BENEMÉRITA UNIVERSIDAD
AUTÓNOMA DE PUEBLA
Facultad de Ciencias de la Computación

PRUEBAS DE PROCESOS SINCRONIZADOS EN UN AMBIENTE DISTRIBUIDO SIMULADO

Tesis que presenta

Alejandra Rufino Alonso

Para obtener el título de

LICENCIATURA EN INGENIERIA EN CIENCIAS DE LA COMPUTACIÓN

Asesor

M.C. Mariano Larios Gómez

Noviembre 2018

Dedicatoria

Gracias Dios por haberme dado las armas necesarias para seguir adelante, la capacidad de poder estudiar, la sabiduría para entender las cosas más difíciles y las fuerzas para finalizar uno de muchos procesos que vendrán. ***"Tú has hecho de mi lo que hoy soy".***

Con gratitud ilimitada, a mis Padres José Miguel y Dulce, sabiendo que no existirá una forma de agradecer una vida de sacrificio y esfuerzo, quiero que sepan que esta meta lograda también es de ustedes y de mis hermanos ya que la fuerza que me ayudó a conseguirlo fue su apoyo, su paciencia y su amor porque la presencia de todos ha sido y será siempre el motivo más grande que me ha impulsado para lograr esto.

Al fin concluye esta etapa tan importante de mi vida. Un periodo en el que aprendí no solo en las aulas, sino también de mis compañeros, amigos y profesores; principalmente de mi querido profesor Mariano Larios Gómez, muchas gracias por guiarme por el camino correcto, sus clases cambiaron mi vida, ahora que ya no será mi maestro, no puedo dejar pasar la oportunidad de darle las gracias por todo lo que me brindo en esta carrera; sus consejos nunca los olvidaré, y su amistad la voy a tener presente como el regalo más grande que puedo recibir de alguien. Gracias por todas las experiencias vividas.

Por último quiero agradecerle a Fer, porque es la persona que me comprende en todo y que da lo mejor de sí mismo sin esperar nada a cambio; porque sabe escuchar y brindar ayuda cuando es necesario, sin duda te has ganado el cariño, la admiración y el respeto de todo el que te conoce, primeramente la mía. Gracias por estar conmigo siempre.

ALE

Índice general

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Antecedentes

Este proyecto de tesis es parte del proyecto de investigación que fue aceptado en el laboratorio de supercómputo de la BUAP-LNS con número 201701071C. Gracias a su aceptación realizamos las pruebas propuestas en los objetivos y metas. De los cuales se obtuvieron resultados que sirvieron para la obtención de una tesis de licenciatura en ciencias de la computación como esta. Además de la publicación de trabajos en congresos nacionales e internacionales como en [1] [2].

1.2. Descripción del problema

En este proyecto de investigación de tesis se diseñó e implementó herramientas de pruebas (Test) para planificación de procesos capaces de mejorar los resultados en tiempo real en la asignación de tareas, con base a un consenso entre diversos nodos de un sistema distribuido, tomando en cuenta la calidad de retardo entre peer´s. De esta forma se obtuvo que los datos de un sistema distribuido se pueden trasladar en una red sin perder información y se permita la localización dentro de esta.

Este proyecto se basó en un ambiente distribuido, principalmente para dar una visión de planificación y ruteo en tiempo real, que permitió la obtención de buenos resultados sin pérdidas de información, buscando no tener mínimos locales. Se propuso trabajar en un ambiente distribuido, aplicando una conexión en red basada en nodos p2p. Una vez obtenido el resultado se diseñó y desarrollaron los algoritmos de planificación con el propósito de observar la calidad de retardo. Se propuso como plan de trabajo la investigación del estado del arte actual y la propuesta de algoritmos de planeación. La implementación y publicación de resultados sobre estos algoritmos se implementaron en los últimos meses de la realización de tesis.

1.3. Metodología de Desarrollo

La metodología que seguimos en este proyecto de investigación se basó en un principio en el estudio bibliográfico que conlleva en el ámbito de los sistemas distribuidos, sistemas de tiempo real y drones, con el fin de realizar un análisis y diseño de una red Ad Hoc en un sistema distribuido simulado bajo las reglas de uso que nos presentó el LNS [3]. Así como en la aplicación de algoritmos de protocolo de enrutamiento h´ıbridos [4], y algoritmos de protocolo reactivos, dependiendo de la conveniencia de la aplicación. Por tal razón se contempló aplicar algoritmos en una plataforma capaz de ejecutarlos en tiempo real, esta plataforma supercomputadora de la BUAP.

1.4. Objetivo del Proyecto

El objetivo principal de este trabajo de tesis fue el diseñar y desarrollar una propuesta educativa con mecanismo de Test para obtener los resultados, que son el crear la base de dos libros dirigidos a la comunidad de la Facultad de ciencias de la Computación Buap, que se espera se publiquen el siguiente año. Aportando a estos una investigación seria y completa sobre los temas actuales de los paradigmas de cómputo distribuido y cómputo concurrente.

1.5. Objetivos Específicos

- Se implementaron algoritmos en un lenguaje de programacion orientado a objetos como lo es Java.
- Se utilizaron librerías paquetes(API) para forzar al lenguaje de programación Java a utilizar recursos reales y crear un verdadero entorno concurrente y distribuido.
- Se realizaron las pruebas en la supercomputadora del LNS para la simulación de un ambiente distribuido.
- Se obtuvieron los resultados finales y las bases para la creación de los libros.
- Se obtuvieron y compararon los resultados con parámetros específicos.
- Se documentó el respaldo y restauración de la información de los libros.

1.6. Aportaciones e impacto socioeconómico

La aportación principal en este proyecto de tesis fue la investigación en el área de cómputo distribuido enfocado al testeo de planificadores de procesos en tiempo real. Dando una investigación documentada en un texto de tesis. Los resultados fueron presentados en un congreso científico y/o divulgación en el área de tecnologías, nacional o internacional. Material didáctico como una antología, libro o capítulo de libro sobre la programación concurrente y distribuida. Este proyecto fue parte de un proyecto realizado en LNS-BUAP y aportó resultados importantes a la comunidad científica.

Capítulo 2

Sincronización

2.1. Introducción

Una de las propuestas del proyecto de investigación de tesis fue el desarrollar una metodología de programación refinada y abstracta que se extendio al caso de los programas imperativos de competencia de recursos, con el fin de llevar a cabo la síntesis automática de programadores en tiempo real [5, 6, 7, 8] de una manera formal certificable. En concreto, en el caso de un conjunto de tareas dadas en Java (SCJ), es decir, compilar automáticamente el método de inicialización de su programa principal que proporciona parámetros de ejecución temporales con una ejecución correcta y eficiente.

El proyecto se basó en un conjunto de resultados recientes que ayudó a desarrollar la metodología de programación abstracta y refinada. En el cruce de la programación de la corriente de la información (estática) y en tiempo real (preventiva), que es formalmente abstracta de un problema de programación compleja (red de ciclo-estática) en una forma lineal para facilitar la solución de su programación lineal. Una vez una solución de dominio (resumen), obtiene una implementación heurística monótona que se utiliza para sintetizar el planificador de procesos. El software ADFG [7] implementa esta técnica.

Las llamadas de estudio considerado para el establecimiento de análisis técnico de programas de la competencia, SCJ o Real-Time Go, para que los representen en forma de autómatas temporizada (utilizando técnicas prestadas ”software de verificación de modelos”), estos controladores dan una representación simbólica de las operaciones y las comunicaciones de un programa con su entorno. El siguiente paso es abstraer la composición de estos autómatas como una red de flujo de datos de ciclo-estática para permitir el uso de ADFG. De este modo, también se trató de estimar los tiempos del ciclo simbólicamente consideradas tareas (en comparación con un período de referencia hiper), utilizando los mismos controladores y el tiempo de comunicación entre las tareas en el apoyo de carreras consideradas.

La tesis requirió amplia experiencia en las áreas de investigación de la teoría de programación (recopilación, análisis estático, interpretación abstracta), la competencia de recursos, además

de conceptos avanzados de la programación concurrente y distribuida. En este estado del arte nos enfocamos en los temas de ambientes distribuidos simulados y sistemas embebidos. En [9] se discute la aplicación de los resultados para el problema de sistemas multiagentes en tiempo real, aquı́se proponen dos algoritmos basados en la matriz Laplaciana del grafo G de redes que logra el consenso en un tiempo finito usando funciones Lyapunov, de esta manera se propone un mapa distribuido especial para la clase de grafos no dirigidos, y dio pie a la propuesta de este proyecto de generar una interfaz con el funcionamiento de grafos no dirigidos, además se propuso un análisis exhaustivo y el diseño de estrategias cooperativas para el consenso, otra contribución es la introducción a las condiciones necesarias y suficiente para dos algoritmos distribuidos discontinuos que lograron el consenso mínimo y máximo en tiempo finito y un consenso asintóticamente.

Como se propone en este trabajo de investigación de tesis, se plantea el uso de redes Ad Hoc, por tal razón se contemplan estos trabajos antes mencionados, por validar sus resultados en redes con topologías de intercomunicación y cambios dinámicos. Otro trabajo con propuestas novedosas sobre los algoritmos de consenso se encuentran en [9, 10, 11, 12], en estos artı́culo se re-direccionó el problema de consenso para sistemas distribuidos multiagentes no lineales y se aplica un controlador a grafos dirigidos y no dirigidos, además este control permite que se pueda trabajar en topologías fijas y cambiables. La aplicación de consensos y acuerdos en multiagentes en ambientes distribuidos y diseño de observadores lo podemos encontrar en [12], este trabajo se basa en las reglas de vecindarios L para la coordinación de multiagentes. En la búsqueda de un líder activo, se describe la dinámica del agente a seguir con entradas interactivas de control. Un ejemplo de las aplicaciones de los sistemas distribuidos (SD) se puede encontrar en [12], este trabajo se enfoca al problema de determinar una ruta óptima; a través de un algoritmo de descubrimiento de rutas. Se aplica el Algoritmo de concenso AC a un ambiente SD empleando una matriz de adyacencia, después se utiliza la distancia numérica para aplicar el consenso para el caso de estudio de topología móvil. Por ejemplo en [8], podemos observar la aplicación del control difuso basándose en la teoría de Takagi-Sugeno, En este experimento fueron basados en plataformas comerciales, aplicando la metodología de Takagi-Sugeno para el control con lógica difusa. Tanto en cuadricóptero y helicóptero se establece tres puntos de regulación. Estas condiciones son importantes para la obtención de las cuantificaciones deseadas, aunque se limitan a solo tres conjuntos de ellas. Otro trabajo de investigación donde se destaca el análisis y diseño de los sistemas de control en redes (NCS), se encuentra en [11], aquí se experimenta un control LQR (Linear-Quadratic Regulator) aplicado a los retardos en un sistema de tiempo real que se encuentra en un NCS. A comparación del trabajo en [10], en [11] se presenta un modelado usando Euler-Lagrange para el helicóptero en 2DoF. Basados en este modelo se aplica un control FF-LQR para controlar un helicóptero regulando al eje en ángulo pitch con FF. El

control FF+LQR+I usa un integrador en el lazo retroalimentado para reducir el error en estado estacionario. Usando la FF y la velocidad integral proporcional (PIV) se regula el ángulo pitch y sólo con la PIV se controla el ángulo de yaw.
El control LQR ajusta de manera efectiva el conjunto de frecuencias sin que el cálculo de la ley de control y la reconfiguración misma impacte en la transición. Es de resaltar que esta última reconfiguración disminuye el índice de desempeño provocada por una utilización de la red fuera de su ancho de banda o debido a bajo muestreo de la transmisión fuera de la región de planificación en los ángulos y velocidades en pitch y yaw, es decir, respectivamente. Otro trabajo de investigación es presentado en [12], donde se presenta una propuesta para el control del helicóptero. El objetivo es mantener el ángulo deseado de pitch y yaw, a través de dos hélices con dos motores como actuadores, además de otros casos de uso como la simulación de un sistema de levitación magnética, con un electro magneto como actuador, y dos sensores que miden la posición de una bola de acero y la corriente del electro magneto. Para mostrar la aplicación del codiseño de un NCS se presentan la pérdida de paquetes en una comunicación remota entre computadores o peers, basada en el protocolo UDP.

El control difuso diseñado es aplicado a ambos casos de estudio, utilizando el modelo difuso diseñado y el modelo de las imperfecciones. Se diseñan las matrices de retroalimentación para cada sub modelo discreto a través de un diseño LQR [12, 16]. Un servidor proxy es una solución software, implementada en la capa de aplicación de TCP/IP el cual intercepta los mensajes protocolarios (HTTP)para realizar la solicitud en representación de los usuarios de la red. Generalmente se encuentran ubicados en la frontera entre una red local y la red del ISP (Internet Servicie Provider) [13]. En otras palabras, un proxy funge como servidor intermediario (o procurador por su traducción al español) para el control del tráfico generado entre dos o más dispositivos en internet, simulando un sustituto de ruteo. Los servidores proxy se pueden encontrar en redes corporativas, instaladas con un diseño especializado para los dispositivos de una intranet. Y comúnmente utilizados por los ISP como parte se los servicios en línea que proveen a sus consumidores [14].
Las funciones claves que un servidor proxy comúnmente provee son:

- Cortafuegos y soporte para filtrado de tráfico.
- Conexión compartida de redes.
- Cache de datos.

Un proxy, como intermediario entre dos dispositivos, permite la solicitud de recursos, por ejemplo, a un servidor web, sin que este sepa a quién va dirigida la información realmente.

A modo de ejemplo, un proxy recibe la consulta de un Host A y realiza nuevamente la consulta hacia Servidor B, bajo un mismo protocolo de comunicación previamente definido, para posteriormente recibir la respuesta y redirigirla de vuelta al Host A.
Su aplicacion más conocida, en la totalidad de los casos, es la ocultación de los dispositivos que realizan las consultas (ej. Host A), comúnmente utilizados en prácticas ilegales como fraudes, suplantacion de identidad, spam, etc. Aunque su principal objetivo es la interconexión dedicada entre muchos ordenadores de una misma red a través de internet, como es en el caso de las VPN (Virtual Private Network), que en los últimos años ha tomado popularidad por su comunicación fuertemente cifrada a través de los ISP. Una VPN es capaz de cifrar el tráfico que pasa a través de él, que a diferencia de un proxy, el tráfico quedaría vulnerable, a menos que se defina un protocolo de cifrado a nivel de capa de aplicación.
Las principales ventajas son la reducción de tiempo de configuración (frente a una VPN), así como el filtrado de paquetes, la comunicación directa, el anonimato y la evitación de restricciones geográficas. Las desventajas son pocas aunque considerables como la implementación ante muchas peticiones puede comprometer su estabilidad y sobrecargar, el almacenamiento de datos en caché de los ordenadores, que por un lado proporciona un acceso más rápido a datos redundantes, pero pudiendo comprometer la confidencialidad de los datos trasmitidos, los proxys desconocidos, el cifrado puede ser vulnerado a menos que se utilice una capa SSL (Secure Socket Layer) y la incoherencia o desactualización, si se utiliza almacenamiento de cache, los datos almacenados en el proxy pueden diferir de los datos actualizados en el servidor destino.

2.2. El problema de sincronización

Al codificar cualquier programa concurrente se debe ser capaz de sincronizar los diferentes threads de forma confiable. Sin la sincronización, dos threads pueden cambiar datos al mismo tiempo y sobre escribir uno del otro.
El problema principal de sincronización se encuentra en las repeticiones que realizan los procesos al tratar que la condición sea verdadera, y así entrar a su SC. Estas repetidas acciones de entrar a la SC provocan la perdidad de ciclos de CPU es por eso que definimos a este problema como: *espera activa*. Centrandonos en el problema de sincronización podemos decir que es importante el orden de la ejecución de los procesos, dando por hecho que ya se resolvio el problema de condiciones de competencia por medio de la exclusión mutua.

Espera activa(busy wait)

Mientras un proceso revisa repetidamente porque la condición sea verdadera, su trabajo no se realiza, además de provocar perdida de ciclos de CPU.

Por ejemplo. Un banco tiene un thread que calcula los dividendos de una cuenta bancaria. En el mismo instante en que un thread calcula los intereses del fin de mes y un segundo thread actualiza el deposito de un cheque de *20,000* Suponiendo que se tienen *10,000* pesos al 1 %, entonces tenemos un nuevo balance (newBalance) de 10100 pesos. El thread deposita el cheque y actualiza la cuenta que ahora tiene 30100 pesos. En caso que primero se deposite y luego se obtenga el interes, tendremos como resultado 30300 pesos!

```
(1) Thread  1                    (1)Thread 2
(2) temp=getBalance();           (2)ChekDeposito();
(3)div=temp*getInteres();        (3)temp=getBalance();
(4)nuevo=div+temp;               (4)nuevo=deposito+temp;
(5)setBalance(nuevo);            (5)setBalance(nuevo);
```

En el segmento de código anterior, hacemos referencia al peoblema del banco con 2 threads. En la línea 4 de ambos threas se puede observar el problema de sincronización.

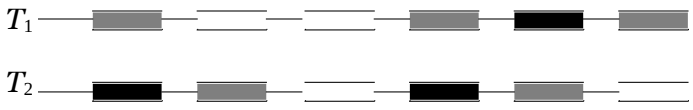


Figura 2.1: Diagrama de threads del problema de la cuenta bancaria.

Donde los bloques de los threads en el *"diagrama de threads"* es:

En esta etapa hacemos referencia a las primitivas de mutex look y unlook(), estas primitivas las representamos con diagramas de thread y las podemos ver en la figura 2.1. Así también podemos observar los bloques que nos permiten señalar el estado actual de un proceso.

Para la sincronización se necesitan estructuras especiales y conjuntos de funciones para manipularlas. Java ofrece esta funcionalidad encapsulando las variables de sincronización en

todos los objetos (synchronized y thread.join()) Estos son método de coordinación de las interacciones entre threads.
Se pueden ver en dos partes básicas de sincronización:

- Proteger los datos que comparten(bloqueos).
- Evitar que se ejecuten threads cuando no tiene ninguna acción que realizar.

2.3. Problemas clásicos de sincronización

2.3.1. Productor consumidor

El problema del productor/consumidor consiste en tener un personaje llamado *Productor* y de otro personaje llamado *Consumidor*. Estos dos personajes tienen en común un objeto llamado *producto,* este producto es utilizado por los dos personajes productor/consumidor. Este problema también es conocido como .El problema del almacen limitado"según [15]. El problema en el productor-consumidor es que solo tiene un buffer para almacenar el producto y para tomarlo, esto nos causa problemas en la exclusión mutua y ahora en la sincronización. En el caso de uso de la figura 2.2 se pude apreciar como el productor debe producir para que el consumidor pueda realizar consumo.

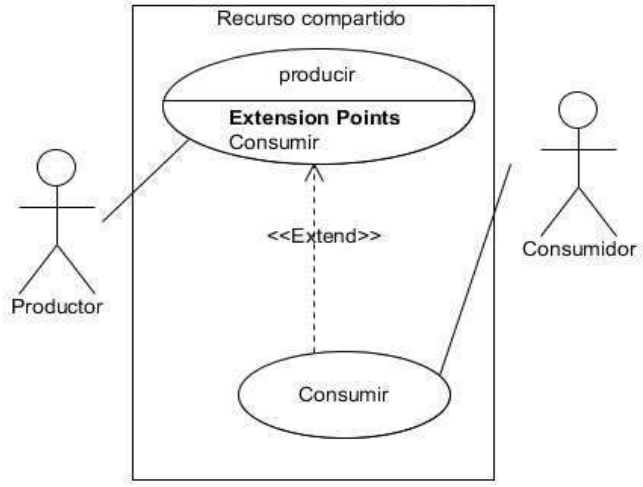


Figura 2.2: Caso de uso del productor-consumidor.

2.3.2. El problema de la cena de los filósofos

E. W. Dijkstra en 1965 [16], planteó la solución de un problema de sincronización llamado el *problema de la cena de los filósofos*. El problema consiste en: cinco filósofos se sientan a la mesa y cada uno tiene un plato de spaghuetti, entre cada plato hay un tenedor y cada filósofo tiene un plato enfrente. El filósofo debe comer con dos tenedores. La vida de un filósofo consta de periodos alternados de comer, pensar y dormir. Cuando un filósofo tiene hambre, intenta obtener dos tenedores. Si un filósofo toma un tenedor, entonces espera hasta obtener el otro tenedor entre el resto de los filósofos. Dikstra propone a cinco personajes llamados *filósofos*, los cuales tienen tres estados:

- Pensando
- Hambriento
- Comiendo

La propuesta original de Dijkstra se basa en que los filósofos coman spaghuetti con dos tenedores, por ser tan resbalozo. Pero para ver la necesidad de utilizar más de un utencilio, se puede pensar en dos palillos chinos y comida japonesa. Se plantea que los filósofos esten sentados en una mesa redonda y cenar comida japonesa en cinco platos y cinco palillos chinos repartidos como se muetra en la figura siguiente. El caso de uso de la cena de los filósofos en la figura 2.3 muestra al filósofo que puede tomar sus tres estados. Se restringe el poder tomar el estado de comiendo con la condición de tomar los dos palillos chinos que se encuentran a la izquierda y a la derecha de cada uno de los filósofos. De forma general el actor filósofo debe preguntarse por los palillos tanto el derecho como el izquierdo y así puede pasar a su estado comiendo(sección crítica), con la condición:

```
if(filosofo.obtienePalilloIzquierdo()& & filosofo.obtienePalilloDerecho())
```

Esta condición nos exige que el filósofo tenga necesariamente los dos palillos para poder comer. Es por eso que en la figura 2.3 el caso de uso *Comiendo* extiende (extends) de los casos de uso ”*Toma palillo izquierdo”* y del caso de uso *”Toma palillo derecho”*, si se cumple estos dos últimos casos de uso , entonces se cumple el caso de uso *Comiendo*. Los casos de uso *Pensando* y *Hambriento* son adquiridos en el momento que el filósofo termina de comer o así también se puede adquirir en cualquier momento que el filósofo lo disponga. El algoritmo propuesto por Dijkstra para la cena de los filósofos se puede encontrar en [15].

