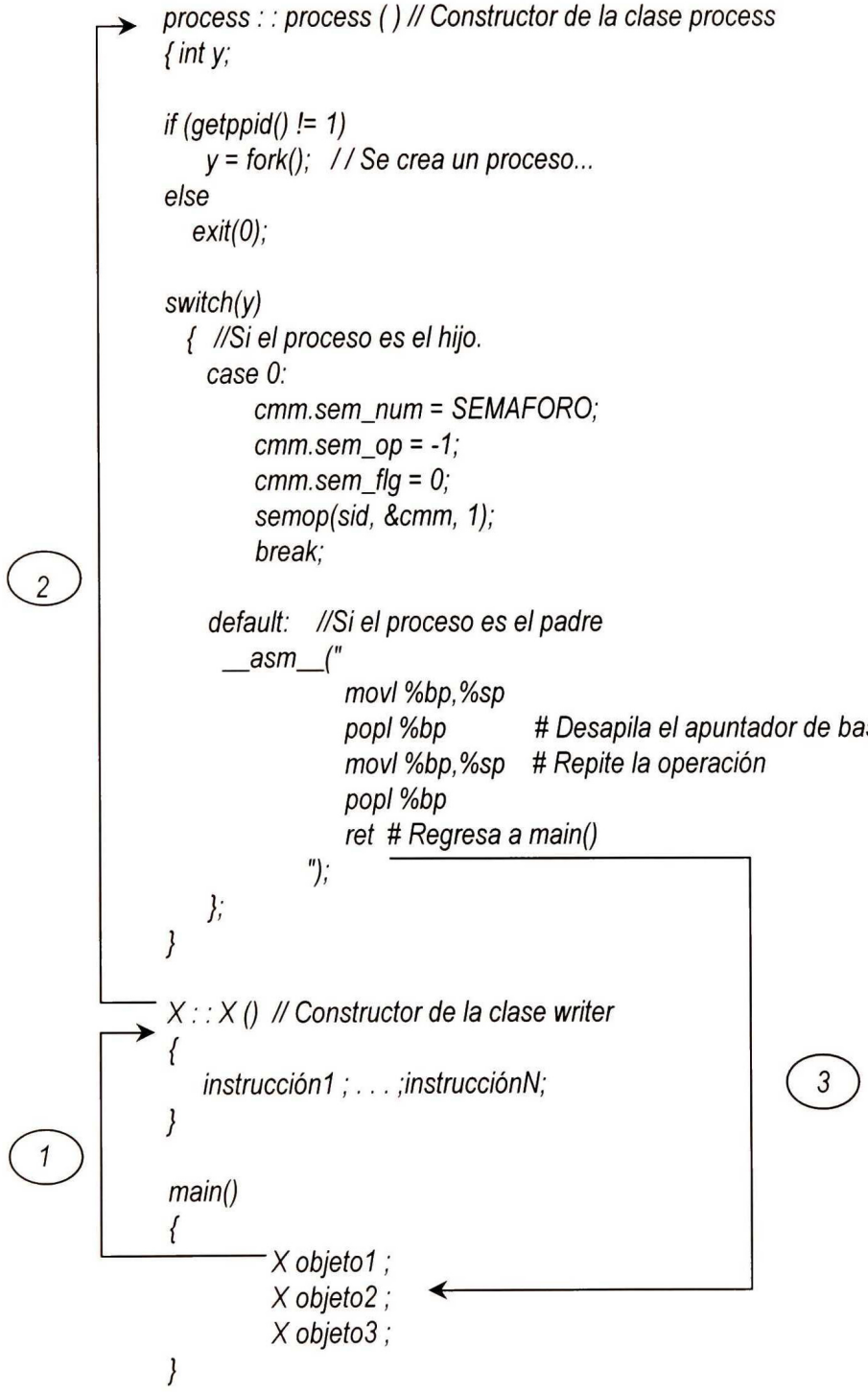


Figura 4.14. Cambio del control de flujo del proceso principal utilizando la clase *process*.

Cuando el proceso principal encuentra la declaración del objeto *objeto1*, el control de flujo pasa al constructor de la clase *X* e inmediatamente al constructor de la clase *process*. Ya dentro del constructor se crea el proceso hijo y se saca de la pila el *stack frame* del constructor de la clase *X* utilizando código en ensamblador. A continuación se obtiene la dirección de retorno a la función *main()* y la asigna al *stack pointer*, lo cual modifica el control de flujo normal y lo manda la ejecución a la siguiente instrucción de la función *main()* que en este caso es la declaración del *objeto2*. Enseguida el proceso principal sigue el mismo flujo de control anterior para el *objeto2* y luego para el *objeto3*.

Como se puede observar, el proceso principal ya no ejecuta el constructor de la clase *X*, sólo el de *process* y se limita a crear los procesos para así terminar su actividad.

Veamos ahora como se resolvió el problema de iniciar la ejecución todos los *objetos-proceso* de forma concurrente, ya que hasta aquí los procesos creados inician su actividad inmediatamente después de la función *fork()*.

4.1.2 Solución para el inicio de la ejecución concurrente.

Para solucionar este problema y que los procesos se ejecuten en forma concurrente, se debe suspender la ejecución de estos hasta que todos se hayan creado. Una vez que todos los procesos estén creados, se les permite iniciar su actividad en forma concurrente. El que gane el control del procesador, será el primero que ejecute las instrucciones siguientes que son las del constructor de la clase *X*, es decir la función que define el comportamiento del *objeto-proceso*.

Para lograr lo anterior se utilizaron semáforos, con lo cual se impide que los *objetos-proceso* inicien su actividad una vez que son creados y también permiten que inicien su actividad cuando todos hayan sido creados. El código de la figura 4.12 de las líneas 38 a la 41, muestran como se implantó esta solución dentro del constructor de la clase *process*: Primeramente se pregunta si el **PID** del proceso que esta ejecutando el código es distinto de 1; si es así, entonces el proceso es el padre o proceso principal, y como este es el único que debe crear procesos, ejecuta la función *fork()* que esta en el constructor, creando un proceso hijo que llamaremos **B**. En el caso de que el proceso que ejecute esto sea un hijo, su **PID** será 1, lo que provocará que este proceso termine.

El proceso padre después de ejecutar la instrucción *fork( )*, pasa al código en ensamblador donde se obtiene la dirección de la función *main* para continuar con el su flujo normal.

El proceso hijo **B** ejecuta el siguiente código, mismo que se ve en la figura 4.12:

```
cmm.sem_num = SEMAFORO;
cmm.sem_op = -1;
cmm.sem_flg = 0;
semop(sid, &cmm, 1);
break;
```

donde se utiliza un semáforo llamado "SEMAFORO" al que se le pone la bandera **cmm.sem_op = -1** para indicar que los procesos que la ejecuten queden bloqueados (función WAIT) hasta que se cambie esta bandera a 1.