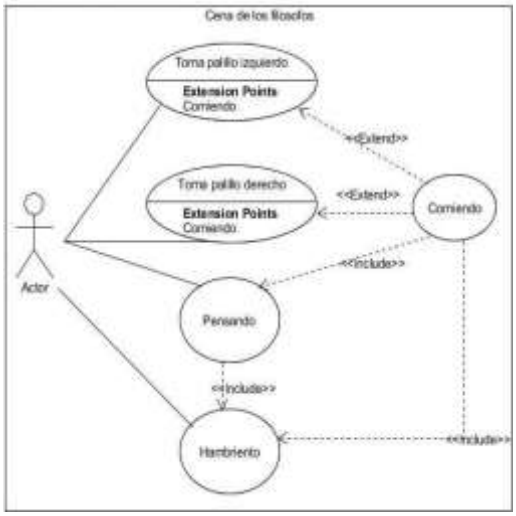


Figura 2.3: Caso de uso filósofos.

Algorithm 1 Algoritmo del problema de la Cena de los filósofos

```
1: public class filosofos{
Require: final eating 2
Require: Semaforo s;
Require: Int state[N];
2: public void filosofos(int i){
3: while true do
4:     pensar();
5:     toma_tenedor(i);
6:     comer();
7:     poner_tenedor(i);
8: end while
9: }
10: public void toma_tenedor(int i){
11: down(mutex);
12: state[i]=hambriento;
13: test(i);
14: up(mutex);
15: down() }
```

Algorithm 2 Algoritmo del problema de la Cena de los filósofos Parte II

```
1: public void put-tenedor(int  i){
2: down(mutex)
3: state[i]=pensar;
4:  test(izq);
5:  test(der);
6: up(mutex); }
7: Public void test(int i){
8: if state[i]==hambriento& &state[izq] != comiendo & &state[der]!=comiendo then
9:     state[i]=comiendo;
10:    up(a[i]);
11: end if
12: }}
```

Dijkstra plantea la combinación de operaciones para la toma de tenedores por medio de la función: P(left hand fork, right hand fork). Los estados de los filósofos se guardan en un vector de estado $\dot{C}$:

C[i]=0

C[i]=1

C[i]=2

La condición básica sería: $\exists i$ (C[i]=2 and C[(i+1) mod 5]=2), para restringir que el filósofo no puede comer si su vecino a la izquierda esta comiendo.

($\exists k$ (C[(K-1) mod 5] $f=$ 2 and C[K]=1 and C[(K+1) mod 5] $f=$ 2)(1)
La condición con respecto a la formula 1 a utilizar es:

Algorithm 3 Toma los dos tenedores.

```
if C[(k-1) % 5] != 2 & & C[k]==1 & & C[(k+1) % 5] != 2 then
   C[k]=2;
   V(Sem[k]);
end if
```

En este universo la vida de un filósofo (w) se puede codificar como:
Sea w los estados del filósofo, w={comiendo, pensando, hambriento}={0,1,2} tomando el conjunto w podemos proponer un algoritmo que verifique el estado de los vecinos de un filósofo como se muestra en el algoritmo 4.

Algorithm 4 Algoritmo con función de chequeo en los vecinos

```
1: while true do
2:     filosofo.pensar();
3:     C[w]=1;
4:     test(w);
5:     V(mutex);
6:     P(Sem[w]);
7:     filosofo.eat();
8:     C[w]=0;
9:     test[(w+1)]
10:    test[(w-1)]
11:    V(mutex);
12: end while
```

2.3.3. El problema de los Lectores y los Escritores

Courtois en 1971 [4] propuso el problema de los escritores y lectores; el cual se basa en el acceso a una base de datos o un buffer compartido entre varios procesos. Una aplicación a este problema es un sistema de reservaciones de una línea aérea, con muchos procesos en competencia que intentan leer y escribir en la base de datos. Los procesos pueden leer la información en cualquier momento y al mismo tiempo, pero si algún proceso esta escribiendo, ningun otro proceso puede leer o escribir la información (recurso compartido), ni los lectores pueden tener acceso al recurso. En este problema de sincronización se tiene una estructura de datos compartidos que se lee a menudo, pero que se escribe de vez en cuando. En este problema sería un desperdicio de tiempo usar mutex para la solución, pues se exigiría que todos lo threads recorran uno a uno cuando no esten modificando nada. De ahí, es el problema de *lectores/escritores*.

En el caso de uso de los escritores y lectores en la figura 2.4 muestra la restricción en que los lectores no podrán leer, si algún escritor esta haciendo uso del recurso compartido (en este caso la base de datos), esta restricción se debe realizar por la relación *extends* que pasa por el caso de uso *leer* al caso de uso *escribir*.

Despúes de que el proceso escribe en la base de datos, los procesos que leen podrán seguir con su ejecución. El algoritmo de los escritores-lectores según [4] se encuentra en (Algorithm 5, Algoritmo del problema de los lectores escritores).

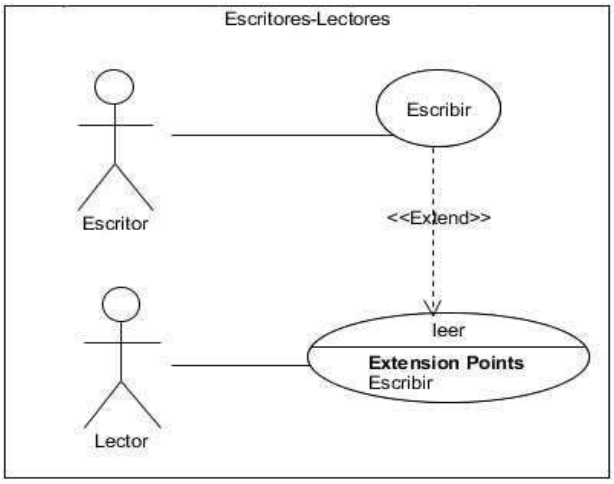


Figura 2.4: Caso de uso Lectores-Escritores.

2.3.4. El problema del barbero dormilón

Otro de los problemas clásicos en la sincronización entre procesos es el barbero dormilón. En este problema se tiene una peluqueria, una silla de peluquero y sillas de espera para que se sienten los clientes a esperar su turno, si es que los hay. Si no hay clientes en espera, entonces el barbero se sienta a dormir en la silla de peluquero. Si llega un cliente y el barbero esta durmiendo, este último despierta y el cliente se sienta en la silla de peluquero y el barbero lo atiende. Si llegan más clientes y el barbero esta ocupado cortando el cabello a un cliente, ellos se van sentando en las sillas de espera hasta que les toque el turno. Se desea que el barbero y los clientes no entren en competencia por el recurso compartido que es la silla. En la figura 2.5 se muestra el caso de uso del barbero dormilon donde se representan las condiciones que existen en el uso de las sillas, que son los recursos compartidos, así como la interacción entre el proceso barbero y los procesos clientes.

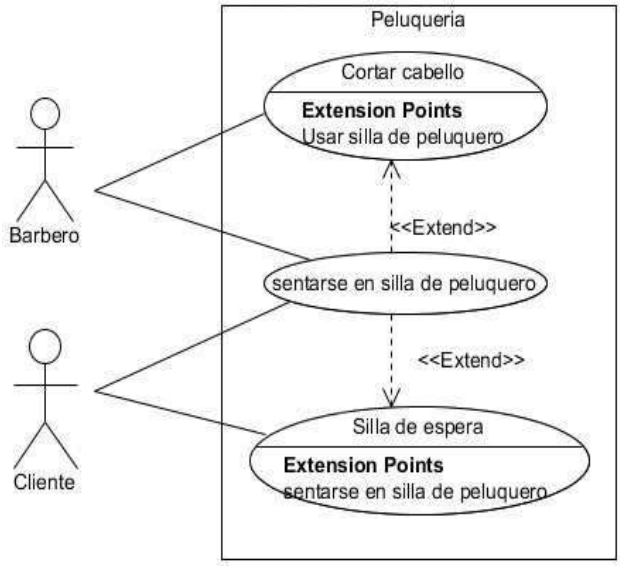


Figura 2.5: Caso de uso del barbero dormilón.

2.3.5. El problema de los fumadores de tabaco

En 1971 S. S. Patil [18] planteó el problema de los fumadores de tabaco. Asumiendo que un cigarrillo requiere de tres ingredientes para poder fumarlo:

- Tabaco
- Papel
- Un fósforo

Asumiendo que hay tres fumadores empedernidos alrededor de una tabla, cada uno tiene una fuente infinita de uno de los tres ingredientes anteriores, por ejemplo el $fumador_1$ tiene una fuente infinita de tabaco, el $fumador_2$ posee una fuente infinita de papel, y por último el $fumador_3$ tiene una fuente infinita de fósforos. Tomando en cuenta que se tiene un árbitro no fumador. El árbitro permite a los fumadores hacer sus cigarrillos seleccionando dos de los fumadores arbitrariamente (no-determinadamente), sacando un artículo de cada uno de sus fuentes, y poniendo los artículos en la tabla. El entonces notifica al tercer fumador que se ha realizado la selección. El tercer fumador quita los dos artículos de la tabla y los utiliza (junto con su propia fuente) para hacer un cigarrillo, que fumará durante algún tiempo. Mientras tanto, el árbitro, viendo la tabla vacía, elige otra vez a dos fumadores al azar y pone sus artículos en la tabla. Este proceso continúa por siempre. Los fumadores no amontonan artículos de la tabla; un fumador comienza solamente a rodar un nuevo cigarrillo una vez que lo acaban que

fuma el anterior. Si el árbitro coloca el tabaco y el papel en la tabla mientras que el hombre del fósforo está fumando, el tabaco y el papel seguirán estando disponibles en la tabla hasta que acaben con su cigarrillo y los recoja el proceso del fósforo. El problema es simular los cuatro papeles como programas informáticos, usando solamente cierto sistema de primitivas de la sincronización. En Patil la formulación original de las primitivas de sincronización son permitibles con un semáforo, y no se permite ninguno de los cuatro programas, se encuentra en (Algorithm 6 Algoritmo problema de los fumadores).

2.4. Técnicas de Sincronización

2.4.1. MUTEX

Existen varios métodos que garantizan la exclusión mutua y mutex es uno de los primeros métodos que dierón mejores resultados, pero ahora pasa a competir en la sincronizacion de procesos contra métodos más apegados a la sincronización basados en las primitivas del mutex (lock y unlock). Para crear estas primitivas en Java, se utilizan los métodos wait() y notify().

Usando el programa del productor−consumidor bloqueamos al acceso al recurso compartido para que un productor escriba y un consumidor. El mutex se envia por medio del constructor tanto al productor y al consumidor. Las clases P/C heredan de la clase thread y en la clase principal main se crean los objetos de la clase productor y de la clase consumidor y al mismo tiempo se les envia el objeto mutex y el objeto producto para que tanto el productor y el consumidor compartan estos dos recursos se encuentra en (Algoritmo 7 y 8, Mutex y Mutex y contunuacion).

Algorithm 9 Productor extends Thread

```
Require: Producto p;
Require: int dato;
Require: Mutex m;
 1: Productor(Producto p,Mutex m){
 2:   m=m;
 3:   p=p; }
 4: void run(){
 5: for int i=0;i<10;i++ do
 6:     while !p.vacia() do
 7:         m.lock();
 8:         dato=(int)(Math.random()*20);
 9:         p.escribe(dato);
10:         m.unlock();
11:     end while
12: end for
13: }
```

Algorithm 10 Consumidor extends Thread

```
1: Producto p;
2: int dato;
3: Mutex m;
4: Consumidor(Producto p,Mutex m){}
5: void run(){
6: m.lock();
7: dato=p.obten();
8:  m.unlock();}
```

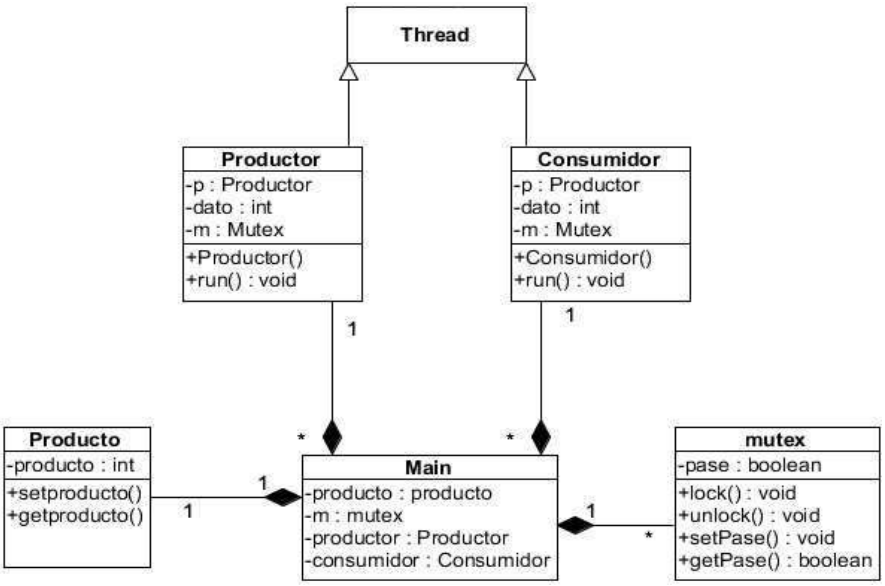
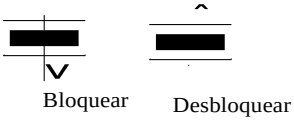


Figura 2.6: Diagrama de clases del P/C con MUTEX.

La representación del mutex en el diagrama de threads se muestra por medio de un bloque de forma rectangular sombreado y cruzado por una flecha hacia abajo o la flecha hacia arriba. El bloque con flecha hacia abajo representa al método lock() que realiza el *bloqueo* y la flecha hacia arriba representa al método unlock() que representa el *desbloqueo*, dependiendo del estado del recurso compartido, si esta ocupado o desocupado.



Existe la posibilidad de que algún otro thread llame a lock() y obtenga el recurso compartido antes de que lo haga el que se acaba de reactivar. La desventaja del *mutex* consiste en que su funcionamiento esta hecho para que los threads obtengan su S.C. por medio de la fuerza bruta. Por ejemplo suponiendo que se tienen tres threads con diferentes prioridades y se determina el orden en que pasan a la cola de espera, los threads han solicitado el bloqueo en el orden T_1, T_2 y T_3, al ser el thread T_1 el primero en ser ejecutado, el thread T_1 obtiene el bloqueo y el resto de los threads pasan a formarse en la cola de espera, pero el siguiente thread en obtener el bloqueo va a depender de la fuerza de su insistencia, pues el primero en bloquear va a ser el que más insistencia realizo. Es decir, Los threads T_2 y T_3 no tiene un orden al momento de la obtención del bloqueo, aunque se formen en una estructura organizada como una estructura cola. Esto es porque el mutex no sabe de prioridades es decir de orden en ejecución y además la estructura cola sólo es una forma de abstraer el concepto del comportamiento de los threads.

En Java la forma de realizar un bloqueo por medio de un mutex que esta acompañado por la clase *syncronorized*. Esta clase contiene la abstraccion del funcionmiento de un monitor, pero como Java carece en el manejo de recursos reales, nosotros utilizaremos los métodos wait(), notify() y notifyAll() de la clase **Thread**, y es así como implementamos un *Mutex* en Java.

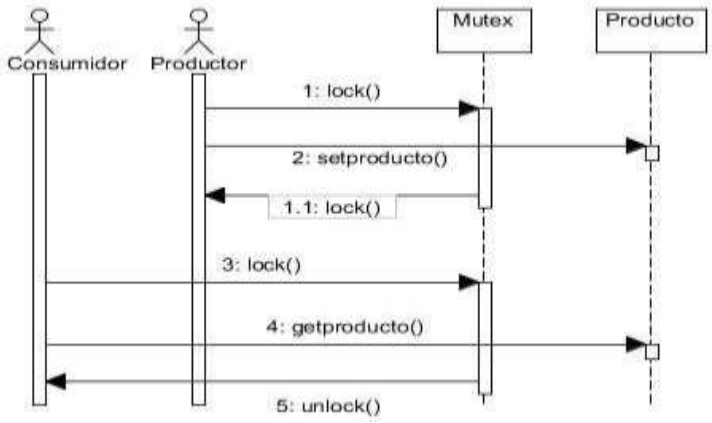


Figura 2.7: Diagrama de secuencia del P/C con MUTEX.

2.4.2. Semáforos

Los semáforos se utilizan para controlar el acceso a memoria, archivos, dispositivos de entrada y salida, entre otros. En general a los recursos compartidos de forma sincronizada.

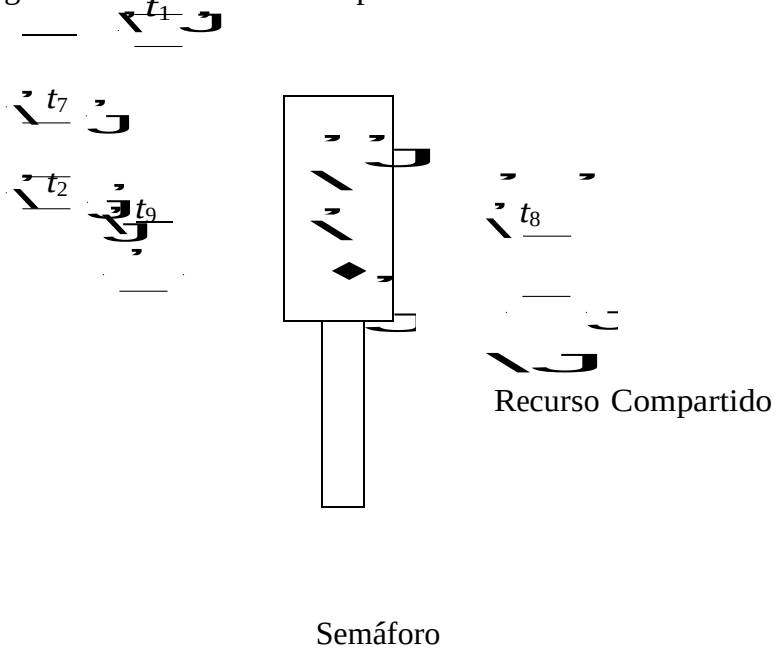


Figura 2.8: Semáforos que sincroniza el acceso al recurso compartido

Características de los semáforos

1. **Desbloqueando procesos.** La primitiva de sincronización llamada semáforo, informa el estado de los procesos que se encuentran bloqueados. Los procesos se desbloquean, y pasan a estado listo, de esta forma el administrador de procesos elije quién pasa a ejecución (Dependiendo de la implementación: FIFO, LIFO, otras estructuras abtractas de datos.).

2. **Atomicidad.** Se garantiza que al iniciar una operación con semáforo, ningún otro proceso puede tener acceso a éste hasta que la operación termine o se bloquee. En este tiempo ningún otro proceso puede simultáneamente modificar el mismo valor de semáforo.

Semáforos Binarios o Contadores

- Útiles cuando un recurso es asignado, tomandolo de un conjunto de recurso idénticos.

- La primitiva de sincronización *semáforo* es inicializada con el número de recursos existentes: P(S) decrementa a S en 1; indicando que un recurso ha sido suprimido del conjunto. Si **S=0** entonces no hay más recursos y el proceso se bloquea, **V(S)** incrementa a S en 1; esto indica que un recurso ha sido regresado al conjunto. Si un proceso esperaba por un recurso, éste despierta.

Semáforos Binarios I

- Los semáforos binarios utilizan una variable booleana, así como una estructura de datos llamada *Cola de procesos bloqueados*, un proceso puede agregarse a ésta, llamando al método P(). Una vez realizando está llamada el valor de V=true, obteniendo como valor del semáforo=false. Cuando esto sucede los procesos obtienen el bloqueo hasta que sea true. La llamada del método V() hace a V=true y también notifica si la cola de *procesos dormidos* en el semáforo es No-Vacia.

- Son utilizados por dos o más procesos para garantizar que sólo uno puede entrar a su sección crítica.

- Antes de entrar a la sección crítica un proceso ejecuta una llamada P(S) y antes de salir de ella realiza una llamada a V(S).

- La variable de tipo semáforo se llama *entrar* y es inicializada en 1. Cada proceso tiene la estructura siguiente:

Semáforos Sincronizadores

- Este tipo de primitiva de sincronización puede ser usado para resistir una gran variedad de problemas de sincronización.
- En Java se usa la idea de monitores para su implementación.
- Sean dos procesos concurrentes P_1 y P_2 que se encuentran corriendo: P_1 con enunciado S_1, P_2 con enunciado S_2. Se desea que S_2 sea ejecutado después de que S_1 haya terminado. Solución:

Semáforo sincrono (inicializado en 0)

P_1: ——- P_2:——-

S_1 S_2

P(sincro) V(sincro)

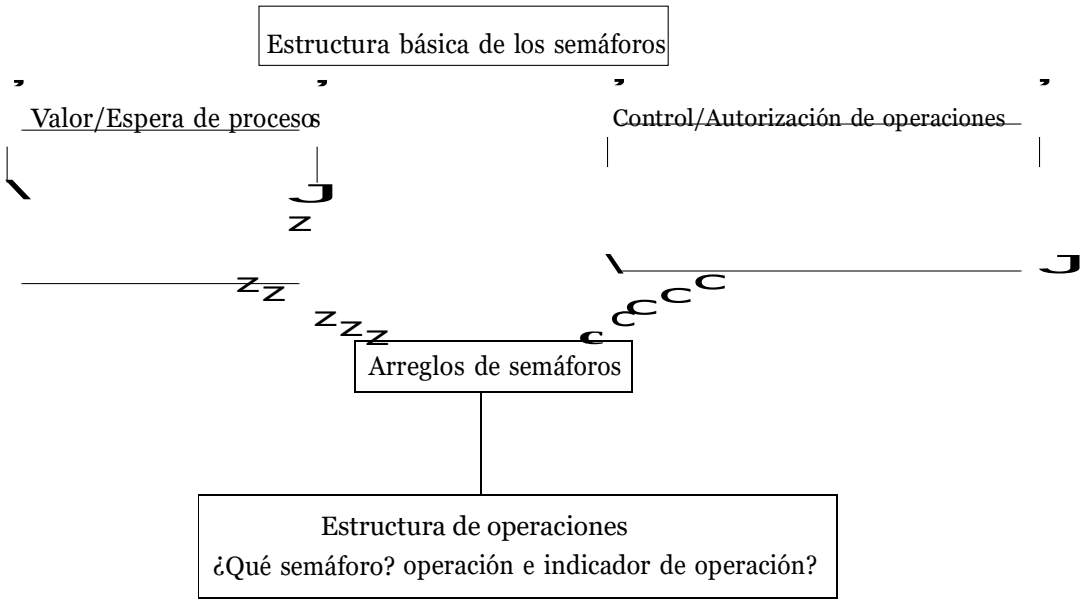


Figura 2.9: Estructura básica de los semáforos

Al ser elemento de datos compartidos, el valor del semáforo debe protegerse en una S.C sem-Wait() al entrar en dicha sección sincronizada (bloquea el mutex). Después comprueba el valor si v>0 entonces se decrementa la variable *v* (v- -). Se libera el mutex y se llama a semWait( ). Si y sólo si v=0 entonces libera el mutex y el thread se pondrá en espera. Tras reactivarse, el thread debe repetir la operación, y volverá a adquirir el mutex , probar el valor semPost(), bloquear el mutex, incrementar el valor, liberar el mutex y reactivar un thread en espera.