Una vez que todos los *objetos-proceso* declarados en la función *main()* hayan sido creados y bloqueados y el siguiente paso es desbloquearlos. Para esto fue necesario agregar una instrucción que realizara esta acción, que desbloqueará el semáforo para permitir el inicio de actividad concurrente de los *objetos-proceso*. A esta instrucción la llamamos *COBEGIN*.

4.1.2.1 La instrucción COBEGIN

Esta instrucción es la que permite iniciar la actividad concurrente de los *objetos-proceso* creados en la clase *process*.

La implantación de esta instrucción, utilizando el coprocesador del lenguaje C, es como sigue:

```
#define cobegin cmm.sem_num = SEMAFORO;        \
                cmm.sem_op = 1;                \
                cmm.sem_flg = 0;               \
                semop(sid, &cmm, 1);           \
                semctl(sid,0,IPC_RMID,x);
```

Figura 4.15. Macro de la instrucción *cobegin*.

En este código se hace referencia al semáforo sobre el cual están bloqueados los *objetos-proceso* creados llamado *SEMAFORO*, poniendo la bandera *sem_op = 1* (signal), lo cual provoca que el núcleo seleccione alguno de estos procesos y lo ponga en actividad. Enseguida la instrucción *semctl(sid,0,IPC_RMID,x)* provoca que el semáforo sea borrado, lo cual provoca que los procesos se desbloqueen y a partir de este punto se deja la administración de los procesos al núcleo del sistema.

Lo único que el programador deberá incluir en su código de la función *main()* es la instrucción *cobegin* como se muestra en el siguiente ejemplo:

```
main()
{ writer W1 ; writer W2 ; writer W3 ;
// Aquí se abre el semáforo y los procesos que estaban
// bloqueados comienzan a ejecutar la instrucción siguiente a donde se quedaron
// que es el constructor de la clase writer que define su comportamiento.
  cobegin ;
}
```

Figura 4.16. Utilización de la instrucción *cobegin*.

Después de ejecutada la sentencia *cobegin* la planificación de los procesos es dejada al núcleo.

La actividad del proceso principal termina cuando ejecuta la ultima instrucción de la función *main()* y la de los *objetos-proceso* cuando ejecutan la instrucción *if-else* que se encuentra en el constructor de la clase *process.*

La implantación del código para la creación de procesos y para la sentencia *cobegin* se encuentra en el archivo de cabecera llamado *class.h.*

En la figura 4.17 se muestra un ejemplo completo que muestra como se utiliza la clase *process* y la instrucción *cobegin.*

```
1   //Este programa crea tres objetos-proceso que escriben un mensaje en la pantalla
2
3   #include <stdio.h>
4
5   #include "class.h"
6
7   class writer: process
8   {
9   public:
10  writer();
11  };
12
13  writer::writer()
14  {
15    printf("ESCRITOR %d\n", getpid());
16  }
17
```

```
18      main()
19      {
20      writer W1;
21      writer W2;
22      writer W3;
23
24      cobegin;
25      }
```

Figura 4.17. Ejemplo del uso de la clase *process* y la instrucción *cobegin*.

En el siguiente capítulo se explica como se realizó la implantación de la instrucción *select-when*, así como los problemas encontrados durante dicha implantación.

5. La Instrucción SELECT-WHEN y la Comunicación Entre Procesos

5.1 Implantación de Select-When

Para la implantación de la instrucción *select-when*, nos basamos en la implantación hecha para el lenguaje ADA, de la misma instrucción, la cual es no determinista en su ejecución. En el caso del lenguaje C++ Distribuido, esta instrucción no es completamente no determinista, ya que para incluir el no-determinismo, es necesario realizar cierta cantidad de instrucciones para hacer la prueba de los guardias en forma no determinista, lo cual implica cierto tiempo de procesamiento adicional que quisimos evitar. Estas operaciones adicionales son las siguientes: Primero se debe contar la cantidad de guardias que tiene la instrucción (esta cantidad es variable dependiendo del programa), después se debe generar un número aleatorio entre uno y el número de guardias que tenga la instrucción *select-when*, para seleccionar uno de los guardias y por último probar el guardia seleccionado y, si este es verdadero, entonces ejecutar sus instrucciones y si es falso, repetir los pasos anteriores sin incluir al guardia que ya se probó. El tiempo de procesamiento provocado por estas instrucciones vuelve más lenta la ejecución de la instrucción *select-when*. Por esta razón, en la instrucción *select-when* que se presenta aquí, la prueba de los guardias se realiza de forma secuencial. Decimos que esta instrucción es de alguna forma no determinista por que la prueba de los guardias en forma secuencial es un subconjunto de la prueba no determinista.

En la implantación de esta instrucción se utilizan los semáforos del sistema UNIX como base, pero el usuario no tendrá el problema de la creación y control de ellos, únicamente debe incluir la instrucción *select-when* en la forma como se muestra en el capítulo 3. Con el uso de esta instrucción, se abstrae al usuario del uso y control de los semáforos.

El primer paso, en la implantación, es crear un semáforo que llamamos MUTEX (por exclusión mutua) y que utilizamos para controlar el acceso, en exclusión mutua, de los procesos cuando se van a probar los guardias y cuando se ejecutan las instrucciones que le siguen a los guardias. En la figura 5.1 se presenta el código de creación e inicialización del semáforo MUTEX.

La instrucción *select-when* se programó con macros utilizando el preprocesador del lenguaje C, ya que esta es la forma en que se pueden escribir nuevas instrucciones como la presentada aquí. Las macros utilizadas para esta instrucción son: *Wait* y *Signal*, utilizadas para bloquear y desbloquear procesos con el semáforo; *Select*, *EndSelect*, *When* y *ElseWhen* que son las palabras reservadas que conforman la instrucción *select-when*. En la figura 5.2 se presenta el código de éstas macros.

Cuando un proceso ejecuta la instrucción *select-when*, primero se ejecuta la palabra *select* que pone en uno el valor del semáforo MUTEX (Macro *Select* en la Figura 5.2, línea 14), provocando que el semáforo bloquee a los siguientes procesos que tratan de ejecutar la

instrucción *select-when* hasta que el proceso que esta ejecutando actualmente la instrucción *select-when* termina de ejecutarla. Una vez que el proceso asegura su acceso exclusivo a la instrucción, utilizando el semáforo, éste ejecuta la instrucción *if(0)* (Línea 17, Figura 5.2) con lo cual el proceso comienza a probar los guardias de las palabras *When(B)* que le siguien.

```
1    /* Incluimos la librería para los semáforos */
2    #include <sys/sem.h>
3
4    /*Definimos el numero de nuestro semáforo */
5    #define MUTEX 0
6
7    struct sembuf semaforo;
8    union semun set_val;
9    key_t llave= ftok ("MUTEX",'K');
10   int semid= semget(llave, 1, IPC_CREAT | 0600);
11   int pid;
```