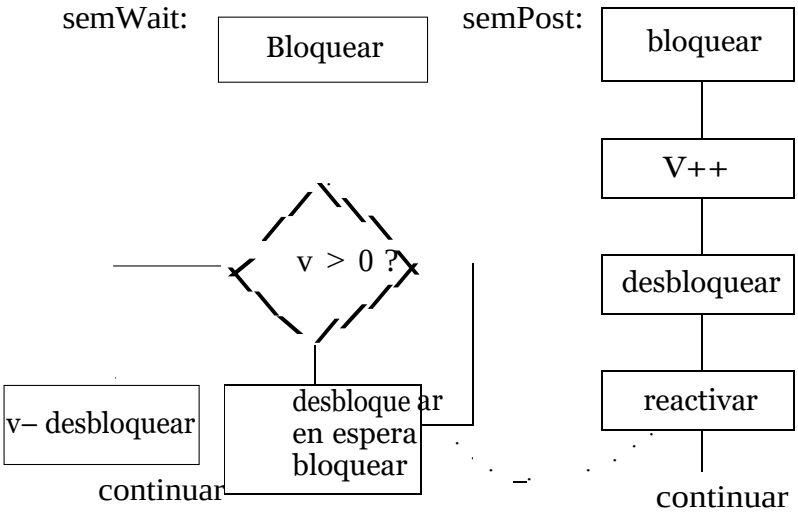


Figura 2.10: Diagrama de flujo del método *"Semáforo"*

La clase semáforo utiliza un mutex en sus primitivas UP y DOWN, pero este mejora al método mutex usando una condición para el acceso al semáforo.

Otra forma para implementar un semáforo sin utilizar un objeto mutex es como se muestra en el siguiente código, pero si analizamos el uso del método wait(), podemos observar que se implementa al mutex de forma implicita.

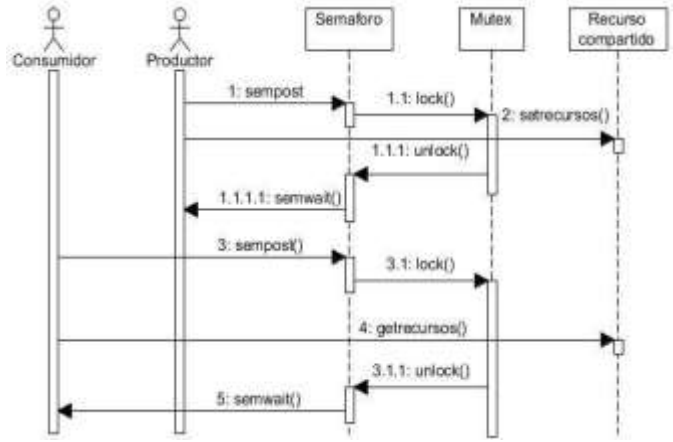


Figura 2.11: Diagrama de secuencia del P/C con MUTEX

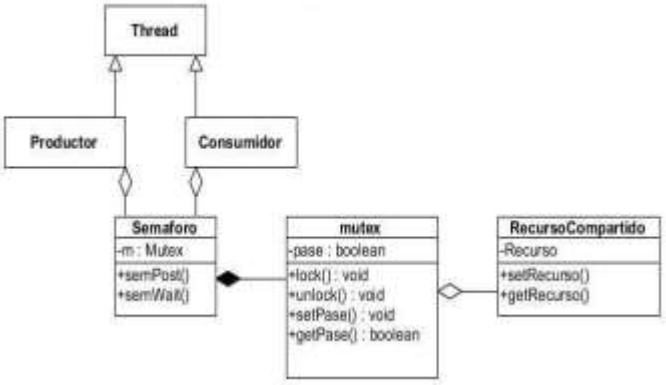


Figura 2.12: Diagrama de clase del P/C con Semáforo

2.4.3. Variables de condición

En el desarrollo de sistemas, el programador es responsable de bloquear y desbloquear el mutex, de probar y cambiar la condición y además de reactivar los threads en espera. Un thread obtiene un mutex (las variables de condición siempre tiene un mutex asociado) y prueba la condición bajo la protección del mutex. Ningún otro thread deberá alterar ningún aspecto de la condición sin tener el mutex. Si la condición *condVar==true* se cumple, entonces el thread realiza su tarea y libera el mutex cuando sea pertinente o en caso de que ya no utilize el recurso compartido. Si no es así, entonces el mutex se libera automáticamente y el thread se pone en espera de que la *variable de condición* sea verdadera. Los problemas que nos ayuda a resolver las variables de condición es cuando el thread debe evaluar la condición, porque puede existir la posibilidad de que otro thread no haya probado toda la condición antes de enviar la reactivación o también

que la condición sea *true* cuando se envío la reactivación, puede haber cambiado antes de que se ejecute el thread. y por último las *variables de condición* no permiten reactivaciones falsas [1]

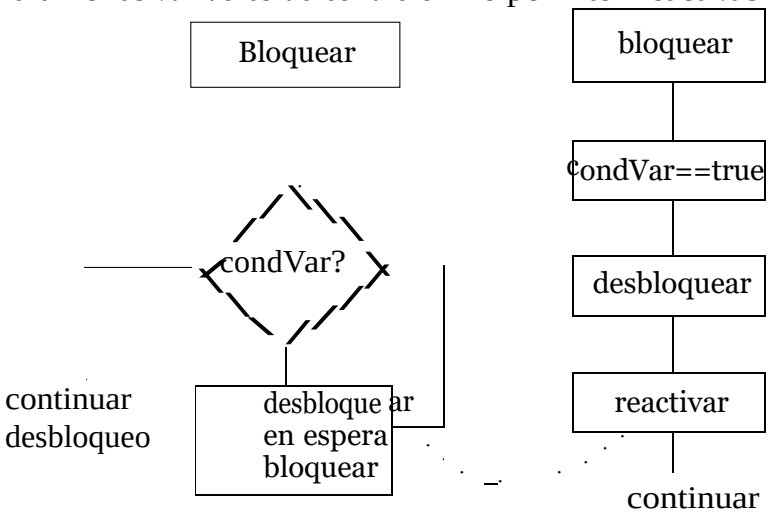


Figura 2.13: Diagrama de flujo del método *"Variable de condición"*.

En la clase de las *variables de condición* enlistada abajo, se puede observar el uso de un mutex para bloquear el acceso al recurso compartido. En la línea() se usa un mutex para la clase *variable de condición* y en los métodos *espera()* y *señal()* implementadas con las primitivas *wait* y *signal* líneas (3) y (9) respectivamente, se usa un desbloqueo e inmediatamente después se realiza un wait() para que la *variable de condición* permita bloquear al recurso y poder entrar a la sección crítica al thread que la pida. en el método senal() del código, esto hace que espere hasta que sea instanciado y poder liberar el recurso notificando a los demás threads que ya se puede tomar el recurso compartido.

[1] Permite reactivarse sin ninguna razón

2.4.4. Monitores

La primitiva de sincronización *Monitores* simplifica tareas que pueden realizarse de manera simultánea; se consideran una especie de organizadores para cada tipo de recurso. Los Monitores son estructuras de datos administrativas, funciones, procedimientos y llamadas a programas. Podemos decir que los monitores pueden ser de canales similares para intentar administrar recursos similares. Los monitores tienen dos métodos o primitivas principales [19]:

- wait: Inserta un thread en una estructura de tipo cola.
- signal: Avisa que un thread puede entrar a su sección crítica y que otro se encuentra en el estado de terminar y continuar su ejecución.

Los monitores manejan *variables de condición* para controlar la espera o el despertar de un procedimiento; estas variables de condición se pueden observar como una cola invisible de procesos en espera. Es a través de esta condición que se puede accesar a los métodos *wait* y *signal*:

- condicion.wait
- condicion.signal

A continuación se muestra un ejemplo de monitor que tiene dos procedimientos:

- *acquire*: Para entrar a la sección crítica.
- *release*: Determinar la salida de la sección crítica.

Una variable booleana:

- *busy*: Indicará si la sección crítica esta ocupada (true) o no (false).

Una variable de condición:

- *nonbusy*: Como todas las de su tipo, podra esperar en un wait y despertar en un signal.

Describiendo el monitor, cuando entra al método *acquire()*, pregunta si la variable *busy* es verdadera, indicando que está un proceso en su S.C. por lo que se realiza un *nonbusy.wait*. Al despertar por un signal, se colocará busy a verdadero para indicar que ya esta en la S.C. Ahora bien, *release*, pone a busy en falso y hace un *nonbusy.signal* para despertar a los procesos que estabán en espera. Es importante mencionar, que hasta ahora, si hay más de dos procesos que hayan realizado un wait, se despertará primero el que tenga más tiempo dormido dentro de la cola de espera de threads.

Interpretación y Reglas de prueba.

Realmente el procedimiento anterior demuestra que mediante monitores podemos implementar semáforos, donde la función *acquire* equivale a un *down* y *release* a un *up*. Así también existe la posibilidad de la acción viceversa con semáforos, es decir, programar monitores con semáforos. Con el apoyo de dos contadores y tres semáforos se puede implementar los métodos wait y signal.

Al mencionarse las llamadas *Reglas de prueba*, se pone en manifiesto la relación entre monitores y la representación de datos, pues a ambas se les asigna una invariante que resulta verdadera antes y después de cada llamada de procedimiento. El valor de ésta, ya sea antes o después de un wait o un signal detemina la continuación de la ejecución del programa. Todos los procesos esperan en una variable de condición *b* asociada a una afirmación, validando el acceso y salida de dichos métodos. Si se entabla un cuestionamiento con las variables de condición puede pro- vocar abrazos mortales en monitores. Una técnica que algunos utilizan para evitar los deadlock (ver sección 2.2.1), es el uso, por parte de los monitores de recursos hasta cierto punto *virtuales* para cada programa.

Ejemplo principal y Esperas programadas.

Otro ejemplo de monitores es el ”Bounded Buffer”ó ”Buffer Limitado”, el cual maneja un arreglo estático de porciones limitadas para ser utilizadas. La función *append* del monitor agrega al final del arreglo *buffer* una nueva porción a considerar, mientras que *remove* elimina la primera localidad de búfer. Se manejan dos variables como apuntadores, *lastposition* que maneja la siguiente posición en la cual debe ser agregada una porción y *count* que cuenta el número de porciones almacenadas; además de dos variables de condición *nonempty* y *nonfull*, las cuales esperan por que no hay porciones y por estar lleno el búfer respectivamente. Para las esperas calendarizadas u organizadas, se les da prioridad a los threads en espera para desper- tar de la cola invisible en wait; dicha prioridad ***p*** se pasa como parámetro en el método wait: condicion.wait(p).

Se observan tres ventajas fundamentales:

- Mejora en tiempo.
- Se aprovecha mejor el tiempo en la CPU

  Cada monitor puede reservar memoria proporcional al número de clientes.

Se introduce una nueva operación en las variables de condición: condicion.queue, que retorna un valor booleano; verdadero para indicar que hay procesos esperando en esa condición o falso si no lo hay.

Para ilustar estos método, se menciona el ejemplo de un monitor *alarmclock*, que tiene dos funciones, una llamada *wake* y la otra de nombre *tick*. El trabajo de wait, es colocar en una variable *alarmsetting* el numero *n* de pulsaciones o *ticks* en los cuales el thread despertará, mediante la suma de la hora actual dada en la variable *now* más *n*. Mientras *now* sea menor a la hora de alarma (now menor que alarmsetting) se pondrán en wait la variable de condición *wakeup* con la asignación de prioridad dada por alarmsetting. El método tick sólo incrementa la hora actual, es decir, *now*, y da un wakeup.signal.

2.5. Complejidades de sincronización

Existen aun más variables o métodos de sincronización más complejas. Existen problemas en que las variables de sincronización no proporcionan suficiente funcionalidad en situaciones que se puede mejorar la eficiencia de una aplicación mediante la implementación de primitivas de bloqueo más complejas. Estas primitivas tiene la desventaja de ser más lentas, por que tiene más instrucciones y más condiciones.

Las primitivas de bloqueo más complejas resuelven problemas que las variables de sincronización anteriormente fallan. Por ejemplo, el problema de prioridades con mutex. Para ser justos con las prioridades y con el rendimiento global del sistema, se deben respetar las prioridades y modificar estas en caso necesario. Seria muy útil poder tener herencia de prioridades en Java, para el manejo de sistemas de *tiempo real*, pero al no ser así, en Java la cuestión de los mutex da cierta controversia. Aún asi se trata de generar este tipo de caracteristicas en Java.

Otra situación en la que requiere complejidad es cuando se necesita que un thread que está bloqueando un *mutex* sea el siguiente propietario de este mismo *mutex* por un número específico de veces. Este concepto no es adecuado en la definición de la *exclusión mutua*, pero tal vez se necesite realizar con la regla de un numero limite de bloqueos. A este tipo de problema podemos llamarlo ***mutex recursivo*** *porque está en un ciclo con una condición de paro(número de veces) y se retroalimenta con sus valores anteriores.* En la actualidad la sincronización sin bloqueos, junto con rendimiento y tiempo de espera son temas de investigación de los trabajos que se centran en la sincronización sin bloqueo [26], es donde se presenta API´s para el desarollo de software con concurrencia sin bloqueos, se tienen muchas ventajas al no usar bloqueos, como el problema de los deadlocks(ver sección 2.6) y en la planificación de procesos. Pero los sistemas sin bloqueo pueden también ser implementados en un sistema multiprocesador. Estos API´s contienen métodos de *libre bloqueo* (lock-free) y también en la sincronización basada en bloqueos lock-based que se usa sólo en opciones de variables.

2.5.1. Modelo abstracto de Concurrencia

El Algoritmo de Dekker no hace suposición acerca del número de procesos soportados por la máquina. El sólo supone que leer, escribir o comprobar una posición en memoria son ejecutadas autómicamente. Si son dos ejecutadas simultáneamente es no-determinista. No hacer suposición acerca de la velocidad de ejecución de un proceso(cero). Cuando un proceso esta en sección no-critica, no puede impedir a otro que entre. Un proceso que quiera entrar en sección crítica no puede ser retrazado indefinidamente.

- Utilizar una variable en turno que contiene el identificador del proceso que puede entrar a la sección crítica (1 o 2).

Algorithm 17 Proceso 1

```
1: while turno!=1 do
2:     Nothing;
3:     S.C
4:     turno=2;
5: end while
```

Algorithm 18 Proceso 2

```
1: while c1==0 do
2:     Nothing;
3:     S.C
4:     turno=1;
5: end while
```

Turno=i; antes del bucle de espera activa de cada proceso P_i.

- Asociar con cada proceso información (clave) que indique que dicho proceso entraba a la sección critica y la información es cambiada cuando el proceso sale.

Algorithm 19 Proceso 1

```
1: C1=C2=1;
2: while true do
3:     Resto;
4:     while c2==0 do
5:         Nothing;
6:     end  while
7:     C2=0;
8:     S.C
9: end while
```

Algorithm 20 Proceso 2

```
1: C1=C2=1;
2: while true do
3:     Resto;
4:     while c1==0  do
5:         Nothing;
6:     end  while
7:     C2=0;
8:     S.C
9: end while
```

Si los procesos P_1 y P_2 se ejecutan a la misma velocidad , ambos comprueban que el otro proceso no esta en la sección critica y entran ambos. Cuando sus claves es cero ya es tarde

(ya pasaron el bucle de espera ocupada).

- Un proceso puede comprobar el estado del otro antes que se modifique. La salida de un proceso de un bucle y la modificación de su clave a cero, se realiza automáticamente. Adelantar la sentencia de asignación de la clave del proceso antes del bucle de espera activa, de esta forma es imposible que un proceso pase el bucle activa con valor de clave distinto de cero.

Algorithm 21 Proceso 1

```
1: C1=C2=1;
2: while true do
3:     Resto;
4:     C1=0;
5:     while c2==0 do
6:         Nothing;
7:     end while
8:     C2=0;
9:     S.C
10:    C1=1:
11: end while
```

Algorithm 22 Proceso 2

```
1: C1=C2=1;
2: while true do
3:     Resto;
4:     C2=0;
5:     while c1==0 do
6:         Nothing;
7:     end while
8:     C2=0;
9:     S.C
10:    C2=1;
11: end while
```

Ambos tienen la misma seguridad y se produce un bloqueo y bucles de espera ocupada.

- Cuando un proceso modifica el valor de su clave, no se sabe si el otro proceso esta haciendo lo mismo.

Algorithm 23 Proceso 1

```
1: C1=C2=1;
2: while true do
3:     Resto;
4:     C1=0;
5:     while C2==0 do
6:         C1=1:
7:         while c2==0 do
8:             nothing;
9:         end while
10:        C1=1:
11:    end while
12:    S.C.
13:    C1=1;
```

14: **end while**

Algorithm 24 Proceso 2

```
1: C1=C2=1;
2: while true do
3:     Resto;
4:     C2=0;
5:     while c1==0 do
6:         C1=1:
7:         while c2==0 do
8:             nothing;
9:         end  while
10:        C2=1;
11:    end while
12:    S.C.
13:    C2=1;
14: end while
```

Las variables C_1 y C_2 pueden interpretarse como variables de estado de procesos (intentan entrar a la sección crítica) En el algoritmo de Dekker, un proceso que intenta entrar en sección critica asigna su clave a cero. Si la clave del otro proceso es cero, entonces se consulta el valor de turno y si posee el turno entonces insiste y comprueba periódicamente la clave del otro proceso. Eventualmente el otro proceso le cede el turno cambiando su clave a 1

Algorithm 25 Proceso 1

```
1: while true do
2:     Resto;
3:     C1=0;
4:     while C2==0 do
5:         if turno==2 then
6:             C1=1;
7:             while turno==2 do
8:                 nothing;
9:             end while
10:        end if
11:        C1=0:
12:    end while
13:    S.C.
14:    turno=2;
15:    C1=1;
16: end while
```

Algorithm 26 Proceso 2

```
1: while true do
2:     Resto;
3:     C2=0;
4:     while C1==0 do
5:         if turno==1 then
6:             C2=1;
7:             while turno==1 do
8:                 nothing;
9:             end while
10:        end if
11:        C2=0;
12:    end while
13:    S.C.
14:    turno=1;
15:    C2=1;
16: end while
```

2.6. Join y Barreras

Las funciones de *join* son similares a las de sincronización y casi todo lo que se puede hacer con *join* se puede hacer con *Variable de sincroniación*.

Las *barreras* permiten a un grupo de threads sincronizarse en algún punto su código. La idea es lograr que un grupo de threads se detenga cuando se llege a un punto predefinido de su cálculo y que espera a que los restantes le alcancen. Por ejemplo, si se tiene un grupo de ocho threads, el punto de inicio de la barrera será ocho. A medida que cada thread llega a dicho punto, la barrera se reduce hasta cero y todos ellos se desbloquean y prosigen. También se puden usar barreras para uno o dos threads, la diferencia es que un threads no esperara a los restantes y el thread incrementa la barrera a medida que finalize su trabajo.

2.7. DeadLock

El *deadlock* es un caso de situación paradójica en la que un thread necesita que otro thread haga algo para poder continuar y éste último necesite algo del primero. Por consiguiente

ninguno de los dos hace nada y los dos se quedan esperando al otro indefinidamente, lo que es perjudicial. El *deadlock* típico se predice cuando el T_1 tiene el bloqueo M_1 y cuando T_2 tiene el bloqueo M_2, luego T_1 necesite el bloqueo de M_2 y T_2 necesite el bloqueo de M_1 Coffman en 1971 [23] propone cuatro condiciones para que exista un bloqueo:

1. Condición de exclusión mutua: Cada recurso esta asignado a un único proceso o está disponible

2. Condición de posición y espera: Los procesos que tienen, en un momento dado, recursos asignados con anterioridad, pueden solicitar nuevos recursos.

3. Condición de no apropiación: Los recursos otorgados con anterioridad no pueden ser forzados a dejar un proceso. El proceso que los posee debe liberarlos en forma explícita.

4. Condición de espera circular: Debe existir una cadena circular de dos o más procesos, cada uno de los cuales espera un recurso poseido por el siguiente miembro de la cadena.

Holt en 1972 [20] estableció el modelo de las cuatro condiciones para el deadlock de Coffman, representado mediante grafos. Donde los rrecurso y procesos son vértices y los arcos la relación entre ellos como se muetra en la figura 2.14. En esta figura el arco de un proceso índica que el proceso está bloqeuando en espera de un recurso. La diferencia de la *solicitud de bloqueo* y la *posición del bloqueo* se da de la dirección de la arista origen y de destino como se usan los grafos dirigidos. Es decir:
Sea G=< V_1, V_2, A > un grafo con vértices V_1, vértices V_2 y arcos A. Se tienen dos proceso P_1 y P_2 con dos recurso r_1 y r_2, además de proponer a los recursos como un tipo de vértice de diferente forma(cuadrada), entonces la posición del bloqueo es: $P_1 \leftarrow r_1$ y la solicitud de bloqueo es: $P_2 \rightarrow r_2$ y por último el deadlock se puede expresar como: $P_1 \rightarrow r_2$, tal recurso r_2 está siendo ocupado por P_2, es decir, $P_2 \leftarrow r_2$ y justo en ese tiempo (exacto) $P_2 \rightarrow r_1$, pero resulta que r_1 esta siendo ocupado por P_1, es decir, $P_1 \leftarrow r_1$.

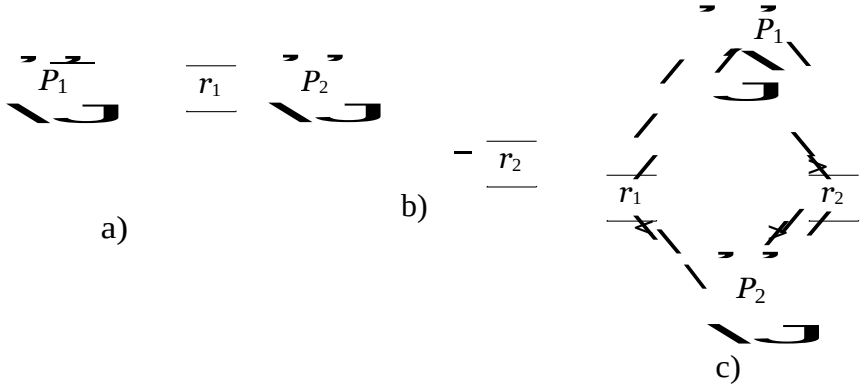


Figura 2.14: Deadlock con grafos según Holt. a) Posición, b)Solicitud y c) Deadlock

Un ejercicio interesante sería llevar al diagrama de threads los grafos de recursos y procesos de Holt, como se propone en el ejercicio 7 de este capitulo. En Coffman [23] se propone la detección de los deadocks. Un deadlock existe en un tiempo $t \therefore \exists$ un ciclo (expresado en la figura 2.14 c)) en el grafo de estados en un tiempo t. El mecanismo de detección de un deadlock, consisten en una rutina que tiene un grafo de estados en cada tiempo que el recurso es requerido, adquirido o relacionado por una tarea, y/o rutina que examina el estado del grafo para determinar donde existe un *ciclo deadlock*.

1. Simplemente ignorar todo el problema.

2. Detección y recuperación.

3. Enviarlos de forma dinámica mediante una cuidadosa asignación de recursos.

4. Prevención mediante la negación estructural de una de las cuatro condiciones necesarias.

La suma de los recuros son permitidos del tipo r_j y deben ser igual a un numero total de tipo en el sistema, es decir que los recursos deben ser soportados en el sistema. Esta suma se expresa de la siguiente manera:

$$v_j = w_j \sum_{i=1}^{n} P_{ij} \qquad (2.1)$$

Donde:

P=((p_{ij})): numero de recursos de tipo r_j permitibles a T_i

T_i: tarea i

v_j: Un elemento v $\in$ V donde V=$\{v_1, v_2,...,v_n\}$ y es el vector de recursos permisibles cual el ith elemento indica el número de elementos de tipo r_i que, asi también, son permisibles.
w_j. Un elemento w $\in$ W donde W=$\{w_1, w_2,...,w_n\}$ y representa el numero de recursos de cada tipo. El mismo Coffman en [23] propone un algoritmo para determinar la existencia de un deadlock en un tiempo t y P=Q.

Algorithm 27 Algoritmo de Coffman

Require: P,Q 1:
int w ∈ W 2:
int v ∈ V
3: Inicializar w ← V(t).
4: **for all** Q P_i(t)=0 **do**
5: se asume que todas las filas estaran sin marca en un conjunto de salida.
6: **end for**
7: i=0
8: **while** Q_i(t)=< W **do**
9: **if** fila f = 0 **then**
10: Buscar una fila sin marca
11: i++
12: **else**
13: *paso*
14: **end if**
15: **end while**
16: **while** W ← W+P_i(t) **do**
17: marcar la i-ésima fila
18: *retornaralpaso*2.
19: **end while**

1. Inicializar W ← V(t). Marcar todas la filas para cada P_i(t)=0 (se asume que todas las filas estarán sin marca en un conjunto de salida.)
2. Buscar una fila sin marca, para cada la i-ésima que Q_i(t)≤W, si uno es encontrado, entonces ir al paso 3, si no termina el algoritmo.
3. ≤El conjunto W ← W+P_i(t), marcar la i-ésima fila y retornar al paso 2.

La forma de romper un deadlock es un tema de investigación y más aun cuando el sistema es distribuido o paralelo, para la detección de deadlock existen varios algoritmos y técnicas como Redes de Petri [21] y [22], coloreado de grafos [25], entre otros.

2.8. Aplicaciones de la programación concurrente

Tanto en aplicaciones empresariales como en aplicaciones de investigación, la concurrencia es una parte fundamental en el funcionamiento de dichas aplicaciones. Sin la concurrencia las aplicaciones podrían devolver datos erroneos y perdida de información. Acontinuación se expresan algunas aplicaciones utilizando algoritmos y métodos de concurrencia

2.8.1. Terminal de autobús

Una de tantas aplicaciones de la concurrencia podría ser en un sistema de asignación de asientos en una línea de autobuses. La concurrencia se encuentra en el momento que compiten threads en la asignación de asientos, ya sea desde las casetas donde venden boletos o en las salas de espera. Así también se pueden vender para viajar por autobús. Al monento de asignar boletos, dos o más procesos podrían asignar el mismo asiento a varios usuarios, y gracias a un método de sincronización, el programador podría evitar estos problemas y poder dar un mejor servicio al usuario. Esta es una aplicacion de la concurrencia aplicada en la vida real.

Modelando el diagrama de clases del sistema de autobuses para la asignación de asientos, se modela:

- La clase *Asiento* para asignación,
- La clase *Usuario* es quien pide asientos,
- La clase *Cajero* es quien asigna los asientos,
- La clase *ThreadClienteReservadorBoletos* es el encargado para asignar boletos a un cliente en un caso de reservación.
- La clase *ThreadClienteActualizadorAreaSalidas* se encarga de la actualización de los asientos de cada terminal cada vez que se realizá una reservación o una asignación.

En el sistema se establecen dos casos:

- Reservación de boletos. En este caso un usuario puede reservar un boleto y pagarlo en un tiempo límite a la fecha de salida del autobus. La reserva podría ser por medio de internet y teléfono.
- Asignación. En el caso de que el usuario realize el pago en efectivo, tarjeta de credito o internet.

2.8.2. El peluquero dormilón

En esta aplicación se utilizarón semáforos como primitivas de sincronización además de objetos necesarios como por ejemplo:

- Silla del barbero
- Peluquer´ıa
- Sillas de espera.

Así también los procesos o threads:

- Peluquero y
- Clientes

En el barbero dormilón podemos observar como los clientes esperan a que el barbero se desocupe. Cuando el barbero logra terminar de atender a todos los clientes sentados en las sillas de espera, el barbero se pone a dormir, pues no tiene otra cosa que hacer. Se crea una estructura dinámica para simular las sillas y se implementan los metodos: insertar(), eliminar(), eslleno() y estavacio() para la administración el acceso de las *sillas de espera*. En el método estalleno() devuelve un valor lógico para indícar que no hay sillas disponibles para los clientes y el método estavacio() indica que hay sillas disponibles para clientes, además indica que el barbero se puede dormir.

Algorithm 5 Algoritmo del problema de los lectores escritores

```
Require: int readCont ;
Require: int writerCont;
Require: Semaphore mutex1, mutex2, mutex3, w, r;
 1: P(mutex3);
 2: P(r);
 3: P(mutex1);
 4: readCount++;
 5: if readCount==1 then
 6:     P(w);
 7: end if
 8: V(mutex1);
 9: V(r);
10: V(mutex 3) ;
    {leer}
11: P(mutex1);
12: readCount–;
13: if readCount==0 then
14:     V(w);
15: end if
16: V(mutex1);
Require: int readCont ; writerCont;
Require: Semaphore mutex1, mutex2, mutex3, w, r ;
17: P(mutex 2);
18: writeCount++;
19: if writeCount==1 then
20:     P(r);
21:  end if
22: V(mutex2);
23: P(w); {escribir}
24: V(w) ;
25: P(mutex2);
26: writeCount–;
27: if writeCount==0 then
28:     V(r);
29:  end if
30: V(mutex2);
```

Algorithm 6 Algoritmo problema de los fumadores

```
1: while true do
2:     wait(T); {escoger un fumador i y j no-deterministico}{se toma al tercer fumador k}
3:     signal(A[k]);
4: end while
5: while true do
6:     wait(A[i]); {tomar un  cigarrillo}
7:     signal(T); {fumar el  cigarrillo}
8: end while
```

Algorithm 7 Mutex

Require: boolean pase;

```
1: Mutex(boolean pase) {
2: this.pase=pase; }
3: synchronized void lock(){
4:   while pase do
5:     try  {wait();
6:     catch(InterruptedException e  ){
7:     print (ERROR; + e.toString());}}
8: end while
9: pase=true;}
```

Algorithm 8 Mutex continuación

```
1: synchronized void unlock(){
2: pase=false;
3: notify();}
4: synchronized boolean obten pase(){return pase;}
5: synchronized void pon pase(boolean pase){pase=pase;}}
```

Algorithm 11 Semáforo

Require: boolean V;

```
1: SemaforoBinario(boolean int V){
2: V=intiV; }
3: synchronized void P(){
4: while V==false do
5:     myWait(this);
6:     V=false;
7: end while
8: }
9: synchronized void V(){
10:  V=false;
11:  notify();}
12: static void myWait(Object obj){
13: try{obj.wait();}
14: catch(InterruptedException e){}
15: }
```

Algorithm 12 Semáforo Contador

Require: boolean V;

```
1: SemaforoContador(boolean int V){
2: V=intiV;}
3: synchronized void P(){
4: V–
5: if V<0 then
6:     myWait(this);
7: end if
8: }
9: synchronized void V(){
10: V++;
11: if V<=0 then
12:     notify();
13: end if
14: }
15: static void myWait(Object obj){
16: try{obj.wait();}
17: catch(InterruptedException e){ }}
```

Algorithm 15 La clase Semáforo con wait y notify

```
1: synchronized void UP(){
2: if esperando > 0 then
3:     notify();
4: else
5:     valor++;
6: end if
7: }
Require: int valor;
Require: int esperando = 0;
8: public Semaforo (int i){
9: valor = i  ;}
10: synchronized void DOWN(){
11: if valor > 0 then
12:     valor–;
13: else
14:     esperando++;
15:     try {wait();}
16:     catch (InterruptedException e){};
17:     esperando–;
18: end if
19: }
```

Algorithm 16 Clase VC

```
Require: mutex m;
1: VC(mutex m){
2: this.m=m;}
3: void espera(mutex m){
4: try{synchronized(this){
5: m.unlock();
6: wait();}}
7: catch(InterruptedException e){}
8: finally {m.lock();}}
Ensure: synchronized void senal(mutex m){
9:  notify();}
```

Capítulo 3

Procesos Centralizados y Distribuidos

En la concurrencia como en el paralelismo la comunicación entre dos o más CPU´s es fundamental, a pesar de que por definición la concurrencia no necesita más que un CPU, las aplicaciones de hoy en día y en un futuro son y serán de forma remota, basadas rede Web e inalámbricas. El principio de toda aplicación se remonta a una CPU, luego en dos o más CPU´s hasta llegar a cluster y más. En este capítulo se exponen tres métodos de comunicación remota los cuales son: SOCKETS, RMI[1] y CORBA[2]. Además de tres importantes paradigmas de comunicación entre computadoras, por ejemplo: Maestro/esclavo, cliente/servidor y peer to peer (participante/participante).

3.1. Maestro-Esclavo

Este paradigma esta orientado a una comunicación básica entre dos o más computadoras, se puede decir que es la forma primitiva en que se comunican dos computadoras. La filosofía del Maestro-Esclavo es muy semejante al de Humano-Computadora. El funcionamiento del Maestro-Esclavo consiste enviar datos a los esclavos, luego de establecer la relación Maestro-Esclavo y la jerarquia de la dirección del control del programa. La única comunicación que se tienen entre maestro y esclavo es enviar los resultados de la tarea asignada, comúnmente no existen dependencias fuertes entre las tareas realizadas por los esclavos (poca o nula comunicación entre esclavos). La sincronización y la asincronización se realiza de forma sencilla pero requiere programar el mecanismo de lanzado de tareas, la distribución de datos, el control del maestro sobre los esclavos y la sincronización en caso de ser necesaria.
La aplicaciones típicas de este paradigma son:

- Técnicas de simulación Monte Carlo.
- Aplicaciones criptográficas.

[1]Remote Method Invocation
[2]Common Object Request Broker Architecture

- Algoritmos de optimización (basados en poblaciones).
- Modelos de partición sencilla de tareas y datos.

Otros modelos del paradigma Maestro-Esclavo como:

- Los Modelos fork-join y FORK L, generan dos ejecuciones concurrentes en un programa. Una se inicia en la instrucción FORK y otra empieza en la instrucción etiquetada L. JOIN permite recombinar varias ejecuciones paralelas en una sola. La rama que ejecuta primero la instrucción JOIN termina su ejecución. Para saber el número de ramas que se deben reunir se usa un parámetro con JOIN (una variable entera no negativa que se inicializa con el número de ejecuciones paralelas a reunir). La instrucción JOIN tiene que ejecutarse *indivisiblemente* es decir, la ejecución concurrente de dos instrucciones JOIN es equivalente a la ejecución secuencial en un orden indeterminado.

La forma general o algoritmo del Maestro-Esclavo es:

Algorithm 28 Maestro

```
1: (num esclavos);
2: for i=0; i=num esclavos do
3:     datos=Determinar datos(i);
4:     Lanzar tarea(i, datos);
5: end for
6: respuestas=0;
7: for i=0; i=num esclavos do
8:     res=Obtener Respuesta();
9:     resultado=Procesar resultado(res);
10: end for
11: Desplegar resultado(resultado);
```

Algorithm 29 Esclavo

```
1: datos=Esperar datos(master);
2: Procesar(datos); {No hay comunicación entre procesos esclavos }
3: Enviar Resultado(master);
```

3.2. Cliente-Servidor

El paradigma cliente-servidor nos proporciona la forma fácil de comunicar dos computadoras, con la restricción de que dos clientes no podrán comunicarse entre ellos. Es decir, se da el nombre de servidor a una computadora y el nombre de cliente a otra, y estas dos se comunican por medio de mensajes. El cliente realiza peticiones de recursos de computo al servidor y el servidor atiende estas peticiones, la restricción se encuentra en que un cliente no puede hacer peticiones de recursos a otro cliente, la única forma de comunicación entre dos cliente es por medio del servidor.

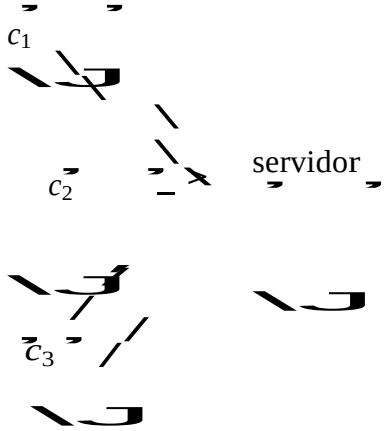


Figura 3.1: Cliente-servidor

En la figura 3.1 se muestra el flujo de comunicación entre varios clientes y un servidor y las respectivas restricciones que se tiene un el paradigma cliente-servidor.

Para que el servidor puede atender todas las peticiones de los clientes, se crean procesos o threads para cada uno de estos clientes conectados. Esto se realiza para que el CPU por completo, que no tenga que resolver cada una de las peticiones de recursos o servicios que los clientes realizan con frecuencia, dado una CPU abstracta o conceptualizada por cada uno de los clientes hace que se máximice el cómputo y mínimice el tiempo de respuesta de recursos o peticiones. En la figura 3.2 se ilustra la forma conceptual en que un servidor puede atender a varios clientes conectados y luego detectados por procesos o threads por medio de un canal de comunicación bidireccional. Este concepto es importante y básico en el paradigma Cliente-Servidor por lo mismo que el servidor.

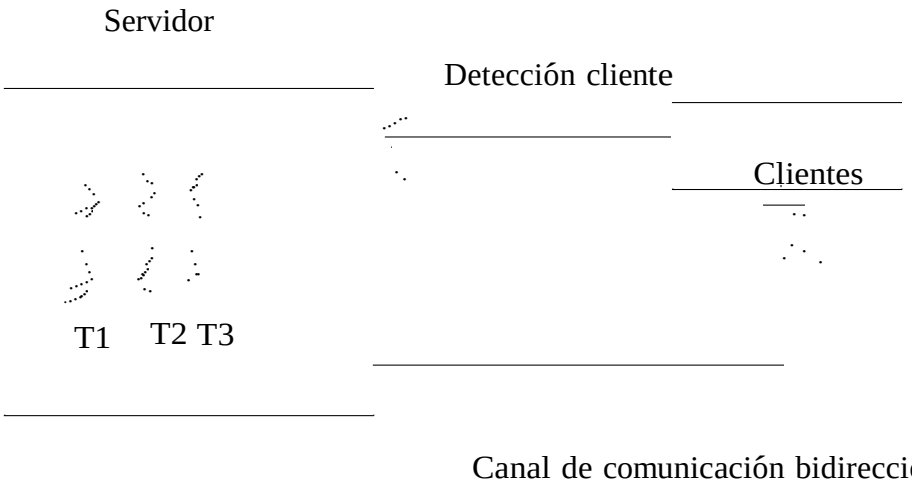


Figura 3.2: Un proceso o thread para cada cliente

Otra forma de expresar el paradigma Cliente-Servidor es estructurando un sistema operativo como un grupo de procesos cooperativos, llamados servidores, que ofrecen servicios a los usuarios llamados clientes. El servidor y el cliente normalmente se ejecutan sobre el mismo kernel o microkernel en máquinas diferentes. El cliente y servidor se basan en un protocolo sin conexión de solicitud/respuesta.

Las ventajas del paradigma Cliente-Servidor en primer lugar, está basado en un sistema simple y eficiente, sostenidos en las tres primeras capas del modelo OSI [3]. Proponiendo dos primitivas *send* y *receive* de la manera siguiente:

- send(dest,menssPtr)
- receive(addr,menssPtr)

Donde *dest* es igual a destino, *addr* la dirección quién envío y *menssPtr* el mensaje como un apuntador de memoria. Un cliente debe conocer la dirección del servidor al que le desea enviar un mensaje. La estrategia para realizar una comunicacion entre cliente y servidor es:

1. Especificación de la máquina. Válido para un proceso en la máquina.
2. Especificar la máquina y el proceso. Las máquinas inician la numeración de sus procesos a partir de cero(no es transparente).
3. Especificar la máquina y un identificador local entero aleatorio de 16 a 22 bits.
4. Especificar el proceso. Definiendo un proceso centralizado que asigne direcciones únicas a los procesos que lo soliciten(no conveniente para grandes sistemas).

[3]Apéndice 8.12

5. Permitir que los procesos seleccionen aleatoriamente su direccionamiento grande.

El modelo de comunicación que identifica dos clases de procesos, como pueden ser una clase *cliente* que solicita servicios a procesos de otra clase *servidor*, que atiende los pedidos y puede ser utilizada para comunicar procesos que se ejecutan en un único equipo, a la vez es una idea potencialmente utilizada para comunicar procesos distribuidos en una red. El modelo Cliente-Servidor provee un mecánismo para comunicar aplicaciones remotas (que funcionen simultáneamente como clientes y servidores para diferentes servicios, convenientemente de acuerdo a las caracter´ısticas de la red.

3.3. Peer-to-Peer

El paradigma *peer-to-peer (P2P)* es abierto con respecto a las restricciones que tienen los paradigmas Maestro-Esclavo y Cliente-Servidor. En el paradigma p2p se puede comunicar un cliente con otro cliente y así también se pueden pedir recursos entre estos, como se muestra en la figura 3.3, donde *c/s* puede ser un cliente-servidor al mismo tiempo o también conocidos como *peer*. La nomeclatura *co* es un posible coordinador entre los *peers*, el cual representa el nodo que contiene la información de entrada y salida de los demás nodos en la red, y cada vez que se realiza algún cambio éste notifica a los demás peers. como se expresa en la figura 3.3, un grafo dirigido representa una red p2p por que el flujo de datos es bidireccional en cada uno de los peers.

Una red informática peer-to-peer anónima es un tipo particular de red en la que los usuarios y sus nodos son pseudoanónimos por defecto. La principal diferencia entre las redes habituales y las anónimas se encuentran en el método de encaminamiento de las respectivas arquitecturas de redes. Estas redes permiten el flujo libre de información. Y el interés de la comunidad p2p en el anonimato ha incrementado muy rápidamente desde hace unos años por varias razones, entre ellas se encuentra la desconfianza en todos los gobiernos y los permisos digitales. Tales redes suelen ser utilizadas por aquellos que comparten ficheros musicales con copyright. Muchas asociaciones referentes a la defensa de los derechos de autor han amenazado con demandar a algunos usuarios de redes p2p no anónimas.

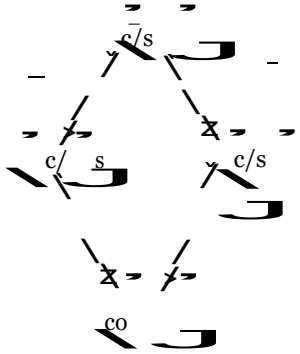


Figura 3.3: Peer-To-Peer

El p2p se basa principalmente en la filosofía y los ideales, que señala que *"todos los usuarios deben compartir recursos"*. Conocida como filosofía p2p, es aplicada en algunas redes en forma de un sistema enteramente meritocrítico en donde ."el que más comparta, más privilegios tiene y más acceso dispone de manera más rápida y a más contenido". Con este sistema se pretende asegurar la disponibilidad del contenido compartido, ya que de lo contrario no sería posible la subsistencia de la red.