Figura 5.1 Inicialización del semáforo *MUTEX*.

```
1    // Macro Wait para el semáforo
2    #define Wait    semaforo.sem_num = MUTEX;        \
3                    semaforo.sem_op = -1;            \
4                    semaforo.sem_flg = 0;            \
5                    semop(semid,&semaforo,1);        \
6
7    // Macro Signal para el semáforo
8    #define Signal  semaforo.sem_num = MUTEX;        \
9                    semaforo.sem_op = 1;             \
10                   semaforo.sem_flg = 0;            \
11                   semop(semid,&semaforo,1);        \
12
13   // Macro Select
14   #define Select  set_val.val=1;                                \
15                   semctl(semid, MUTEX, SETVAL, set_val);       \
16                   Wait(MUTEX);                                 \
17                   if(0) ;                                      \
18   //  Macros EndSelect, When y ElseWhen
19   #define EndSelect Signal(MUTEX);
20   #define When(B) else if(B)
21
22   #define ElseWhen else
```

Figura 5.2 Macros de la instrucción Select-When

La macro *WAIT* (Figura 5.2, líneas 2-5) asigna el número –1 a *semaforo.sem_op* lo cual provoca que el siguiente proceso que trata de ejecutar esta macro se bloquee hasta que *semaforo.sem_op* sea igual a 1.

Cuando un proceso termina de ejecutar la instrucción *select-when*, ejecuta la macro *EndSelect* la cual a su vez llama a la macro *SIGNAL* que pone *semaforo.sem_op = 1* provocando que un proceso de los que están bloqueados con el semáforo *MUTEX*, se desbloqueé y pueda ejecutar la instrucción *select-when*.

En la figura 5.3 se muestra como se substituyen las palabras reservadas *Select, When(B), ElseWhen* y *EndSelect* por sus respectivas macros. De esta forma se puede observar más claramente como funciona la implantación de esta instrucción.

```
Select                          set_val.val=1;
                                semctl(semid, MUTEX, SETVAL, set_val);
                                Wait(MUTEX);
                                if(0) ;
When(B1)                        else if(B1)
        Instrucciones;                  instrucciones;
When(B2)                        else if(B2)
        Instrucciones;                  instrucciones;
    :                                   :
When(Bn)                        else if(Bn)
        Instrucciones;                  instrucciones;
ElseWhen(Bn)                    else
        Instrucciones;                  instrucciones;
EndSelect                       Signal(MUTEX);
```

Figura 5.3. Instrucción *Select-When* con la substitución de sus respectivas macros.

La palabra *ElseWhen* es opcional y funciona de la misma forma que la instrucción *else* del lenguaje C.

La implementación de esta instrucción se encuentra en el archivo *select.h*, que a su vez está incluido en el archivo *cppd.h*.

A continuación se explica como se implantó la comunicación entre procesos para el lenguaje ***C++ Distribuido***.

5.2 Comunicación entre procesos

En esta parte del trabajo se explica la implantación de la clase *channel* y de sus dos funciones básicas para la transmisión y recepción de mensajes. La implantación de ésta clase se realizó tomando como base los *sockets* y las primitivas de comunicación entre procesos implantadas en varios lenguajes de programación distribuidos y en sistemas operativos distribuidos. Las primitivas seleccionadas para ser implantadas en este lenguaje, son las funciones básicas de transmisión y recepción de mensajes utilizando un puerto virtual. Estas primitivas son generalmente conocidas como *send()* y *receive()*, respectivamente. Su forma básica es la siguiente: En la función *send()*, un proceso envía un mensaje a un canal *ch* ejecutando:

```
send (ch, mensaje) ;
```

Dado que la cola de mensajes es conceptualmente no-acotada, la ejecución de *send()* no utiliza tiempo para el envío del mensaje, por lo que *send()* es una primitiva no bloqueante.

El efecto de ejecutar *receive()* es esperar hasta que exista al menos un mensaje en la cola del canal. Entonces, el mensaje es tomado de la cola. Así, en contraste a *send()*, *receive()* es una primitiva bloqueante, esto es que sí causa perdida de tiempo al ejecutarse.

En algunos sistemas operativos distribuidos, como Mach [9,29], se define el canal de comunicación, usado para pasar datos, como un *PUERTO*. Un puerto es implantado como una cola de mensajes que maneja el *Núcleo*. Los mensajes contienen un tipo de colección de objetos de datos. El programador define que datos son pasados en un mensaje y el tipo de dato.

5.2.1 Las funciones cpp_send() y cpp_receive()

Teniendo como base lo anterior, se pensó en crear dos funciones básicas: *cpp_send()* y *cpp_receive()*, que permitieran enviar y recibir mensajes, respectivamente; a través de un puerto virtual.

Esta funciones fueron incorporadas dentro de una nueva clase para comunicación entre procesos, llamada *channel*. A continuación se muestra el código de dicha clase:

```
class channel
{
 public:
        channel() {};
   int cpp_send(char*,int, void*);
   int cpp_receive(char*,int, void*);
};
```

Nota: Para ver con detalle el contenido de dichas funciones, referirse al apéndice B.

La implantación de estas dos funciones se basa en el modelo cliente-servidor, en donde el cliente es el proceso que envía el mensaje y el servidor es el proceso que recibe el mensaje.

La función *cpp_send()* (ver código en Apéndice B) internamente trabaja de la siguiente forma:

1. Abre un canal de comunicaciones y se conecta a la dirección de red de la máquina donde se encuentra el proceso receptor. Naturalmente esta dirección debe ser conocida por el proceso emisor y debe corresponder al esquema de generación de direcciones de la familia internet.
2. Enviar un mensaje al receptor.
3. Cerrar el canal de comunicaciones y terminar la ejecución.

La función *cpp_receive ()* (ver código en Apéndice B), por su parte, lleva acabo las siguientes acciones:

1. Abrir el canal de comunicaciones e informar a la red tanto a la dirección que responderá como a su disposición para aceptar mensajes.
2. Esperar a que un proceso emisor envíe un mensaje a la dirección que él tiene declarada.
3. Cerrar el canal de comunicaciones y terminar la ejecución.

Cabe mencionar que una vez que un proceso ejecuta esta función (proceso receptor), se bloquea y si ningún mensaje es recibido durante cierto tiempo, la función devuelve el control inmediatamente indicando que se ha producido un error de conexión.

En la figura 5.4 se puede ver claramente cual es la secuencia de llamadas al sistema que realizan las funciones *cpp_receive* y *cpp_send* para establecer una comunicación.

Los pasos mostrados en la figura 5.4 están ocultos para el programador que utiliza éstas funciones, lo cual abstrae al programador de la creación y control de los *sockets*. El programador únicamente debe usar las funciones de la forma mostrada en el capitulo 3.