Características deseables de las redes p2p:

- Escalabilidad. Las redes p2p tienen un alcance mundial con cientos de millones de usuarios potenciales. En general, lo deseable es que cuantos más nodos están conectados a una red p2p mejor sería su funcionamiento. Así, cuando los nodos llegan y comparten sus propios recursos. Esto es diferente en una arquitectura del modo cliente-servidor con un sistema fijo de servidores, en los cuales la adición de más clientes podría significar una transferencia de datos más lenta para todos los usuarios. Algunos autores advierten de que si proliferan mucho este tipo de redes podrían llegar a su fin, ya que a cada una de estas redes se conectarán muy pocos usuarios.

- Robustez. La naturaleza distribuida de las redes peer-to-peer también incrementa la robustez en caso de haber fallos en la réplica excesiva de los datos hacia múltiples destinos, y en sistemas p2p puros, permitiendo a los peers encontrar la información sin hacer peticiones a ningún servidor centralizado de indexado. En el último caso, no hay ningún punto singular de falla del sistema.

- Descentralización. Estas redes por definición son descentralizadas y todos los nodos son iguales. No existen nodos con funciones especiales, y por tanto ningún nodo es imprescindible para el funcionamiento de la red.

- Costes. Los costes están repartidos entre los usuarios. Se comparten o donan recursos a cambio de recursos, según la aplicación de la red, los recursos pueden ser archivos, ancho de banda, ciclos de proceso o almacenamiento de disco.

- Anonimato. Es deseable que en estas redes quede anónimo el autor de un contenido, el editor, el lector, el servidor que lo alberga y la petición para encontrarlo siempre que así lo necesiten los usuarios. Muchas veces el derecho al anonimato y los derechos de autor

son incompatibles entre sí, y la industria propone mecanismos como el DRM[4] para limitar ambos.

- Seguridad. Es una de las características deseables de las redes p2p menos implementadas. Los objetivos de un p2p seguro serían identificar y evitar los nodos maliciosos, evitar el contenido infectado, evitar el espionaje de las comunicaciones entre nodos, creación de grupos seguros de nodos dentro de la red, protección de los recursos de la red, entre otras. En su mayoría aún están bajo investigación, pero los mecanismos más prometedores son:
  - Cifrado multiclave,
  - Cajas de arena,
  - Gestión de derechos de autor (la industria define qué puede hacer el usuario, por ejemplo la segunda vez que se oye la canción se apaga)
  - Reputación (sí se permitir el acceso)
  - Comunicaciones seguras
  - Comentarios sobre los ficheros, etc.

3.3.1. Arquitectura Peer to Peer

Peer-to-peer es una arquitectura donde los recursos y los servicios son directamente intercambiados entre las computadoras, estos recursos y servicios incluyen el intercambio de información, los ciclos de procesamiento, almacenamiento en caché y el almacenamiento en disco para archivos; así también el tipo de arquitectura, los equipos que han sido tradicionalmente utilizados únicamente como los clientes se comunican directamente entre sí y pueden actuar como clientes y servidores, asumiendo cualquier papel que resulte más eficiente para la red. En el paradigma peer-to-peer, los procesos participantes juegan un papel igual, con las capacida- des y responsabilidades equivalentes (de ahí el término ”peer”).Cada participante podrá enviar una petición a otro participante y recibir una respuesta, mientras que el paradigma cliente-servidor es un modelo ideal para un servicio centralizado de la red, el paradigma peer-to-peer es más apropiado para aplicaciones como la mensajería instantánea, transferencias de archivos, video conferencia y el trabajo colaborativo. Por ejemplo un servicio de intercambio de archivos peer-to-peer de transferencia, como lo fue *Napster.com*, entre varios sitios similares que permiten que los archivos (archivos de audio sobre todo) se transmite entre las computadoras conectadas a Internet y puedan ser descargados sin derechos de autor.

[4]Gestión Digital de Restricciones

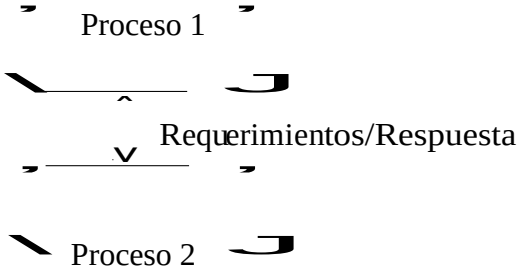


Figura 3.4: Arquitectura Peer-To-Peer

3.4. Sistemas Distribuidos

Anteriormente la computación fue concebida mediante un sólo procesador. La computación con una CPU llamada servidos, puede llamarse computación centralizada y varias computadoras que comparten obligaciones y recursos se llama computación distribuida

Sistema Distribuido.

Es una colección de computadoras independientes conectadas vía Red capaces de realizar trabajos colaborativamente. La computación distribuida es la computación que se lleva a cabo en un sistema distribuido, cuyo objetivo es el compartir recursos y llevar un control de usuarios y anónimos..

3.4.1. Ejemplos de sistemas distribuidos

- Red de estaciones de trabajo (NOW): Un grupo de estaciones de trabajo en red conectadas a una o más máquinas de servidores.

- Una Intranet. Una red de computadoras y estaciones de trabajo dentro de una organización, separados de la Internet a través de un dispositivo de protección (un firewall).

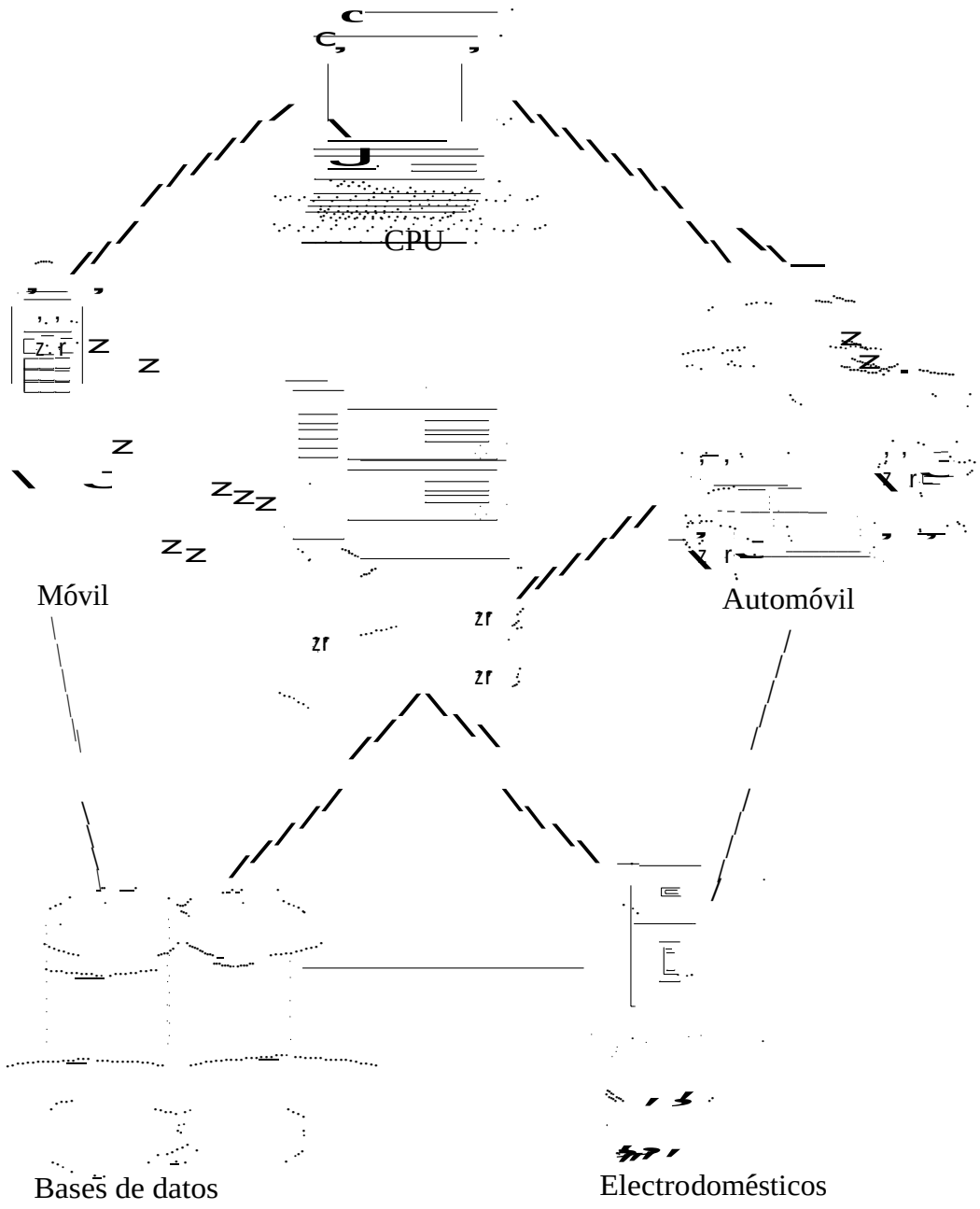


Figura 3.5: Ambiente distribuido

-
-

3.4.2. Computadoras en un sistema distribuido

- Workstations: computadoras utilizadas por los usuarios finales para llevar a cabo el
- computo.

Server: Computadoras que proporcionan recursos y servicios.

Personal Assistance Devices(PDA): computadoras portátiles conectados a la red a través de un enlace de comunicación inalámbrica.

Dispositivos móviles.

-
-
-

3.4.3. Aplicaciones monolíticas centrales vs Aplicaciones distribuidas

Arquitectura de las aplicaciones monol´ıticas:

- Independiente, de una sola función en la aplicación, tales como el orden de entrada o de facturación.
- Las aplicaciones no pueden compartir los datos u otros recursos.
- Los desarrolladores deben crear varias instancias de la misma funcionalidad (servicios).

Arquitectura de las aplicaciones distribuidas

- Aplicaciones integradas.
- Las aplicaciones pueden compartir recursos.
- Una sola instancia de la funcionalidad (servicios) pueden ser reutilizados.
- Interfaces de usuario común.

Evolución de los paradigmas

- Cliente-Servidor. Este paradigma se puede implementar con métodos como: Sockets, métodos de invocación remota, entre otros.
- Objetos Distribuidos
- Servicios de Red : Jini
- Espacio de Objetos: Java Spaces

  Agentes móviles

  Mensajes orientados a middleware (MOM):Java MessageService

  Aplicaciones colaborativas.

¿Por qué computación distribuida?

Economía: Los sistemas distribuidos permiten el uso común de recursos, incluidos los ciclos de CPU, almacenamiento de datos de entrada / salida, y los servicios.

Fiabilidad: Un sistema distribuido permite la replicación de los recursos y / o servicios, reduciendo así la interrupción del servicio debido a fallas. La Internet se ha convertido en una plataforma universal para la informática distribuida.

Debilidades y fortalezas de la computación distribuida

En cualquier forma de computo, siempre hay ventajas y desventajas, algunas por razones de popularidad del computación distribuido son:

- La accesibilidad de las computadoras y la disponibilidad de acceso a la red.
- Distribución de recursos.
- Escalabilidad.
- Tolerancia a fallas.

Las desventajas de la computación distribuida:

- **Múltiples puntos de error:** Una o más computadoras o uno o más enlaces de red.
- **Seguridad:** En un sistema distribuido existen más oportunidades de un acceso no autorizado.

3.5. Sockets

3.5.1. Protocolos

Un **protocolo** es un conjunto de reglas, formato de datos y convenciones que es necesario respetar para conseguir la comunicación entre dos entidades, ejemplos: IP, TCP, UDP, FTP, etc.[5], Cuando se crea un socket es necesario indicar cual es el protocolo de trasporte a emplear para el intercambio de datos que se realizará a través de él.

Es un mecanismo que comunica a más de dos procesos, que pueden intercambiar diferente tipos de datos ya sea que los procesos estén corriendo en un computador o en una red de computadoras, además guarda cierta similitud con las tuberías (pipe). Para que la comunicación se establezca entre los procesos son necesarios los siguientes aspectos:

- Dirección IP
- Protocolo
- Número de puerto.

Para conseguir ésto, tomamos el concepto de **conector o socket**; dos procesos distintos crean cada uno su conector por lo tanto.

[5]IP Protocolo de Internet, TCP Protocolo de Control de Transmición, UDP Protocolo de datagramas de usuario, FTP Protocolo de Transferencia de Archivos.

1. Cada conector está ligado a una **dirección**.

2. Un proceso puede enviar información, a través de un socket propio, al socket de otro proceso, siempre y cuando conozca la dirección asociada a otro socket.

3. La comunicación se realiza entre una pareja de sockets.

3.5.2. Dominios y direcciones

Definimos un dominio de comunicación como una familia de protocolos que se pueden emplear para conseguir el intercambio de datos entre sockets. Un sistema **UNIX** particular puede ofertar varios dominios, aunque los más habituales son estos dos:

1. Dominio UNIX

$$(PF\_UNIX)$$

   Para comunicarse entre procesos dentro de la misma máquina.

2. Dominio Internet **(PF INET)**[6]. Para comunicarse entre procesos en dos computadoras conectadas mediante los protocolos de Internet (TCP, UDP, IP).

Además un socket del dominio PF_UNIX no puede dialogar con sockets del dominio PF_INET. Cada familia de protocolos define una serie de convenciones a la hora de especificar los formatos de las direcciones. Tenemos, por lo tanto, estas dos familias de direcciones:

1. **Formato**

$$UNIX(AF\_UNIX)$$

   Una dirección de socket es como los nombres de los ficheros(pathname).

2. **Formato Internet(AF_INET)** [7]**.**. Una dirección de sockets precisa de estos tres campos:

   - Una dirección de red de la máquina (Dirección IP).
   - Un protocolo (TCP o UDP) y
   - Un puerto correspondiente a ese protocolo.

Es importante resaltar que un socket puede ser referenciado desde el exterior sólo si su dirección es conocida. Sin embargo, se puede utilizar un socket local, aunque no se conozca la dirección que tiene.

[6] PF Protocol Family
[7] AF Address Family.

3.6. Estilos de comunicación

3.6.1. Orientados a conexión

Mecanismo que sirve para conseguir un canal para el intercambio de octenos. Un proceso pone a su ritmo bytes en el canal, que recibe de forma fiable y ordenada en el otro extremo donde hay un proceso que los recoge a su conveniencia.
Ejemplo: tuberias y socketpairs

3.6.2. Comunicación sin conexión

También denominada comunicación con datagramas. El emisor envía mensajes, un mensaje autónomo es aquel que el receptor debe recibir entero por algún camino. Los mensajes pueden perderse en el camino, retrasarse o llegar desordenados. La comunicación entre dos sockets puede realizarse con cualquiera de estas dos formas (aunque los dos sockets que se comunican entre ellos deben de crearse con el mismo estilo, además de el mismo dominio). Para ello, al crear un socket se indica el estilo de comunicación correspondiente:

3.6.3. SOCK_STREAM

La comunicación pasa por tres fases:

1. Apertura de conexión
2. Intercambio de datos
3. Cierre de conexión

El intercambio de datos es fiable y orientado a octenos (como los pipes): no hay frontera de mensajes

3.6.4. SOCK_DGRAM

. No hay conexiones. Cada mensaje (o datagrama) es autocontenido. Los datagramas no se mezclan unos con otros. No hay garant´ıa de entrega: se pueden perder, estropear o desordenar. De forma gráfica se puede describir la conexión entre cliente- servidor utilizando un socket como se muestra a continuación.

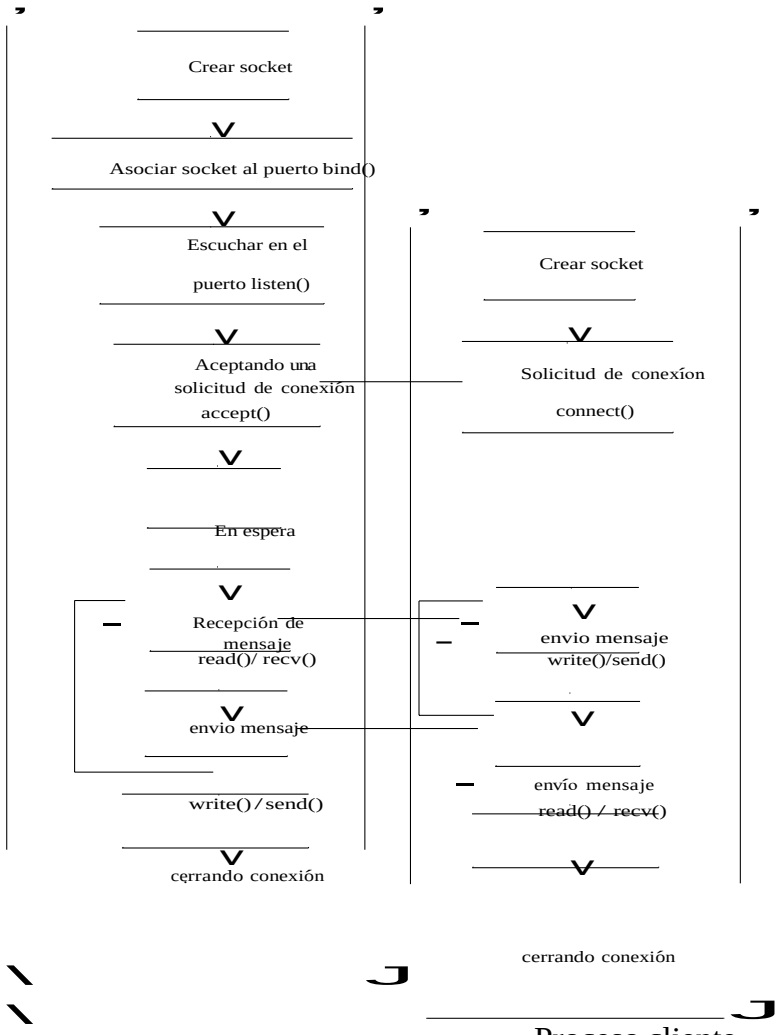


Figura 3.6: Modo conexión

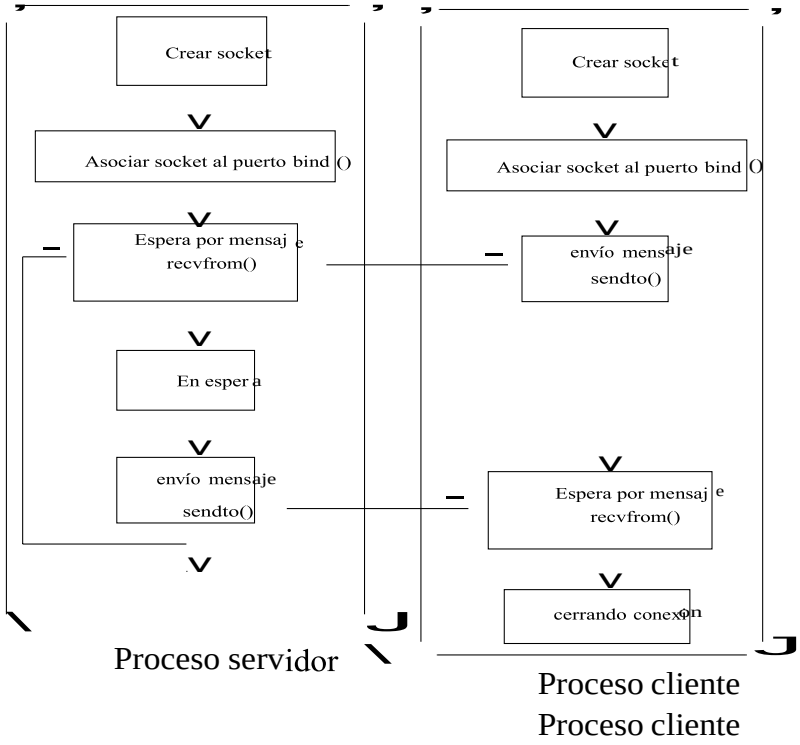


Figura 3.7: Modo no conexión

El Socket es un concepto abstracto por el cual dos programas (posiblemente situados en CPU´s distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. Los sockets son una forma de comunicación de computadoras que funcionan como si se conectara una computadora a otra por medio de un cable. Los sockets se dividen en dos tipos:

- UDP (User Datagram Protocol)
- TCP (Transmisión Control Protocol)

El Socket UDP transmite paquetes individuales de información. Los Sockets UDP son muy diferentes a los TCP, pues los UDP dan servicio sin conexión, pero no garantizan que los paquetes llegan en alguna forma en particular. El problema de los UDP es que los paquetes pueden perderse, duplicarse o llegar en desorden, es por eso que se requiere una programación adicional significativa para la solución de este problema. Los servicios sin conexión generalmente ofrecen mayor velocidad, pero menor confiabilidad que los servicios orientados a conexion.
Los API´s Socket en Java contiene dos tipos de clases para los UDP:

- DatagramSocket. Intercambio de datos
- DatagramPacket. Intercambio de datagramas

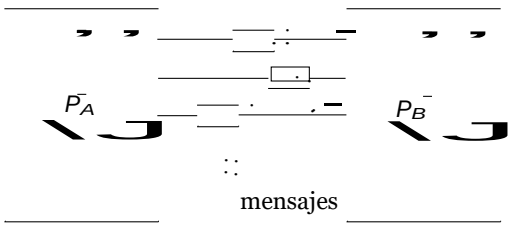


Figura 3.8: Comunicación remota con Sockets datagrama sin conexión

El socket TCP o también llamados socket de flujo, emplean un proceso que establece una conexión de un proceso P_A a un proceso P_B u otros procesos y está comunicación se sostiene mientras exista la misma conexión, así los datos fluyen entre los procesos en un flujo continuo proporcionando un servicio orientado a conexión, empleando un protocolo para la transmisión popular. El flujo donde pasan los mensajes en una comunicación con sockets datagrama orientado a conexión es mediante un camino virtual y los mensajes son estrictamente enviados de forma sincronizada y ordenada como se muestra en la figura 3.9.