Como se puede ver en la figura 5.4 la función *cpp_receive()* se debe ejecutar antes que la función *cpp_send()*, para que el proceso emisor pueda establecer la conexión con el proceso receptor antes de enviar el mensaje. Es decir, el proceso receptor debe estar en espera de la petición de conexión en la función *accept()* antes de que el proceso emisor haga esa petición con la función *connect()*. Si sucede lo contrario, la función *cpp_send()*, inmediatamente muestra un mensaje indicando un error en la conexión. Esto sucede por no encontrar un proceso que este esperando por esa conexión.

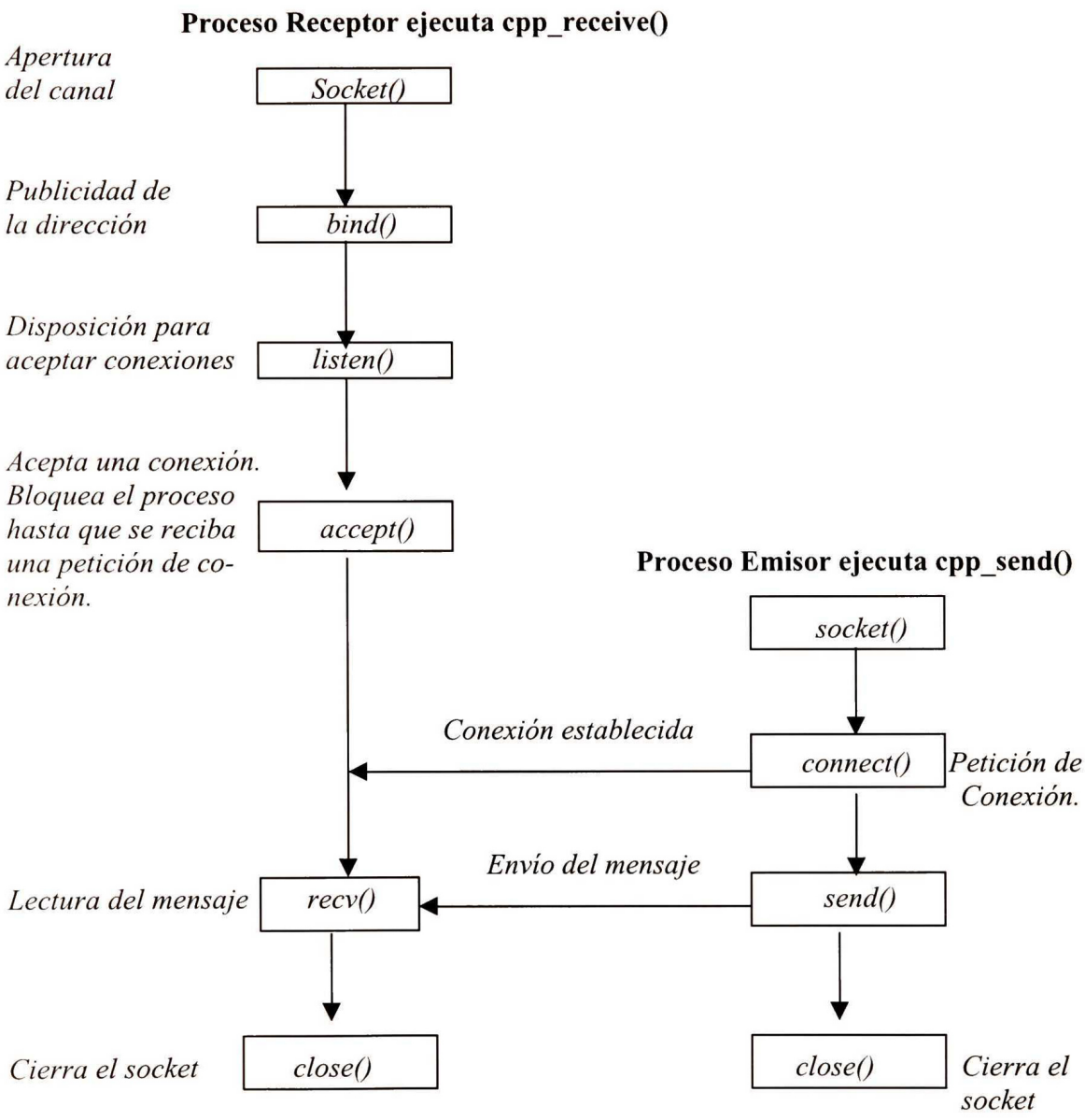


Figura 5.4. Secuencia de instrucciones utilizadas por las funciones *cpp_receive()* y *cpp_send()*.

En este trabajo no se explica con detalle cada una de las llamadas al sistema mostradas en el diagrama anterior. Para una explicación más amplia de ellas, se puede consultar bibliografía de comunicación en UNIX [19,20].

El archivo *msg.h* contiene la definición y especificación de la clase *channel* con sus dos funciones para el paso de mensajes: *cpp_receive y cpp_send*. Cada una de las funciones crea su propio *socket* y sus propias estructuras de datos para su manejo.

Hasta aquí se han presentado las herramientas que forman al lenguaje C++ Distribuido. Se a explicado su utilización y como se realizó la implantación de ellas. En el siguiente capítulo se presenta una aplicación que muestra como las herramientas añadidas al lenguaje C++ facilitan mucho la programación.

6. Una Aplicación

Para mostrar como se utilizan las extensiones que tiene el ***C++ Distribuido***, se presenta a continuación un programa que muestra una solución al problema de los filósofos comelones. Este problema de sincronización de procesos fue propuesto y resuelto por Dijkstra en 1965.

En el problema de los filósofos comelones se presentan a cinco filósofos que alternan sus actividades de pensar y comer. Cuando un filósofo tiene hambre, se dirige a una mesa redonda y toma los dos cubiertos que se encuentran al lado de su plato y comienza a comer. Sin embargo, solamente hay cinco cubiertos en la mesa, así que un filósofo sólo puede comer cuando ninguno de sus vecinos inmediatos lo esta haciendo. Cuando un filósofo termina de comer, pone los dos cubiertos de nuevo en la mesa y la abandona [31].

Muchos autores han propuesto una variedad de soluciones a este problema para procesos que se ejecutan en una sola computadora. En este trabajo se presenta una solución más a este problema pero extendiéndolo a arquitecturas distribuidas, propuesta por Chandy y Misra [26]. En esta solución los procesos filósofos se ejecutan en distintas máquinas conectadas a través de una red.

La forma en que trabaja este programa es muy sencilla: Cuando un proceso tiene hambre, primeramente revisa si tiene los tenedores derecho e izquierdo, para poder comer; si los tiene entonces come y después de comer, libera los tenedores y se pone a pensar. En caso de que el proceso filósofo no tenga los tenedores, envía un mensaje a su respectivo proceso vecino para solicitarle el tenedor (función hambriento, figura 6.2, líneas 9-34), si su vecino lo esta ocupando entonces el proceso solicitante debe esperar un momento y después volverlo a solicitar hasta que lo pueda obtener y entonces comer. Estos pasos se repiten varias veces.