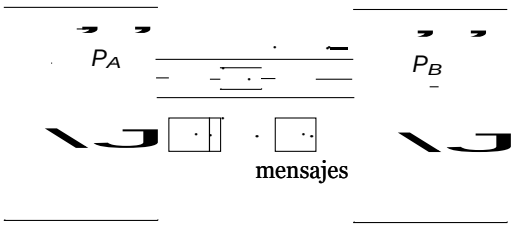


Figura 3.9: Comunicación remota con Sockets datagrama orientada a conexión

Creación de socket

Un proceso puede crear un socket utilizando la función *socket()*.
Su prototipo es:

```
#include <sys/types.h>
#include <sys/socket.h>
Int socket(int domain, int type, int protocolo);
```

Domain

- PF_UNIX
- PF_INET

(3.1)

Type socket

- $SOCK\_STREAM$
- $SOCK\_DGRAM$
- $SOCK_RAW(root)$

(3.2)

Un protocolo sirve para especificar la comunicación a emplear, en general este argumento es 0, indicando que se usa el protocolo por omisión.

3.6.5. Llamadas de sistema para la creación y uso de sockets

Creación del socket

El primer paso es crear un socket, para realizar esta tarea tenemos la llamanda del sistema *socket()*, y su s´intaxis es la siguiente:

```
int socket(int domain, int type_socket, int protocolo);
```

Asociar el socket a un puerto

Ya creado el socket, es necesario asociarlo a un puerto para que el proceso inicie la tarea de escuchar a través de dicho puerto, y para cumplir con esta tarea tenemos la llamada del sistema *bind( )*:

```
int bind(int sockfd, const strcuct sockaddr *address, size_t address_len );
```

Escuchar en el puerto

Como sabemos el uso de socket se da dentro de un contexto maestro-esclavo, entonces esta llamada del sistema es usada por el servidor en protocolos orientados a conexión, después de asociar el socket a un puerto es necesario ponerse a escuchar a través del socket, la llamada al sistema para escuchar es *listen( )*.

```
int    listen(int sockfd, int backbg);
```

Aceptando una llamada

Los protocolos orientados a conexión el servidor debe de esperar una solicitud de conexión, para después crear una vía de comunicación, la cual realiza una función de tubería bidireccional (pipe), esta tarea la realiza la llamada al sistema *accept( )*.

```
int accept(int sockfd, struct sockaddr *cltaddr, int *addrlen);
```

La función *accept( )* es una llamada bloqueante, es decir el proceso se bloquea hasta que llegue una petición de conexión.

Conexión al puerto

Las dos últimas llamadas descritas (listen, accept) son usadas por el servidor para obtener solicitudes de conexión y crear dichas vías, por otro lado el cliente cuenta con la llamada *connect ( )* para solicitar una conexión.

```
int    connect(int sockfd, struct sockaddr *servaddr, int *addrlen);
```

Recepción de mensajes

Existen tres llamadas que se pueden utilizar para la repetición de mensajes y son las siguientes:

```
int read (int fildes, void *buf, size_t nbyte);
int  recv (int  sockfd,   const void *buffer,      size_t length, int flags);
ssize_t   recvfrom(int sockfd, void *buffer, size_t length, int flags, struct sockadd
```

- *read( )* es utilizada para lectura de archivos.
- *recv( )* lee un mensaje
- *ssize_t recvfrom ( )* espera por un mensaje por sockfd

Donde:

sockfd	Es el socket creado con socket(..)
address	Tiene la información del emisor.
address_len	Si address_len y address tiene valor cero, entonces se comportan como recv( ).
backbg	Es el número máximo de procesos en espera, en otras palabras podemos decir que es la longitud de la cola de espera,usualmente este parámetro tiene valor de cinco.
cltaddr	Es la dirección del proceso con el que se establece la via de comunicación (cliente)
addrlen	Es la longitud de la dirección .
servaddr	Es la dirección del proceso con el que se estableció la via de comunicación (servidor)
fildes	Se lee del archivo al cual hace referencia fildes
buf	La lectura es almacenada en el búfer
nbytes	Byte de información leída
buffer	Almacena lo leído en búfer.
length	Longitud del mensaje a leer.
flags	Es utilizado para protocolos especiales, pero en nuestro caso lo pondremos en cero.

Si todos los datos son proporcionados correctamente, las tres funciones regresarán el número de bytes recibidos, en caso contrario -1 si hubo error. La selección del uso de estas llamadas depende del protocolo: conexión y no conexión, las dos *read( ) y recv( )* son usadas en protocolos tipo conexión.

Enviando mensajes

Existen tres llamadas que se pueden utilizar para el envió de mensajes y son las siguientes:

```
int write (int fildes, void *buf, size_t nbyte);
int send (int sockfd, const void *buffer, size_t length, int flags);
ssize_t sendto(int sockfd, void *buffer, size_t length,int flags,struct sockaddr *addre
```

- *read( )* es utilizada para lectura de archivos.
- *send( )* envia un mensaje

  ssize t sendto ( ) envia el mensaje por sockfd

Donde:

fildes	Se lee del archivo al cual hace referencia fildes
buf	La lectura es almacenada en este búfer
nbytes	Byte de información leída
sockfd	Socket del cual se va a leer.
buffer	Almacena lo leído en búfer.
length	Longitud del mensaje a leer.
flags	Es utilizado para protocolos especiales, pero en nuestro caso lo pondremos en cero.
length	Longitud del mensaje a leer.
flags	Es utilizado para protocolos especiales, pero en nuestro caso lo pondremos en cero.
address	Tiene la información del emisor
address_len	Si address len y address tiene valor cero, entonces se comportan como recv( ).

Comunicación con socketpairs

Este mecanismo es similar a las tuberías, sólo con la particularidad de que son bidireccionales y son usados cuando los procesos tienen un ancestro en común. Para crear un socketpairs se hace de la siguiente manera:

```
#include <sys/types.h>
#include <sys/socket.h>
Int socketpair(int domain, int type, int protocolo, int vecsock[2]
```

Las tareas que se deben realizar se muestran en la siguiente tabla.

PADRE	HIJO
Crear un socketpair	
Crear el hijo	
Leer un mensaje	Escribir un mensaje
Escribir un mensaje	Leer un mensaje
Terminar	Terminar

Tabla 3.1: Entradas y salidas de un socketpair

Algorithm 30 Conexión de sockets usando procesos

```
1: if (child=fork())==-1 then
2:     perror(çreando el proceso hijo");
       {este es el padre}
3:     close(sockets[0]);
4:     if read(sockets[1],buf,1024)<0 then
5:         perror("padre leyendo el mensaje  ");
6:     end if
7:     print  (ÏD padre = %d mensaje de mi hijo –> %s ",getpid(),buf);
8:     if write(sockets[1],men2,strlen(men2)+1)<0 then
9:         perror("padre escribiendo el mensaje");
10:    end if
11:    print  (ÏD padre = %d respuesta –> %s ",getpid(),men2);
12:    close(sockets[1]);
13:    exit(1);
       {este es el  hijo}
14:    close(sockets[1]);
15:    print  (ÏD hijo = %d pregunta –> %s",getpid(),men1);
16:    if write(sockets[0],men1,strlen(men1)+1)<0 then
17:        perror("hijo escribiendo  mensaje");
18:    end if
19:    wait(&n);
20:    if read(sockets[0],buf,1024)<0 then
21:        perror("hijo leyedo  mensaje");
22:    end if
23:    print  (ÏD hijo = %d respuesa de papa –> %s",getpid(),buf);
24:    close(sockets[0]);
25: end if
```

Algorithm 31 Ejemplo sobre datagramas en el dominio PF_UNIX

```
    #define NAME ”socketunix”
 2: int sock, length, lon2;
    struct sockaddr un name;
 4: char buf[1024];
    sock=socket(PF UNIX, SOCK DGRAM,0);
 6: if sock<0 then
        perror(.ªbriendo socket de dtagramas”);
 8:     exit(1);
    end if
10: name.sun family=AF UNIX;
    strcpy(name.sun _path,NAME);
12: if bind(sock,(struct sockaddr )&name,sizeof(name))<0 then
        perror(.ªsociado con nombre al  socket”);
14:     exit(1);
    end if
16: print (”Direccion del socket → %s ”,NAME);
    if recvfrom(sock,buf,1024,0,NULL,&lon2)<0 then
18:     perror(recibiendo un  datagrama”);
    end if
20: print   (ÏD proceso %d →%s”,getpid(),buf);
    close(sock);
22: unlink(NAME);
    exit(0);}
```

Algorithm 32 Ejemplo sobre datagrama en el dominio PF_UNIX

```
1: #define DATA .emisor esta ahi"
2: int sock;
3: struct sockaddr un name;
4: sock=socket(PF UNIX, SOCK DGRAM,0);
5: if sock<0 then
6:     perror(.Abriendo scoket de  datagrama");
7:     exit(1);
8: end if
9: name.sun family=AF UNIX;
10: strcpy(name.sun path,argv[1]);
11:  print  (ÏD proceso %d pregunta: →%s",getpid(),DATA);
12: if sendto(sock,DATA,strlen(DATA)+1,0,(struct sockaddr )&name,sizeof name)<0 then
13:     perror(.enviando un  datagrama");
14:     close(sock);
15:     exit(0);
16: end if
```

Algorithm 33 Serveco3.c

Require: extern int errno;

Require: struct sockaddr_in sin,fsin;

Require: int s,ssock,alen;

```
 1: sin.sin family=AF  INET;
 2: sin.sin addr.s addr =htonl(INADDR ANY);
 3: sin.sin port  =htons(4000);
 4: if (s=socket(PF INET,SOCK STREAM,0))<0  then
 5:     perror("No se puede crear el socket");
 6:     exit(1);
 7: end if
 8: if bind (s,(struct sockaddr )&sin,sizeof(sin))<0 then
 9:     perror("No se puede asignar la dirección");
10:     exit(2);
11: end if
12: if listen(s,5)<0 then
13:     perror("No puedo poner el socket en modo escucha");
14:     exit(3);
15: end if
16: signal(SIGCHLD, SIG IGN);
```

Algorithm 34 continuación Serveco3.c

```
1: while 1 do
2:    alen=sizeof(fsin);
3:    if (ssock =accept(s, (struct sockaddr )&fsin,&alen))<0  then
4:       if errno == EINTR then
5:          continue;
6:       end if
7:       perror(”Fallo en función accept”);
8:       exit(4);
9:    end if
10: end while
11: switch(fork())
12: case 1: perror(”No puedo crear hijo”);
13: exit(5);
14: case 0 :  close(s);
15: Proceso hijo
16: do_echo(ssock);
17: break;
18: default: close(ssock);
19: break;
```

Algorithm 35 Función do_echo

Ensure: void do echo(int fd)

Require: char buf[4096],resp[100];

Require: int cc,org,faltan,cc2,opc;

```
1: while cc=read(fd,buf,sizeof(buf)) do
2:    if cc<0 then
3:       perror(Read");
4:       exit(6);
5:    end if
6:    switch(buf[0])
7:    case '0':strcpy(resp,"number zero");
8:    break;
9:    case '1':strcpy(resp,"number one");
10:   break;
11:   case '2':strcpy(resp,"number two");
12:   break;
13:   default: strcpy(resp,"fuera de rango[0,2]");
14:   break;
15:   org=0;
16:   faltan=strlen(resp);
17:   Los que hay que mandar
18:   while faltan do
19:      if (cc2=write(fd,&resp[org],faltan))<0 then
20:         perror("Fallo al escribir");
21:         exit(7);
22:      end if
23:      org+=cc2;
24:      faltan=cc2;
25:   end while
26: end while
27: close(fd);
```

3.7. Interconexión de Threads bajo una Red

Los subprocesos independientes proporcionan la concurrencia en una aplicación y puede hacer que mejore el rendimiento global. El uso de subprocesos permite que las aplicaciones sean estructuradas de manera eficiente atendiendo a varias tareas en paralelo. El multithreading es especialmente útil cuando:

- Hay un conjunto de operaciones largas que no dependen necesariamente de otro tipo de procesamiento.
- La cantidad de datos que se va a compartir es pequeño e identificable.
- Puede interrumpir la tarea en las diferentes actividades que se ejecutan en paralelo.
- Hay ocasiones en que los objetos deben ser reentrantes.

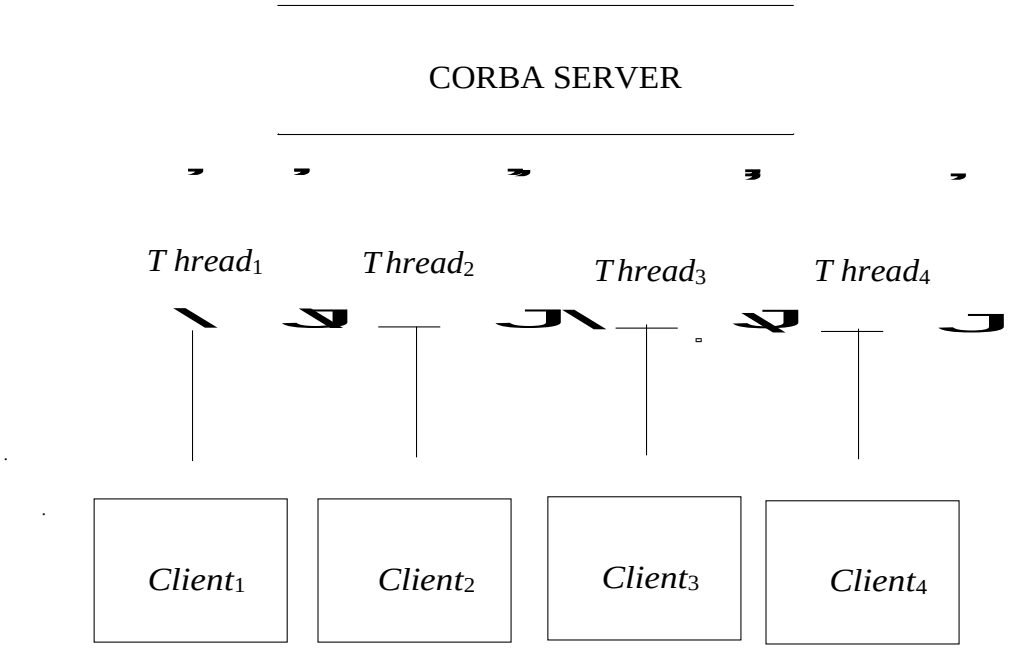


Figura 3.10: Aplicación CORBA con servidor y multihilos

En la figura se pueden observar algunas caracter´ısticas de las aplicaciones CORBA con servidor y multihilos como lo son:

- Las instancias de objetos de un servidor pueden manejar varias peticiones del cliente simultáneamente.
- Un objeto de un servidor puede hacer invocaciones recursivas sobre s´ımismo.

- Los objetos de un servidor pueden crear y supervisar sus propios subprocesos para implementar el paralelismo dentro de un método.

Un diseño multiproceso puede reducir significativamente el tiempo de espera entre la solicitud y la finalización de las operaciones. Esto es cierto solo en situaciones como:

- Cuando las operaciones realizan un gran número de operaciones de E/S.
- Cuando se accede a una base de datos.
- Invocando las operaciones en objetos remotos o si están vinculados a la CPU en un equipo multiprocesador.

Implementar el subprocesamiento múltiple en un proceso servidor puede aumentar el número de peticiones al servidor en una cantidad fija de tiempo. El requisito principal para aplicaciones servidoras con multiproceso es la manipulación simultánea de múltiples solicitudes de clientes. **Las motivaciones para el desarrollo de este tipo de servidor son:**

1. **Simplificar el diseño del programa:** ésto se logra permitiendo múltiples tareas de servidor para proceder de forma independiente mediante abstracciones de programación convencional.
2. **Mejorar el rendimiento:** ésto se consigue aprovechando las capacidades de procesamiento paralelo de las plataformas hardware de multiprocesos y la superposición de la computación con la comunicación.
3. **Mejorar el tiempo de respuesta:** asociando subprocesos independientes con diferentes tareas de servidor, los clientes no se bloquearán mutuamente durante un período.
4. **Simplificar la codificación de llamadas de procedimiento remoto y conversaciones:** algunas aplicaciones son más fáciles en el código cuando se utilizan subprocesos independientes para interactuar con diferentes llamadas a procedimientos remotos (RPC) y conversaciones.
5. **Proporcionar acceso simultáneo a múltiples aplicaciones:** cuando ocurre la envoltura de aplicaciones heredadas o bases de datos en un servidor CORBA, las implementaciones pueden interactuar con más de una aplicación heredada en un momento.
6. **Reducir el número de servidores necesarios:** debido a que un servidor puede enviar múltiples subprocesos de servicio, el número de servidores que necesita su aplicación puede ser reducido.

En general, las aplicaciones de servidores con multiproceso requieren procesos complicados para la sincronización de servidores. El rendimiento real de un determinado modelo de simultaneidad depende de los siguientes factores:

- **Características de las solicitudes del cliente:**son las peticiones de larga o corta duración.
- **Cómo se implementan los hilos:** los hilos son administrados en el kernel del sistema operativo en una biblioteca, en el espacio de usuario o alguna combinación de ambos.
- **Sistema operativo y la sobrecarga de la red:** cuánta sobrecarga adicional se presenta varias veces en la configuración y derriba las conexiones.
- **Configuración del sistema de niveles superiores:** realizar el balanceo dinámico de la carga o de otros factores que afectan el desempeño.

3.8. Modelos de subprocesos

Hay un número de modelos diferentes que se pueden utilizar para diseñar la concurrencia en los servidores, los siguientes casos son:

- **Thread-Per-Request Model:**

  En este modelo, cada petición de un cliente es procesada en un hilo diferente de control. Este modelo es útil cuando un servidor normalmente recibe peticiones de larga duración desde varios clientes. Es menos útil para las solicitudes de corta duración debido a la sobrecarga de crear un nuevo subproceso para cada solicitud. Cada vez que llega una solicitud nueva, BEA Tuxedo asocia dicha solicitud con un hilo y lo ejecuta. Ya que una aplicación servidor con multiproceso puede alojar más de un subproceso a la vez, se puede ejecutar simultáneamente más de una solicitud del cliente al mismo tiempo. Este modelo de solicitud requiere que el diseño de los servidores en las aplicaciones sean seguros, ya que debe implementar mecanismos de concurrencia para controlar el acceso a los datos que pueden ser compartidas entre múltiples objetos de servidor. La necesidad de utilizar mecanismos de control de concurrencia aumenta la complejidad del proceso de desarrollo de aplicaciones. Además, si muchos clientes realizan solicitudes simultáneamente, este diseño puede consumir una gran cantidad de recursos del sistema operativo.

- **Thread-Per-Object Model** En este método se asocia cada objeto activo en el proceso de servidor con un único subproceso. Cada petición de un objeto establece una asociación entre un subproceso de despacho y de objeto. Las peticiones seguidas para el mismo objeto

pueden ser atendidas por diferentes hilos. Un subproceso específico puede ser compartido por varios objetos.

- **The Thread Pool** Los grupos de hilos son un medio para reducir el costo de administración de éstos. Al inicio y según sea necesario, los hilos se crean, asignan y liberan a un grupo de hilos disponibles donde el hilo espera hasta que vuelva a ser necesario para procesar las solicitudes en el futuro. Los grupos de hilos pueden utilizarse para apoyar cualquiera de los modelos de hilos descritos anteriormente. Los grupos de hilos son adecuados para situaciones en las que desea limitar la cantidad de recursos del sistema que pueden ser consumidos por el servidor de los subprocesos. Cuando se utiliza un grupo de hilos, las peticiones del cliente se ejecutan simultáneamente hasta que el número de solicitudes supera el número de subprocesos asignado. El conjunto de hilos tiene las siguientes caracter´ısticas y comportamientos:

  - Puede establecer el tamaño máximo de la asignación de hilos como una función de administración
  - El software BEA Tuxedo asigna hilos del grupo según sea necesario. Los hilos se utilizan durante el procesamiento de una solicitud y luego se liberan de nuevo.
  - Los hilos pueden ser reutilizados consecutivamente para prestar servicio a varias solicitudes y varios objetos.

3.9. Webservice

El término Servicios Web designa una tecnología que permite que las aplicaciones se comuniquen en una forma que no depende de la plataforma ni del lenguaje de programación. Un servicio web es una interfaz de software que describe un conjunto de operaciones a las cuales se puede acceder por la red a través de mensajería XML estandarizada. Usa protocolos basados en el lenguaje XML con el objetivo de describir una operación para ejecutar o para intercambiar datos con otro servicio web. Un grupo de servicios web que interactúa de esa forma define la aplicación de un servicio web específico en una arquitectura orientada a servicios (SOA). Los servicios web usan XML, que puede describir cualquier tipo de datos en una forma realmente independiente de plataforma para el intercambio entre sistemas, lo que permite el movimiento hacia aplicaciones flojamente acopladas. Además, los servicios web pueden funcionar a un nivel más abstracto que puede reevaluar, modificar o manejar tipos de datos dinámicamente en demanda(mediante solicitud). Por tanto, en términos técnicos, los servicios web pueden manejar datos con mucho

más facilidad y permiten una comunicación más libre entre los software. Servicios que ofrece un web service

Los servicios web permiten:

- Interacción entre servicios en cualquier plataforma, escritos en cualquier lenguaje.
- Conceptualizar funciones de aplicaciones en tareas, lo que lleva al desarrollo y a flujos de trabajo orientados a tareas. Eso posibilita más abstracción del software que puede ser empleado por usuarios menos técnicos que trabajan con análisis en el ámbito de negocios.
- Permite el acoplamiento flojo, lo que significa que las interacciones entre aplicaciones de servicio no se rompen siempre que haya un cambio en la forma de diseño o implementación de un servicio o más.
- Adaptar las aplicaciones ya existentes a las cambiantes condiciones empresariales y necesidades de clientes.
- Proporcionar aplicaciones de software ya existentes o legadas con interfaces de servicio sin cambiar las aplicaciones originales, lo que permite operar totalmente en el entorno de servicios.
- Introducir otras funciones administrativas o de gestión de operaciones como confiabilidad, rendición de cuentas, seguridad, etc., independientemente de la función original, lo que aumenta su versatilidad y utilidad en el entorno de computación empresarial.

3.10. DCOM

La arquitectura DCOM es una extensión de COM, y éste define como los componentes y sus clientes interactúan entre sí. Esta interacción es definida de tal manera que el cliente y el componente pueden conectar sin la necesidad de un sistema intermedio. El cliente llama a los métodos del componente sin tener que preocuparse de niveles más complejos. En los actuales sistemas operativos, los procesos están separados unos de otros. Un cliente que necesita comunicarse con un componente en otro proceso no puede llamarlo directamente y tendrá que utilizar alguna forma de comunicación entre procesos que proporcione el sistema operativo. COM proporciona este tipo de comunicación de una forma transparente: intercepta las llamadas del cliente y las reenvía al componente que está en otro proceso. Cuando el cliente y el componente residen en distintas máquinas, DCOM simplemente reemplaza la comunicación entre procesos locales por un protocolo de red. Ni el cliente ni el componente se enteran de que la unión que los conecta es ahora un poco más grande. Las librerías de COM proporcionan

servicios orientados a objetos a los clientes y componentes y utilizan RPC y un proveedor de seguridad para generar paquetes de red estándar que entiendan el protocolo estándar de DCOM. DCOM toma ventaja de forma directa y transparente de los componentes COM y herramientas ya existentes. Un gran mercado de todos los componentes disponibles haría posible reducir el tiempo de desarrollo integrando soluciones estandarizadas en las aplicaciones de usuario. Muchos desarrolladores están familiarizados con COM y pueden aplicar fácilmente sus conocimientos a las aplicaciones distribuidas basadas en DCOM. Cualquier componente que sea desarrollado como una parte de una aplicación distribuida es un candidato para ser reutilizado. Organizando los procesos de desarrollo alrededor del paradigma de los componentes se permite continuar aumentando el nivel de funcionalidad en las nuevas aplicaciones y reducir el tiempo de desarrollo. Diseñando para COM y DCOM se asegura que los componentes creados serán útiles ahora y en el futuro.

3.10.1. Independencia del lenguaje de programación.

Una cuestión importante durante el diseño e implementación de una aplicación distribuida es la elección del lenguaje o herramienta de programación. La elección es generalmente un término medio entre el coste de desarrollo, la experiencia disponible y la funcionalidad. Como una extensión de COM, DCOM es completamente independiente del lenguaje. Virtualmente cualquier lenguaje puede ser utilizado para crear componentes COM y estos componentes pueden ser utilizadoz por muchos lenguajes y herramientas. Java, Microsoft Visual C++, Microsoft Visual Basic, Delphi, PowerBuilder y Micro Focus COBOL interactúan perfectamente con DCOM.

Con la independencia de lenguaje de DCOM, los desarrolladores de aplicaciones pueden elegir las herramientas y lenguajes con los que estén más familiarizados. La independencia del lenguaje permite crear componentes en lenguajes de nivel superior como Microsoft Visual Basic, y después reimplementarlos en distintos lenguajes como C++ o Java, que permiten tomar ventaja de características avanzadas como multihilo.

3.10.2. Independencia del protocolo.

Muchas aplicaciones distribuidas tienen que ser integradas en la infraestructura de una red existente.Los desarrolladores de aplicaciones deben tener cuidado de mantener la aplicación lo más independiente posible de la infraestructura de la red. DCOM proporciona esta transparencia: DCOM puede utilizar cualquier protocolo de transporte, como TCP/IP, UDP, IPX/SPX y NetBIOS. DCOM proporciona un marco de seguridad a todos estos protocolos. Los desarrolladores pueden simplemente utilizar las características proporcionadas por DCOM

y asegurar que sus aplicaciones son completamente independiente del protocolo.

3.11. Comunicación remota en una red

Una red de computadoras, también llamada red de ordenadores, red de comunicaciones de datos o red informática, es un conjunto de equipos informáticos y software conectados entre sí por medio de dispositivos físicos que envían y reciben impulsos eléctricos, ondas electromagnéticas o cualquier otro medio para el transporte de datos, con la finalidad de compartir información, recursos y ofrecer servicios.
Como en todo proceso de comunicación, se requiere de un emisor, un mensaje, un medio y un receptor. La finalidad principal para la creación de una red de computadoras es compartir los recursos y la información en la distancia, asegurar la confiabilidad y la disponibilidad de la información, aumentar la velocidad de transmisión de los datos y reducir el costo. Un ejemplo es Internet, la cual es una gran red de millones de computadoras ubicadas en distintos puntos del planeta interconectadas básicamente para compartir información y recursos.

Definición: Una red de computadoras, por lo tanto, es un conjunto de estas máquinas donde cada uno de los integrantes comparte información, servicios y recursos con el otro.

3.11.1. Componentes de una Red

Servidor: Es una computadora que, formando parte de una red, provee servicios a otras computadoras denominadas clientes.

Estaciones de Trabajo: Cuando una computadora se conecta a una red, la primera se convierte en un nodo de la última y se puede tratar como una estación de trabajo o cliente

Sistema de Cableado: El sistema de la red esta constituido por el cable utilizado para conectar entre si el servidor y las estaciones de trabajo.

Disponibilidad: Infraestructura de alta disponibilidad, que incluye servicios de checkpointing, recuperación tras fallo, tolerancia a fallos entre otros.

Sistema operativo: Pueden utilizarse la mayor´ıa de los S.O. del mercado:

- UNIX: es un sistema operativo portable, multitarea y multiusuario.
- Linux: Es un sistema operativo libre, basado en Unix.

- Wxp: (Windows Experience) es una versión de Microsoft Windows, fue el más avanzado de la compañía Microsoft.

3.11.2. Protocolos rápidos de comunicación

- Active message: es un mecanismo de comunicación asíncrona para exponer la flexibilidad y el rendimiento del hardware en las redes de interconexión modernas.
- Fast protocol: es un estándar de tecnología desarrollada por FIX Protocolo Ltd., dirigida específicamente a la optimización de la representación de datos en la red. Se utiliza para apoyar de alto rendimiento, baja latencia comunicaciones de datos entre instituciones financieras.
- VIA (Virtual Interface Adapter): es un protocolo especialmente utilizado para optimizar la representación de datos en una red.

3.12. Topologías de una red

Red bus

Red cuya topología se caracteriza por tener un único canal de comunicaciones (denominado bus, troncal o backbone) al cual se conectan los diferentes dispositivos. De esta forma todos los dispositivos comparten el mismo canal para comunicarse entre sí.

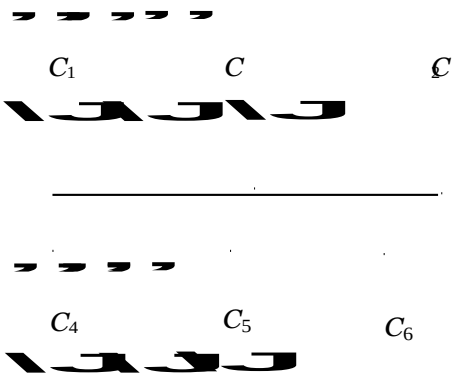


Figura 3.11: Topología Bus

Red de Estrella

Una red en estrella es una red en la cual las estaciones están conectadas directamente a un punto central y todas las comunicaciones se deben hacer necesariamente a través de éste. Dada su transmisión, una red en estrella activa tiene un nodo central activo que normalmente tiene los medios para prevenir problemas relacionados con el eco.

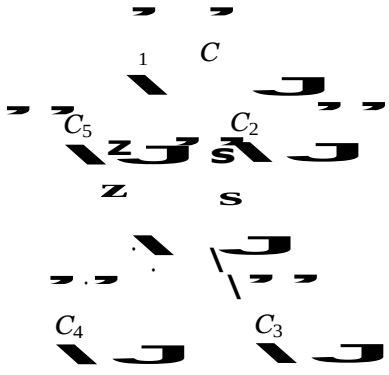


Figura 3.12: Topología Estrella

Red en anillo

Topología de red en la que cada estación está conectada a la siguiente y la última está conectada a la primera. Cada estación tiene un receptor y un transmisor que hace la función de repetidor, pasando la señal a la siguiente estación.

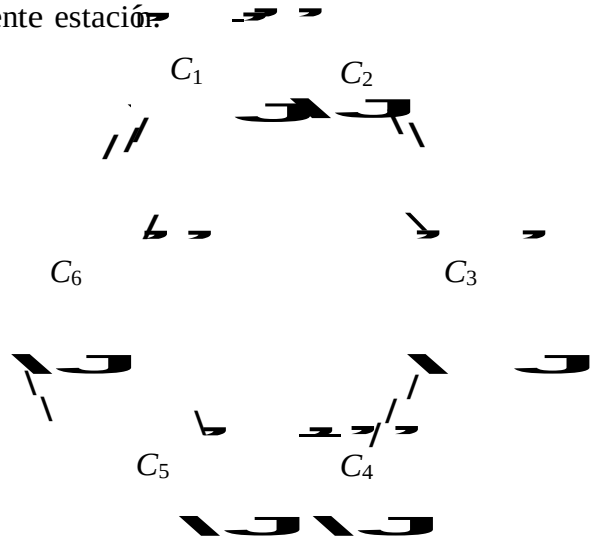


Figura 3.13: Topología Anillo

Red en malla

La topología en malla es una topología de red en la que cada nodo está conectado a todos los nodos. De esta manera es posible llevar los mensajes de un nodo a otro por diferentes caminos. Si la red de malla está completamente conectada, no puede existir absolutamente ninguna interrupción en las comunicaciones. Cada servidor tiene sus propias conexiones con todos los demás servidores.

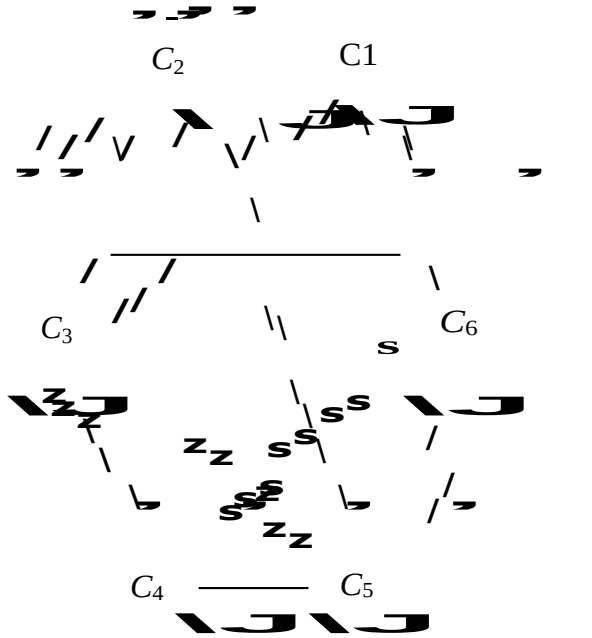


Figura 3.14: Topología Malla

Red en árbol

Topología de red en la que los nodos están colocados en forma de árbol. Desde una visión topológica, la conexión en árbol es parecida a una serie deredes en estrella interconectadas salvo en que no tiene un nodo central. En cambio, tiene un nodo de enlace troncal, generalmente ocupado por un hub o switch, desde el que se ramifican los demás nodos. Es una variación de la red en bus, la falla de un nodo no implica interrupción en las comunicaciones. Se comparte el mismo canal de comunicaciones.

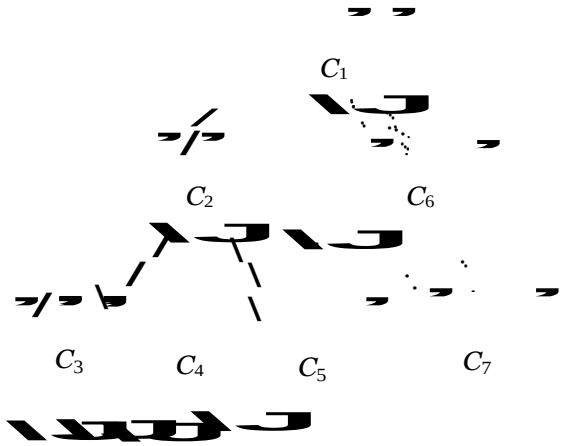


Figura 3.15: Topología Arbol

Anillo Doble

La topología de anillo doble es igual a la topología de anillo, con la diferencia de que hay un segundo anillo redundante que conecta los mismos dispositivos. Uno de los anillos se utiliza para la transmisión y el otro actúa como anillo de seguridad o reserva. Si aparece un problema, como un fallo en el anillo o una ruptura del cable, se reconfigura el anillo y continúa la transmisión.

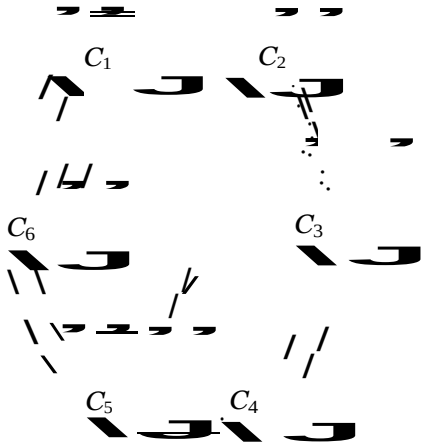


Figura 3.16: Topología Doble Anillo

Topologia Estrella - Bus

Podemos ver una red en bus al que están conectados los hubs de pequeñas redes en estrella. Por lo tanto, no hay ningún ordenador que se conecte directamente al bus. En esta topología mixta, si un ordenador falla, entonces es detectado por el hub al que está conectado y simplemente lo aísla del resto de la red. Sin embargo, si uno de los hubs falla, entonces los ordenadores que están conectados a él en la red en estrella no podrán comunicarse y además, el bus se partirá en dos partes que no pueden comunicarse entre ellas.

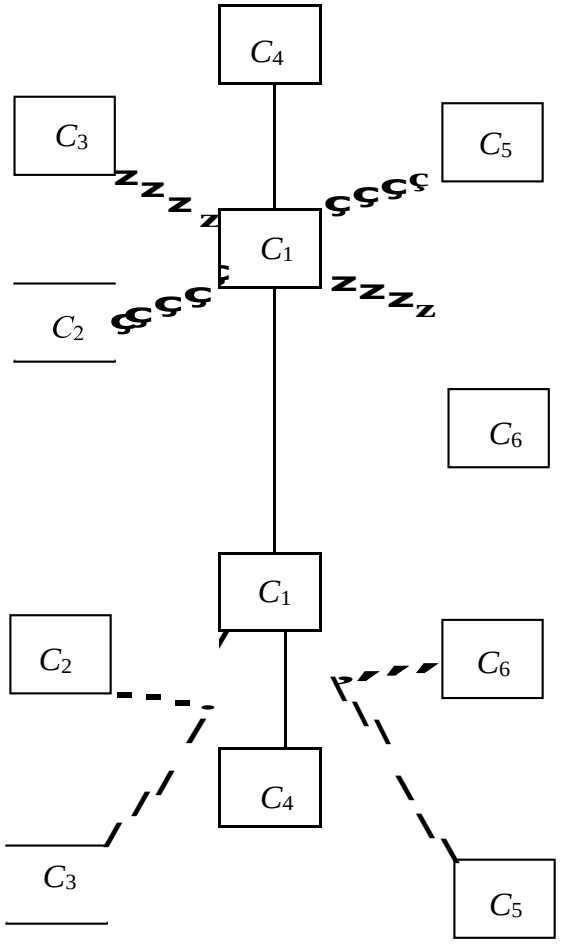


Figura 3.17: Topología Estrella-Bus

Topologia de estrella anillo

Encontramos que el cableado forma f´ısicamente una estrella, pero el hub al que se conecta hace que la red funcione como un anillo con la ventaja de que si uno de los ordenadores falla, el hub se encarga de sacarlo del anillo para que éste siga funcionando.

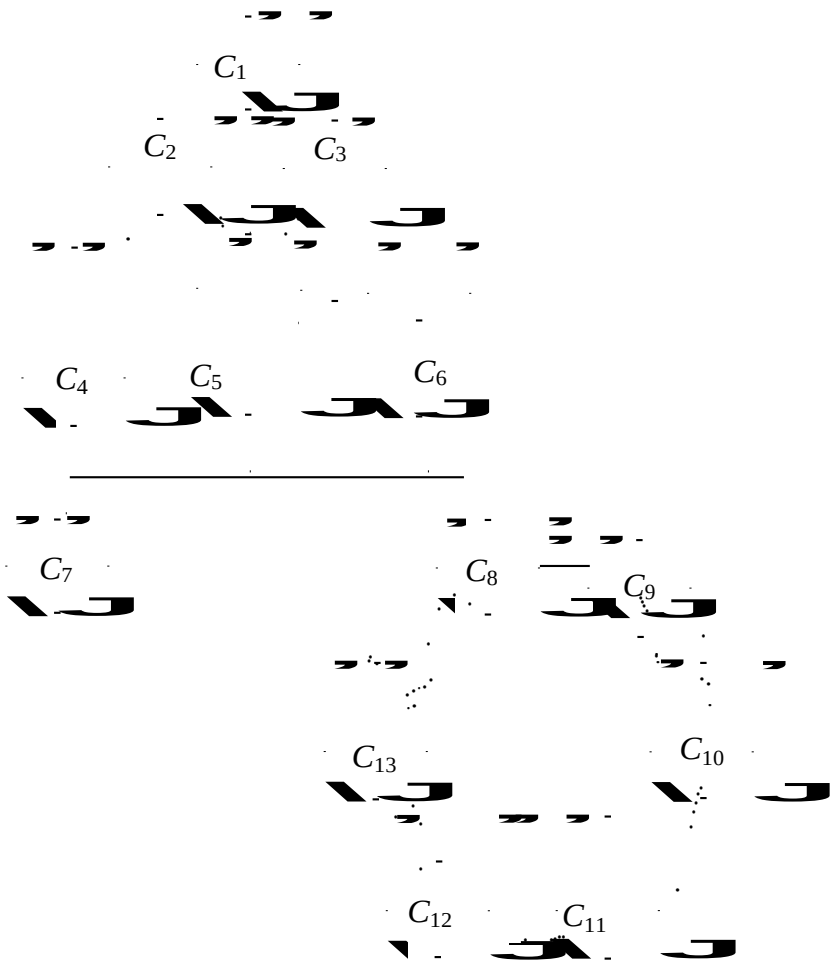


Figura 3.18: Topología Estrella Anillo

Red Mixta

La red totalmente conexa es una topología muy eficaz ya que está unida totalmente. En caso de que uno de los cableados se llegue a dañar de algún nodo la información no se verá afectada para los demás nodos.

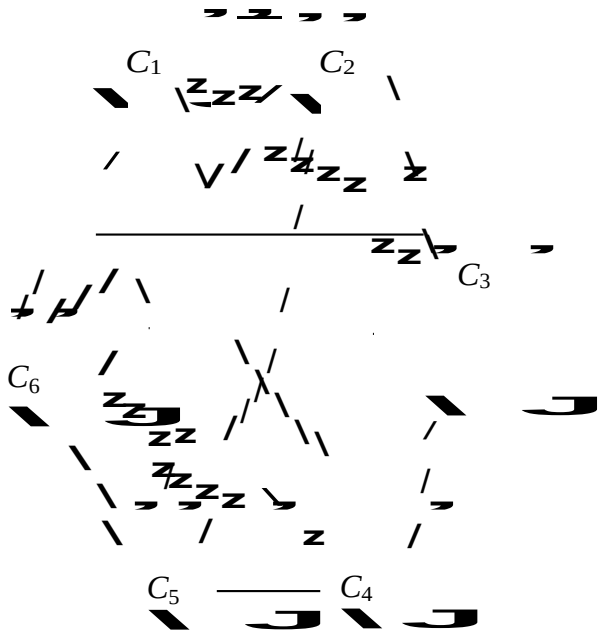


Figura 3.19: Topología Conexa

3.13. Middleware

El middleware es un software que generalmente actúa entre el sistema operativo y las aplicaciones con la finalidad de proveer a una red de computadoras lo siguiente:

- Una interfaz única de acceso al sistema, denominada SSI (Single System Image), la cual genera la sensación al usuario de que utiliza un único ordenador muy potente.

- Herramientas para la optimización y mantenimiento del sistema: migración de procesos, checkpoint-restart (congelar uno o varios procesos, mudarlos de servidor y continuar su funcionamiento en el nuevo host), balanceo de carga, tolerancia a fallos, etc.

- Escalabilidad: debe poder detectar automáticamente nuevos servidores conectados a la red de computadoras para proceder a su utilización.

El middleware recibe los trabajos entrantes a la red y los redistribuye de manera que el proceso se ejecute más rápido y el sistema no sufra sobrecargas en un servidor. Esto se realiza mediante políticas definidas en el sistema (automáticamente o por un administrador) que le indican dónde y cómo debe distribuir los procesos, por un sistema de monitorización, el cual controla la carga

de cada CPU y la cantidad de procesos en él.

Existen diversos tipos de middleware, como por ejemplo: MOSIX, OpenMOSIX, Cóndor, OpenSSI, etc.

3.14. Object Request Broker (ORB)

El núcleo de ORB provee las facilidades necesarias para escribir sistemas distribuidos, usar diferentes lenguajes de programación y sistemas operativos, además permite integrar aplicaciones a nuevos sistemas. OMG ha definido CORBASERVICES y CORBAFACILITIES para extender la ayuda incorporada a las aplicaciones. Así CORBA,CORBASERVICES y CORBAFACILITIES crean lo que se conoce como Object Managment Architecture(OMA).
Los servicios de CORBA son los siguientes: **Servicios relacionados con Sistemas Distribuidos y Servicios Generales**

1. **Naming Service:** permite que el usuario pueda encontrar un objeto dando su nombre.
2. **Event Service:** permite que un cliente o servidor pueda mandar un mensaje a un evento a cualquier número de recibidores.
3. **Security Service:** un objeto o un grupo de objetos pueden estar protegidos, para que los usuarios privilegiados puedan realizar operaciones específicas. Las comunicaciones en la red también son encriptadas.
4. **Trading Service:** permite al cliente encontrar un objeto dando un valor.
5. **Life Cycle service:** define interfaces que permiten a los objetos ser creados, trasladados y copiados.
6. **Licensing Service:** permite que un servidor llame a una computadora y determine si tiene licencia para ejecutar el software.
7. **Time Service:** se usa para encontrar la hora del día para obtener un evento llamado antes de un tiempo específico.