Los tenedores están representados por variables enteras que indican si se tiene o no el tenedor. Si el valor del número es cero, entonces el proceso no tiene el tenedor y si es uno, entonces el proceso tiene el correspondiente tenedor.

Para esta aplicación se crearon cinco programas y en cada uno de ellos se crea un filósofo. Cada *objeto-proceso filo(i)* definido en cada uno de estos programas, representa a un filósofo con su identificador. Se hubiera podido hacer un solo programa donde se crearan los cinco filósofos, por ejemplo:

```
main()
{       filosofo filo(1); filosofo filo(2); filosofo filo(3);
        filosofo filo(4); filosofo filo(5);
        cobegin;
}
```

y haciendo algunas modificaciones en las funciones mostradas en la figura 6.2, sin embargo, con el fin de mostrar con más claridad el comportamiento de cada filósofo, se decidió hacerlo por separado.

En el programa de la figura 6.2 se utiliza la clase *process* para crear los filósofos y la clase *channel* para comunicarlos. Todo programa que utilice la clase *process*, las instrucciones *select-when, cobegin* y/o la clase *channel* del ***C++ Distribuido***, debe incluir el archivo cabecera: *cppd.h* ya que en este archivo están definidos los archivos: *select.h, process.h* y *msg.h* que a su vez contiene las extensiones realizadas al C++.

El comportamiento de cada uno los filósofos lo da la función constructora de la clase filósofo. Cada filósofo realiza tres principales funciones: *Hambriento(), Comiendo()* y *Pensando()* (figura 6.2).

La comunicación entre los filósofos se hace a través de las funciones *cpp_send*() y *cpp recieve()* de la clase *channel*. Cada filósofo manda mensajes a su vecino utilizando el mismo número de canal, como se muestra en la figura 6.1.

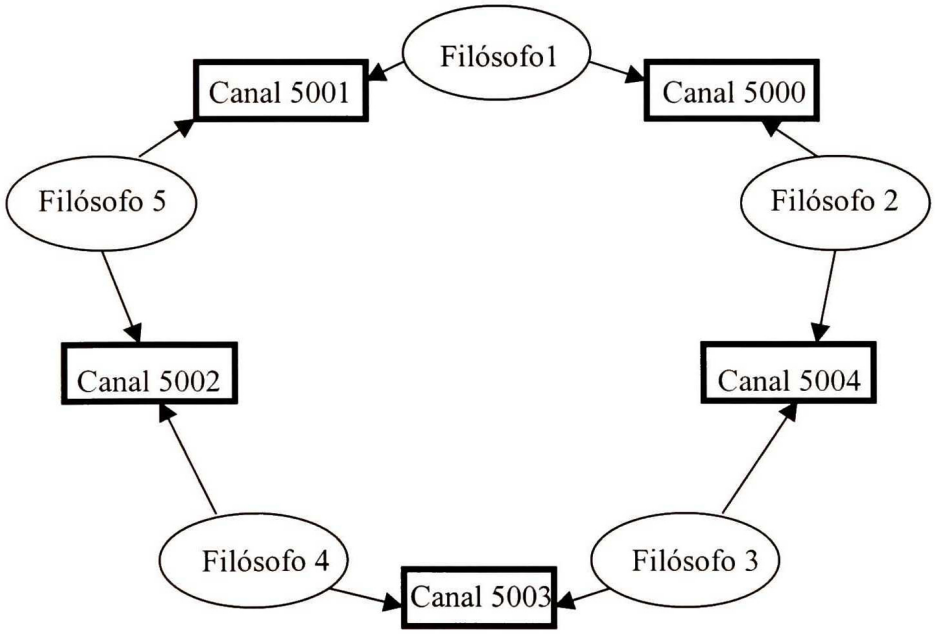


Figura 6.1. Filósofos comelones para arquitecturas distribuidas.

En este reporte sólo se presenta el código de uno de los filósofos. El código de los demás filósofos es similar al presentado aquí, lo único que cambia es su identificador y el canal al que deben mandar y del que deben recibir los mensajes.

A continuación se presenta el código completo de uno de los filósofos:

```
1   #include "cppd.h"    // Programa que contiene las extensiones
2
3   int tenedor_izq;     // Variable que representa al tenedor izquierdo
4   int tenedor_der;     // Variable que representa al tenedor derecho
5   char t_d[1],t_i[1];  // Arreglos de caracteres para enviar mensaje de posesión de tenedor.
6
7   // Función: hambriento
8
9   int hambriento(int id)
10  {
11   channel ch;
12  int resp;
13
14          while(1)
15          { //revisa que tenga los dos tenedores
16          if(tenedor_izq==1 && tenedor_der==1)
17              {
18                  printf("Filosofo %d, Tengo los dos tenedores y puedo comer", id);
19                  break;
20              }
21          else
22          {
23            if(tenedor_izq==0)
24              {
25                  resp=ch.cpp_receive("148.247.2.205",5002,t_i);
26                  if (resp==1)
27                          tenedor_izq=1;
28              }
29             if(tenedor_der==0)
30              {
31                  resp=ch.cpp_receive("148.247.2.205",5001,t_d);
32                  if (resp==1)
33                          tenedor_der=1;
34              }
35          }
36     }
37  return 1;
38  }
```

```
39
40      // Función: pensando
41
42      int pensando(int id)
43      {
44        channel ch;
45        int tiempo; respuesta
46        int contador;
47
46        tiempo=rand()/45;
48        contador=0;
49
50        printf("Filosofo %d: PENSANDO... ",id);
51
52        while (contador < tiempo)
53            {
54               contador++;
55            }
56
57        if(tenedor_izq)
58          {
59              t_i[0]='1';
60              respuesta=ch.cpp_send("148.247.2.205",5002,t_i);
61
62              if (respuesta==1)
63                  tenedor_izq= 0;
64          }
65        if(tenedor_der)
66          {
67              t_d[0]='1';
68              respuesta=ch.cpp_send("148.247.2.205",5001,t_d);
69              if (respuesta==1)
70                   tenedor_der = 0;
71          }
72      }
73
```

```
74      // Función: comiendo
75      int comiendo(int id)
76      {
77      int tiempo_comida; respuesta
78      int contador;
79
80      tiempo_comida=rand()/45;
81      contador=0;
82
83      while(contador < tiempo_comida)
84           {
85                contador++;
86           }
87
88        printf("Filosofo %d : TERMINE DE COMER... \n",id);
89      };
90
91       class filosofo: process{
92       public:
93      filosofo(int);
94      };
95
```

```
96      filosofo::filosofo(int id)
97      {
98        int i;
99       // Inicialmente, éste filósofo tiene los dos tenedores
100       tenedor_izq=1;
101       tenedor_der=1;
102
103       for(i=0; i<50; i++)
104         {
105               printf("\nFilosofo %d PENSANDO:\n",id);
106               pensando(id);
107               printf("\nFilosofo %d HAMBRIENTO:\n",id);
108               hambriento(id);
109               printf("\nFilosofo %d COMIENDO:\n",id);
110               comiendo(id);
111         }
112     };
113
114     main()
115     {
116     filosofo filo(1);
117     cobegin;
118     }
```

Figura 6.2 Código de un filósofo del problema de los Filósofos Comelones Distribuidos.

Como se puede observar en la figura 6.2, la utilización de la clase *channel* con sus respectivas funciones facilita la programación, ya que abstrae al programador de la utilización de los sockets para la comunicación de procesos, además de disminuir el código considerablemente. Por otro lado, la utilización de la clase *process*, permite que los programas sean más estructurados ya que hace uso de clases y objetos. La instrucción *select-when* no se utilizó, ya que aquí no se utilizan variables compartidas, pero su utilidad ya fue mostrada en los capítulos tres y cinco.

Los ejemplos presentados en los capítulos anteriores y en este, muestran la sencillez y la claridad del diseño y de la implementación aprovechando las características del lenguaje C++ Distribuido: La creación dinámica de *objetos-proceso*, la comunicación bidireccional, las regiones protegidas y el no determinismo.

Conclusiones

El lenguaje ***C++ Distribuido*** descrito en este trabajo difiere significativamente de otros similares como, por ejemplo, el ***C Concurrente*** de Gehani y Roome[1], basado en el modelo de procesos secuenciales comunicantes de Hoare. El ***C++ Distribuido*** es un lenguaje compatible con el C++ en el sentido que incorpora construcciones adicionales e independientes. Esto hace, sin duda, al lenguaje más poderoso pero mucho más complejo a la vez. La idea que se ha seguido en este trabajo consiste en extender al C++ incorporando la construcción *select-when*, la clase *process*, la instrucción *cobegin* y funciones para comunicación entre procesos, dando lugar a un lenguaje más simple para la programación de sistemas distribuidos.

La idea de considerar al proceso como objeto activo conlleva a una metodología eficaz para el análisis y diseño de sistemas complejos. Un sistema formado por objetos que interactuan entre si se puede modelar mediante una colección de procesos que se comunican entre sí.

Los ejemplos presentados en este trabajo muestran la sencillez y claridad del diseño y de la implementación aprovechando las características del lenguaje: la creación y destrucción de procesos, la comunicación síncrona bidireccional, las regiones protegidas y el no-determinismo. El ***C++ Distribuido*** ofrece una solución sencilla y practica para desarrollar aplicaciones. Puesto que no requiere de ningún preprocesamiento externo, no se pierde la capacidad interactiva del ambiente integrado, por ejemplo, un programa que usa la construcción *select-when* se compila y depura como si se tratara de cualquier otra construcción del lenguaje

En los ejemplos presentados en este trabajo se vio como el tiempo de programación disminuyó considerablemente, comparado con el tiempo empleado en desarrollar los mismos programas utilizando únicamente el C++ y no las extensiones con las que cuenta al ***C++ Distribuido***. En cuanto a la diferencia de tiempos en la ejecución, es prácticamente indistinguible.

Este trabajo puede extenderse en varias direcciones:

1. Diseñando clases especiales para incluir objetos remotos y persistentes.
2. Introduciendo nuevas construcciones (manejo de interrupciones, temporización o incertidumbre).
3. Incluyendo un manejo más real del no-determinismo en la instrucción *select-when*.
4. Incluyendo una clase similar a *process* pero usando *threads*, para mejorar la eficiencia de la implantación.
5. Incluyendo ambientes de programación (programación visual con redes de autómatas comunicantes o redes de Petri),
8. Incluyendo la implantación de un servidor de nombres para la comunicación de objetos.
9. Diseñando un mecanismo para la llamada remota a métodos de una clase.

Estas direcciones son temas importantes de investigación actual y el planteamiento de la programación distribuida orientada a objetos puede usarse como marco de referencia para el desarrollo de sistemas simples y eficientes.

Apéndice A. Los Sockets

A.1 Que es un Socket

Un *socket* es un punto de comunicación por el cual un proceso puede emitir o recibir información. En el interior de un proceso, un socket se identificará con un descriptor de la misma naturaleza que los que identifican los archivos (es decir, en el mismo conjunto. Esto significa que todo nuevo proceso (creado por *fork( )*) hereda los descriptores de sockets de su padre.

Si tomamos como base el modelo OSI, la comunicación mediante sockets es una interfaz con la capa de transporte (nivel cuatro) de la jerarquía de OSI.

La creación de un socket se realiza por la primitiva *socket* cuyo valor de vuelta es un descriptor sobre el cual es posible realizar operaciones de lectura y escritura. Un *socket* permite la comunicación en los dos sentidos, es decir, es bidireccional.

A cada una de las características de un socket le corresponde una constante simbólica predefinida en el archivo *<sys/socket.h>*.

A.2 El dominio de un Socket.

El dominio de un *socket* especifica el formato de las direcciones que se podrán dar al *socket* y los diferentes protocolos soportados por las comunicaciones vía los *sockets* de este dominio.

Muchas llamadas al sistema 4.3BSD de la capa de transporte necesitan un apuntador a una estructura de dirección de *socket* para trabajar. Esta estructura tiene la siguiente forma:

```
struct sockaddr {
   u_short     sa_family ;      /*familia de dirección*/
   char        sa_data [14];    /*14 octetos de dirección (máximo)*/
};
```

Para una aplicación particular, esta estructura se deberá reemplazar por la estructura correspondiente al dominio de comunicación utilizado.

a) Para el dominio UNIX (*AF_UNIX*), los *sockets* son locales al sistema donde han sido definidos. Permite la comunicación interna de procesos. Su designación se realiza por medio de una referencia UNIX. La estructura de una dirección en este dominio esta predefinida en *<sys/un.h>*.

```
struct sockaddr_un
```

```
{
        short       sun_family;     /*dominio UNIX; AF_UNIX*/
        char        sun_data[108];  /*referencia de dirección*/
};
```

b) Para el dominio Internet (*AF_INET*), las direcciones de los sockets tienen la estructura *sockaddr_in.*

```
struct sockaddr _in
    {
       short        sin_family;     /*la familia de la dirección AF_INET*/
       u_short      sin_port;       /*el número de puerto de 16 bits*/
       struct in_addr sin_addr;     /*la dirección Internet de 32 bits*/
       char         sin_zero[8];    /*un campo de 8 ceros*/
    };
```

c) Pueden existir otro cierto número de dominios, como son:

```
AF_NS           /*Protocolos XEROX NS*/
AF_CCITT /*Protocolos  CCITT, protocolos X.25, etc*/
AF_SNA          /*IBM SNA */
AF_DECnet       /*DECnet*/
AF_APPLETALK    /*Apple Talk*/
```

A.3 Las propiedades de la comunicación

El tipo de un *socket* define un tipo de las propiedades de las comunicaciones en las cuales está implicado.

a) La fiabilidad de la transmisión.- Ningún dato se pierde.

b)La conservación del orden de los datos.- Los datos llegan en el orden en el que han sido emitidos.

c) La no-duplicación de los datos. Solo llegan al destino un ejemplar de cada dato emitido

d) La comunicación en modo conectado.- Se establece una conexión entre dos puntos antes del principio de la comunicación . A partir de entonces, una emisión desde un extremo está implícitamente destinada al otro extremo conectado.

e) La conservación de los limites de los mensajes.- Los limites de los mensajes emitidos se pueden encontrar en el destino

f) El envío de mensajes "urgentes".- Corresponden a la posibilidad de enviar datos fuera del flujo normal y por consecuencia accesibles inmediatamente (se habla de datos fuera de flujo u out of band).

A.4 Tipos de sockets disponibles

Para definir el tipo de socket que se va a utilizar es necesarios conocer el tipo de conexión que se requiere. El tipo de conexión indica el tipo de circuito que se va a establecer entre los dos procesos que se están comunicando. El circuito puede ser virtual (orientado a conexión o en modo conectado) o datagrama (no orientado a conexión o modo no conectado). Para establecer un circuito orientado a conexión, se realiza una búsqueda de enlaces libres que unan las dos computadoras que se van a enlazar. Una vez establecida la conexión, se puede proceder al envío de los datos, ya que la conexión es permanente. Por el contrario, los datagramas no trabajan con conexiones permanentes. La transmisión por datagramas es al nivel de paquetes, donde cada paquete puede seguir una ruta distinta y no se garantiza una recepción secuencial de la información.

La comunicación en modo conectado es el utilizado por las aplicaciones estándar de la familia Internet tales como telnet, ftp o aplicaciones UNIX como rlogin. Aporta la fiabilidad de los intercambios de información con el precio de un incremento de su volumen.

Los tipos de sockets disponibles son:

a)El tipo SOCK_DGRAM.- Corresponde a los *sockets* destinados a la comunicación en modo no conectado para el envío de datagramas de tamaño limitado. Las comunicaciones correspondientes tienen la propiedad **e** del punto anterior. En el dominio Internet, el protocolo subyace es el producto UDP.

b) El tipo SOCK_STREAM.- Los *sockets* de este tipo permiten comunicaciones fiables en modo conectado (propiedades a, b, c y d) y eventualmente autorizan (según el protocolo aplicado) los mensajes fuera de banda (propiedad f). El protocolo subyace en el dominio Internet es TCP.

c)El tipo SOCK_RAW.- Permite el acceso a los protocolos de más bajo nivel (ejemplo, el protocolo IP en el dominio Internet). Su uso está reservado al superusuario. Permiten implantar nuevos protocolos.

d) El tipo SOCK_SEQPACKET.- Corresponde a las comunicaciones que poseen las propiedades a, b, c, d y e. Estas comunicaciones se encuentran en el dominio XEROX NS.

Los dos tipos de *sockets* más utilizados, son los dos primeros y el utilizado en este trabajo fue el de tipo SOCK_STREAM, ya que es el mas seguro y utilizado por la aplicaciones de UNIX, como ya se menciono anteriormente.

Apéndice B. Código de la clase Channel

En este apéndice se muestra el archivo *msg.h*, que contiene las funciones *cpp_send()* y *cpp_receive()*, respectivamente:

```
/*********************************************************************/
/* Programa : msg.h                                                  */
/*                                                                   */
/* Este programa contiene las funciones para                         */
/* paso de mensajes entre procesos del C++                           */
/* Distribuido.                                                      */
/*                                                                   */
/*********************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include  <varargs.h>

/*****************************************************************************/
// Clase channel.
// Tiene tres funciones: El constructor y las dos funciones para
// emisión y recepción de mensajes, respectivamente

class channel{
  public:
          channel() {};
    int cpp_send(char*,int, void*);
    int cpp_receive(char*,int, void*);
};

/*****************************************************************************/
// Función:  cpp_send()
// Esta función es usada por un cliente orientado a conexión
// cpp_send() usa el protocolo TCP

int channel::cpp_send(char *dir_serv, int puerto, void *point_datos)
{
    int             sockfd;
    struct sockaddr_in  serv_addr;
```

```
// LLena la estructura "serv_addr" con la dirección
// del servidor con el que nos queremos conectar

   serv_addr.sin_family          = AF_INET;
   serv_addr.sin_addr.s_addr     = inet_addr(dir_serv);
   serv_addr.sin_port            = htons(puerto);

   // Abre un socket TCP (Un socket Internet)
   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      printf("cte: No puede abrir un socket stream\n");
      return -1;
   }

   // Conexión al servidor
   if (connect(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
      {
      printf("cte: No se logro la conexión \n");
      return -1;
      }

   if(send(sockfd, point_datos, strlen(point_datos),0) == -1)
   {
         perror("send");
         return -1;
   }

// Si un hubo ningún error, entonces la transmisión del
// mensaje fue exitosa y se procede a cerrar el socket .

   close(sockfd);

   return 1;
}

/*****************************************************************************/
// Función: cpp_receive()
// Esta función es utilizada para recibir un mensaje y es bloqueante.
// cpp_receive( )  también utiliza el protocolo TCP.

int channel::cpp_receive(char *dir_serv, int puerto, void *point_datos)
{
   int            newsockfd,sockfd, clilen, childpid;
   struct sockaddr_in   cli_addr, serv_addr;
```

```
// Abre un socket TCP (Un socket stream Internet)
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    printf("No se puede abrir un Socket ... \n");
    exit(0);
  }

// Ligando el socket local de modo que los emisores de
//mensajes puedan enviar
 /* bzero((char *) &serv_addr, sizeof(serv_addr) );*/
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr  = inet_addr(dir_serv);
  serv_addr.sin_port        = htons(puerto);

  if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr))
    < 0) {
    printf("No se puede ligar la dirección local ...\n ");
    return -1;
  }

  listen(sockfd, 5);

 // Esperando para una conexión de un proceso Cliente.
 // Este es un ejemplo de un servidor concurrente

  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,&clilen);

  if (newsockfd < 0)
     {
      printf("Server: Error en accept ...\n");
      return -1;
     }

  if( recv(newsockfd, point_datos, sizeof(point_datos), 0)== -1)
    {
        perror("recv");
        return -1;
    }
//Si no se presento ningún error, entonces la recepción fue exitosa
// y se procede a cerrar el socket.
close(newsockfd);
  return 1;
}
```

Glosario

Objeto.- Es una abstracción de una entidad del mundo real. Es una entidad que encapsula alguna información de estado privada. El *comportamiento* de un objeto lo define una colección de operaciones que modelan su creación, destrucción e interacciones con otros objetos.

Estado.- El estado de un objeto son los valores que poseen las variables de la estructura que lo representa. Un objeto puede encontrarse exclusivamente en un conjunto de estados validos. La ejecución de una operación es el único medio para modificar el estado interno de un objeto y solo pueden llevar a un objeto de un estado valido a otro estado valido.

Objeto local.- Un objeto es local con respecto a otro si se encuentran ambos físicamente en la misma estación de trabajo; en otro caso el objeto se considera remoto.

Objeto persistente.- Un objeto se considera persistente cuando su estado se mantiene en un medio de almacenamiento estable

Clase.- Una clase es una colección de objetos con estructura y comportamiento similares. Un objeto es una instancia de una clase.

Proceso.- Informalmente un proceso es un programa en ejecución. Cuando un programa es leído del disco por el *núcleo* y es cargado en memoria para ejecutarse, se convierte en proceso y se componen de tres bloques fundamentales: Segmento de Texto, segmento de datos y segmento de pila (stack).

- **Segmento de Texto**.- Contiene las instrucciones que entiende la CPU de la maquina. Este bloque es una copia del bloque de texto del programa.

- **Segmento de datos**.- Contiene los datos que deben ser inicializados al arrancar el proceso. Si el programa ha sido generado por un compilador del lenguaje C, este bloque contendrá la variables globales y estáticas.

- **Segmento de pila (stack) .-** Lo crea el *Núcleo* al arrancar el proceso y su tamaño es asignado dinámicamente por el núcleo. La pila se compone de bloques lógicos llamados stack frames que guardan el contexto del programa.

Objeto-proceso.- Es un proceso creado al declarar un objeto de una clase derivada de la clase process del lenguaje ***C++ Distribuido***.

Bibliografía

[1] Andrews Gregory, R. [1991]. "Concurrent Programming: Principles and Practice". Benjamin Cummings.

[2] Olmedo, O. [1993]. "El lenguaje C++ Concurrente: Sinopsis y Ejemplos". Reporte Técnico. Sección Computación. Departamento de Ingeniería Eléctrica. CINVESTAV-IPN.

[3] Wu, Zhixue. [1993]. "Making C++ a Distributed Programming Language". IEEE.

[4] Carr, Harold and Kessler, Robert R. [1993]. "Compiling Distributed C++". IEEE.

[5] Gehani, Narain H. [1991]. "Concurrent C: Real-time programming and fault tolerance". Software Engineering Jurnal. Mayo 1991.

[6] Weldy, Dennis M. y Shiva, Sajian G. [1994]."A study of tasking models in concurrent C, SR and ADA". IEEE.

[7] Gehani, Narain H. [1993]. "Capsules : A Shared Memory Access Mechanism for Concurrent C/C++". IEEE Transactions on parallel and distributed systems, Vol4, No.7, Julio 1993.

[8] Brinch Hansen, P. [1978]. "Distributed process: a concurrent programming concept". CACM,21(11).

[9] Chin,S. [1991]. "Distributed object-based programming system". ACM Computing Survey. Vol. 23. No. 1, Marzo de 1991.

[10] Geist,Al. [1994]. "PVM 3 User's guide and reference manual". Engineer Physics and Mathematics Division. Mathematical Sciences Section.

[11] Gehani, Narain H. [1983]. "ADA, An Advanced Introduction". Prentice Hall.

[12] Wiener, Richard. [1983]. "Programming in ADA". John Wiley & Sons, Inc.

[13] Wegner, Peter. [1980]. "ADA. Programming whit ada : an introduction by means of graduate examples". Prentice Hall.

[14] Bal E. [1989]. "Programming lenguages for distributed computing systems". ACM Computing Survey.

[15] Gehani, N. H. [1988] "Concurrent Programming". International Cumputer Science Series.

[16] Coulouris, G. [1994]. "Distributed Systems: concepts and desing". Addison-Wesley.

[17] Gries, D. [1978]. "Programming Methodology: A collection of articles by membersof IFIP WG2.3". Springer-Verlag.

[18] Sarmiento Nava, Liuba. [1996]. "Unificando Criterios sobre Concurrencia y Distribución". Soluciones Avanzadas, Enero de 1996.

[19] Rifflet, Jean-Masrie. [1992]. "Comunicaciones en UNIX". McGraw-Hill.

[20] Márquez García, Fco. Manuel. [1994]. "UNIX Programación Avanzada". Addison-Wesley Iberoamericana.

[21] Godfrey, J. Terry [1991]. "Lenguaje Ensamblador para microcomputadoras IBM. Para principiantes y avanzados". Prentice Hall.

[22] Peterson, James L., Silberschatz, Abraham. [1982]. "Operatyng System Concepts". Addison-Wesley. University of Texas at Austin.

[23] Dijkstra, Edsger W. [1975]. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". Communitions of the ACM. Vol 18. No. 8.

[24] Bernstein, Arthur J. [1980]. Output Guards and Nondeterminism in "Communicating Sequential Processes". ACM Transactions on Programming Languages and Systems. Vol2, No.2, April. Pages 234-238.

[25] Boykin,Joseph, Kirschen, David. [1993]. "Programming under MACH. Edit". Unix and open systems series.

[26] Chandy, K. Mani y Misra, Jayadev. [1988]. "Parallel Program Desing. A Foundation". Addison-Wesley. University of Texas at Austin.

[27] Hekmatpour, Sharam. [1992]. "C++ Guía para programadores en C". Prentice Hall.

[28] Budd, Timothy. [1994]. "Introducción a la Programación Orientada a Objetos". Addison-Wesley Iberoamericana.

[29] Tanenbaum, Andrews S. [1996]. "Sistemas Operativos Distribuidos". Prentice Hall.

[30] Tanenbaum, Andrews S. [1996]. "Sistemas Operativos Modernos". Prentice Hall.

[31] Tanenbaum, Andrews S. [1996]. "Sistemas Operativos: Diseño e Implantación". Prentice Hall.

[32] Orfail,Robert. [1996]. "Essential Client/Server Survival Guide".Wiley.

Los abajo firmantes, integrantes de jurado para el examen de grado que sustentará **el Ing. José Mitre Silva,** declaramos que hemos revisado la tesis titulada:

"C++ Distribuido", consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente

Dr. Héctor Ruíz Barradas

Dr. Feliú Davino Sagols Troncoso

M. en C. Uriel Tirado Rios

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITECNICO NACIONAL

BIBLIOTECA DE INGENIERIA ELECTRICA
FECHA DE DEVOLUCION

El lector está obligado a devolver este libro antes del vencimiento de préstamo señalado por el último sello.

23 MAR. 2000

DEVOLUCION

AUTOR MITRE SILVA, J.

TITULO C++ DISTRIBUIDO

CLASIF. XM 98.14 RGTRO. BI 15361

NOMBRE DEL LECTOR	FECHA PREST.	FECHA DEVOL.
Héctor Ruiz Barradas		Dev
Armando Jiménez Flores	2/III/2000	4-III-00

BIBL