Una implementación del núcleo de CORBA debe soportar lo siguiente:

1. El lenguaje IDL.
2. Mapeo de IDL a algún lenguaje de programación (C,Java,C++,Visual Basic,etc).
3. Soporte en tiempo de ejecución para pasar una solicitud del objeto entre un interlocutor y un objeto de destino.
4. Soporte en tiempo de ejecución para la programación de interfaces de una aplicación bajo CORBA.

5. El repositorio de interfaz, que te da la información en tiempo de ejecución.

La encarnación física de un ORB se compone por:

- Una librería que esta ligada con el código del cliente y servidor la cual permite a un cliente enviar y aceptar solicitudes de CORBA.

- Un proceso demonio que establece la comunicación de las conexiones entre clientes y servidores.

3.15. OMG

En 1989, se constituyó el Object Managment Group para solventar los problemas de desarrollo de aplicaciones distribuidas portales para sistemas heterogéneas. Las primeras especificaciones producidas por el OMG son la arquitectura de gestión de objetos(OMA) y su núcleo, la especificacion de CORBA, proporcionan un entorno arquitectural completo.

OMA usa dos modelos para describir cómo interaccionan los objetos distribuidos entre sí y cómo se pueden especificar estas interacciones de forma independiente de la plataforma. El modelo de objetos define cómo se describen las interfaces de los objetos distribuidos a través de entornos heterogéneos y el modelo de referencia caracteriza las interacciones entre estos objetos, el modelo de objetos define un objeto como una entidad encapsulada con una identidad distinto e inmutable a cuyos servicios se puede acceder únicamente a través de interfaces bien definidas. Los clientes usan los servicios de un objeto para emitir peticiones al mismo.

El modelo de referencia proporciona categor´ıas de interfaz que son agrupaciones generales de interfaces de objetos, un ORB permite la comunicación transparente entre clientes de objetos, activando los objetos que están inactivos cuando se emiten las peticiones para los mismos.

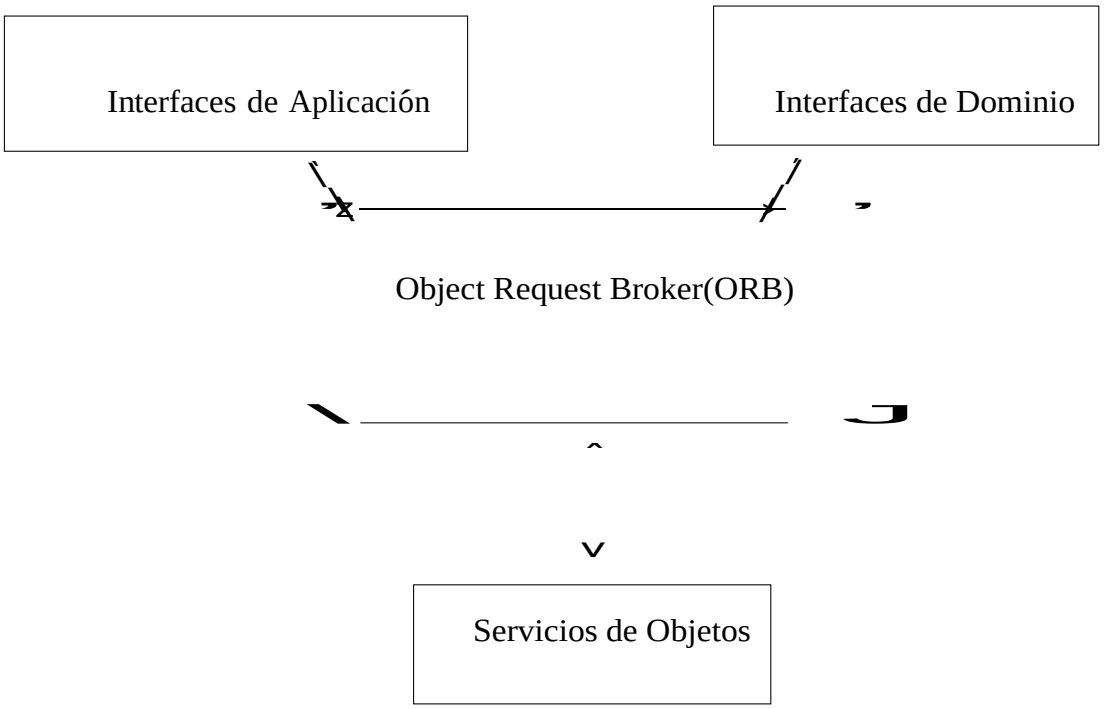


Figura 3.20: Comunicación de la interfaz con ORB

En la figura 3.20 se muestra la comunicación de la interfaz con ORB, a continuación se muestran las facilidades de comunicación y activación del ORB las cuales son :

- Servicios de objetos: son interfaces independientes del dominio, usadas por muchas aplicaciones con objetos distribuidos.
- Interfaces de dominio: estas interfaces son específicas de un dominio o verticalmente orientadas.
- Interfaces de aplicación: desarrolladas para una determinada aplicación.

Adaptadores de Objetos

Sirven en CORBA como un adhesivo entre los sirvientes y el ORB. Un adaptador de objetos es un objeto que adapta la interfaz de un objeto a una interfaz distinta esperada por un usuario. Los adaptadores de objetos de CORBA satisfacen tres requisitos clave:

1.Referencias de objetos, que permiten a los clientes conocer las direcciones de los objetos.

2.Aseguran que cada objeto destino esté encarnado a un sirviente .

3.Reciben las peticiones emitidas por el ORB del servidor y las redirigen a los sirvientes que encarnan a los objetos destino.

3.16. CORBA

Especificaba a BOA (The basic object adaptor), sin embargo, no evolucionó como se esperaba debido a los siguientes problemas en la especificación del BOA:

- BOA estaba escrito para proporcionar únicamente a sirvientes C, un adaptador de objetos que proporciona soporte sólido para sirvientes escritos en un lenguaje de programación no suele ser adecuado para proporcionar soporte para sirvientes de lenguajes distintos.
- Ciertas interfaces no fueron definidas, y como resultado, no hab´ıa operaciones de registro de sirvientes.

La versión de CORBA 2.2 introdujo al Adaptador de objetos Portable (Portable Object Adapter,POA) para reemplazar a BOA.

3.16.1. Protocolos Inter-ORB

CORBA 2.0 introdujo una arquitectura de interoperabilidad general para el ORB, denominada Protocolo General Inter-ORB(GIOP), es un protocolo abstracto que especifica una sintaxis de transferencia y un conjunto estándar de formato de mensajes para permitir que los ORB desarrollados independientemente se puedan comunicar usando cualquier conexión de transporte. El protocolo Internet Inter-ORB(IIOP) especifica cómo se implementa el GIOP sobre el protocolo de control de transmisión/protocolo de Internet(TCP/IP). El formato de referencia del objeto estándar, denominado Referencia de Objeto interoperable es suficientemente flexible para almacenar la información de casi cualquier protocolo Inter-ORB imaginable. Un IOR identifica a uno o más protocolos de apoyo y contiene información específica para cada protocolo, este arreglo permite que se puedan añadir nuevos protocolos sin tener que rehacer las aplicaciones. Un IOR y un IIOP contiene un nombre máquina, un número de Puerto TCP/IP y una clave de objeto que identifica al objeto destino en la combinación de nombre de máquina y puerto dado.

3.16.2. Invocación de peticiones

El ORB envía un mensaje a un objeto cada vez que el cliente llama a un operación. Para enviar un mensaje a un objeto, el cliente debe tener una referencia a un objeto, la cual es donde actúa como un manejador que identifica de manera única al objeto destino y encapsula toda la información necesaria para que el ORB pueda enviar el mensaje destino correcto.

Cuando un cliente llama a una operación a través de un cliente a través de una referencia a objeto, el ORB hace lo siguiente:

1. Localiza al objeto destino.
2. Activa la aplicación servidor si aún no está funcionando.
3. Transmite los argumentos necesarios para realizar la llamada.
4. Activa un sirviente para el usuario si es necesario.
5. Espera hasta que se complete la petición.
6. Devuelve cualquier parámetro out e inout y el valor devuelto al cliente cuando la llamada termina con éxito.
7. Devuelve una excepción al cliente cuando falta la llamada.

Una inovación de peticiones tiene la siguientes características:

- Transparencia de localización: los procesos servidores no están obligados a permanecer en la misma máquina, se pueden cambiar de una máquina a otra sin que los clientes sean conscientes de ello.
- Transparencia de servidor: el cliente no necesita saber que servidor implementa que objetos.
- Independencia del lenguaje: el cliente no necesita preocuparse del lenguaje usado por el servidor.
- Independencia de la implementación: el cliente no sabe como funciona la implementación, el servidor puede implementar sus objetos como servicios prontos de C++ o el servidor no puede implementar sus objetos usando técnicas no orientadas a objetos.
- Independencia de la arquitectura : el cliente no es consciente de la arquitectura de la CPU que se usa en el servidor y es independiente de detalles como la ordenación de bytes y coherencia de estructuras
- Independencia del sistema operativo: al cliente no le preocupa el sistema operativo que usa el servidor, el servidor puede incluso estar implementando sin el soporte de un sistema operativo.

- Independencia del protocolo: el cliente no sabe que protocolo de comunicación se usa para enviar los mensajes. Si hay varios tipos de protocolos disponibles para comunicarse con el servidor, el ORB selecciona transparentemente un protocolo en tiempo de ejecución.
- Independencia del transporte : el cliente ignora el nivel de transporte y de enlace usado para transmitir los mensajes. El ORB puede usar transparentemente varias tecnolog´ıas de red como Ethernet, ATM.

3.16.3. Semántica de referencias a objetos

Las referencias a objetos tienen una semántica muy similar a los punteros a instancias de clases ordinarios de C++:

- Cada referencia a objeto identifica exactamente a una instancia de un objeto.
- Una referencia a un objeto denota exactamente a un objeto CORBA, una referencia a un objeto puede dejar de funcionar únicamente cuando el objeto destino se ha destruido de forma permanente, esto significa que una referencia a un objeto destruido no puede denotar de forma accidental a un objeto creado posteriormente.
- Varias referencias distintas pueden identificar al mismo objeto.
- Cada referencia nombra exactamente a un objeto, pero un objeto puede tener varios nombres.
- Las referencias pueden ser nulas: las referencias nulas son útiles para presentar semánticas no encontrado o no esta ahí, también se pueden usar para implementar parámetros opcionales. El paso de un valor nulo en tiempo de ejecución indica que el parámetro no está ahí.

  Las referencias pueden quedar inválidas: después de que un servidor haya pasado una referencia a un objeto a un cliente, esa referencia está permanentemente fuera del control del servidor y se puede propagar libremente por medios invisibles al ORB. Esto significa que CORBA no tiene mecanismos automáticos para que el servidor informe a un cliente
- cuando un objeto perteneciente a una referencia ha sido destruido.

  Las referencias son opacas: las referencias a objetos contienen un cierto número de componentes estandarizados iguales para todos los ORB, así como información particular
- específica para cada ORB. Para permitir la compatibilidad de código fuente entre distintos ORB, a los clientes y a los servidores no se les permite ver la representación de la referencia a objeto.

- Las referencias usan tipado fuerte: cada referencia a objeto contiene una indicación de la interfaz proporcionada por ese objeto. Este arreglo permite al ORB forzar las correspondencia de tipos. Para lenguajes con tipado estático como C++, la correspondencia de tipos se fuerza también en el momento de la compilación. El mapping del lenguaje no permite llamar a una operación a menos que el objeto destino haya garantizado la oferta de esa operación en su interfaz.

- Las referencias proporcionan enlace tard´ıo: los clientes pueden tratar una referencia a un objeto derivado como si fuera una referencia a un objeto base. Este arreglo es análogo a las llamadas funciones virtuales de C++, la llamada a un método a través de un puntero base invoca a una función virtual de una instancia de la derivada. Una de las mayores ventajas de CORBA es que el polimorfismo y el enlace diferido para objetos remotos funcionan exactamente igual que lo hacen para objetos locales de C++.

- Las referencias pueden ser persistentes: los clientes y servidores pueden convertir una referencia a un objeto en un string y escribir el string a un disco. Más tarde, el string puede reconvertir en una referencia a objeto que denota al mismo objeto original.

- Las referencias pueden ser interoperables : CORBA especifica un formato estándar para referencias a objetos, también conocidos como referencias de objetos interoperables, además un ORB puede proporcionar codificaciones de referencias propietarios esta capacidad puede ser útil si un ORB ha sido adaptado para un entorno particular, como una base de datos orientada a objetos.

3.16.4. Adquisición de referencias

Las referencias a objetos son la única forma que tiene un cliente para acceder a los objetos destino. Un cliente no se puede comunicar, a menos que tenga una referencia al objeto destino. La forma más frecuente que un cliente puede adquirir referencias de objeto es recibirlas como respuesta a una llamada a una operación, los clientes simplemente contactan con un objeto y el objeto devuelve una o más referencias. Las referencias a objeto son siempre creadas por el ORB en tiempo de ejecución y en representación del cliente.

3.16.5. Contenido de referencia a un objeto

Un IOR tiene tres piezas principales de información:

- ID de repositorio.
- El ID de repositorio es un string que identifica al tipo más específico del IOR, en el momento en que se creó.
- La ID del repositorio permite localizar una descripción detallada de la interfaz en el repositorio de interfaces.

El ORB puede usar también el ID del repositorio para ofrecer conversión segura de tipos.

3.16.6. Información del destino

Este campo contiene toda la información que necesita el ORB para establecer una conexión física con el servidor que implementa el objeto. La información del destino especifica qué protocolo se debe usar y contiene información de direccionamiento físico apropiada para ese transporte en particular. La información de direccionamiento en el campo información destino puede contener directamente la dirección y el número de puerto del servidor que implementa el objeto. CORBA también permita que se incluya en la referencia información para varios protocolos y transportes distintos, permitiendo que una única referencia proporcione mas de un protocolo.

3.16.7. Clave del objetivo

El ID del repositorio y la información del destino son estándares, mientras que la clave del objeto contiene la información de cada propietario, como se organiza exactamente esta información y como se usa, depende de cada ORB. Sin embargo, todos los ORB permiten a los servidores incluir un identificador de objeto propio de la aplicación dentro de la clave del objeto cuando crean la referencia. La combinación de información del destino y la clave del objeto pueden aparecer muchas veces en un IOR. Esta multiplicidad de pares destino-clave, conocida como perfiles multicomponente, permite que un IOR también pueda contener múltiples perfiles, cada uno de los cuales contiene información separada para transporte y protocolo. La información del destino se usa en el ORB del cliente para identificar el espacio de direcciones del destino apropiado y la clave del objeto se usa en el ORB del servidor para identificar el objeto de destino dentro de su espacio de direcciones.

3.16.8. Referencias y proxies

Cuando un cliente recibe una referencia, el ORB del cliente instancia a un objeto proxy en el espacio de direcciones del cliente. Un proxy es una instancia de una clase C++ que proporciona al cliente una interfaz al objeto destino, el proxy delega las peticiones al correspondiente sirviente remoto y actúa como un embajador local para el objeto. El mapping de C++ no cambia si el cliente y el servidor se colocan en el mismo espacio de direcciones. Consideramos que los ORB que no mantienen un proxy en el caso de objetos situados en la misma localización son deficientes.

En ambos escenarios,remotos y locales, el proxy delega en un sirviente las llamadas hechas por el cliente a las operaciones. En el escenario remoto, el proxy envía la petición por la red mientras que en el escenario local las peticiones se resuelven mediante llamadas a funciones de C++.

Capítulo 4

Resultados de aplicaciones Concurrentes y Distribuidas

4.1. Buffer circular productor - consumidor

Un buffer circular, buffer cíclico o buffer de anillo es una estructura de datos que utiliza un buffer único o array ordinario y que adopta su nombre por la forma en que se ponen o sacan sus elementos. Estos buffers son de tamaño fijo, internamente es como si estuviera conectado de extremo a extremo.

- Un proceso produce datos que son posteriormente procesados por otro proceso
- Lo más cómodo es emplear un buffer circular

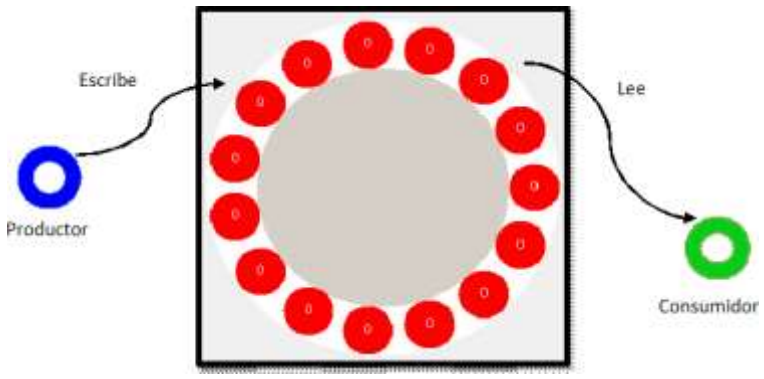


Figura 4.1: Buffer circular.

En la figura 4.1 se puede ver como se representa un buffer circular con dos procesos, un proceso productor que escribe en el buffer y un proceso consumidor que lee del buffer.

Interfaz gráfica

Figura 4.2: Buffer vacio.

En la figura 4.2 se muestra como la aplicación concurrente inicia con el Buffer está vacío. El consumidor espera a que se inicie el proceso productor para que éste produzca y le de y el proceso consumidor pueda obtener la información necesaria y consuma el producto del buffer circular. En este estado inicial tanto el proceso productor como el proceso consumidor no evitan las condiciones de competencia, esto puede causar confusión, porque las condiciones de competencia deben evitarse a toda costa, como se mencionó desde el capítulo 2, sin embargo decir que no evita las condiciones, es solo por un momento, porque inmediatamente después se aplican las reglas que garantizan una mejor exclusión mutua, esto por medio del método Mutex.

Fase experimental: iniciando ejecucion

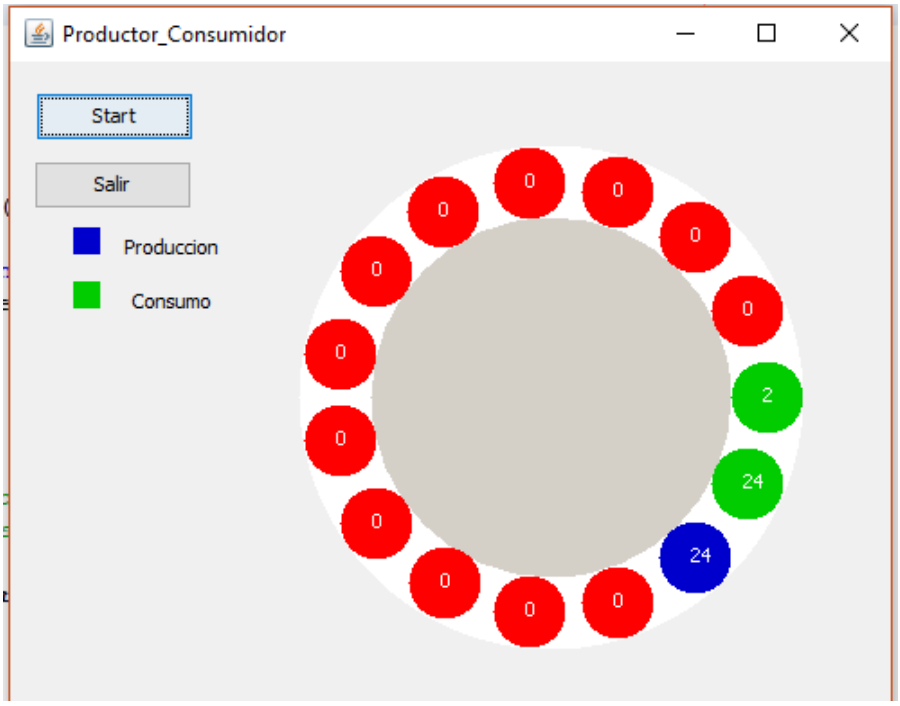


Figura 4.3: Iniciar ejecución.

En la figura 4.3 se inicia la ejecución de los procesos productor y consumidor en un buffer circular, se aplica la exclusión mutua por medio del método Mutex y la sincronización con el método de variables de condición, que este método mejora el rendimiento a comparación de utilizar semáforos.

Ejemplo de funcionamiento

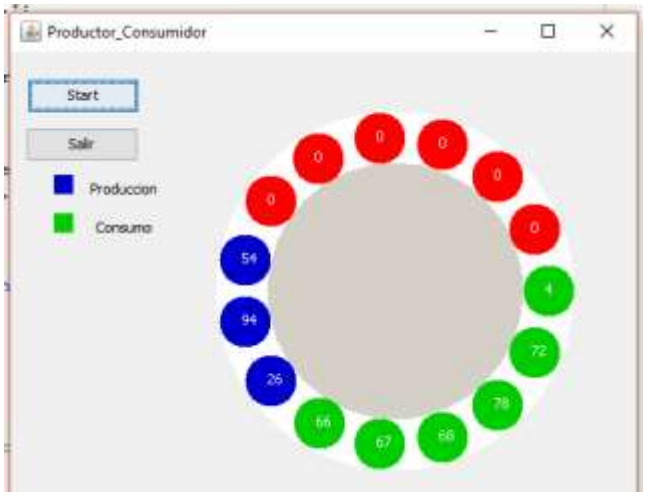


Figura 4.4: Buffer funcionando.

En la figura 4.4 conforme avanza la producción, el consumidor consume el producto después que el productor produce, esto debido a la sincronización. No se aplicó el método de monitores, por que este método es robusto y la aplicación del buffer circular es de un modo más sencilla. Todos esta teoría de mutex y sincronización junto con sus métodos se explican en los capítulos anteriores, donde se explican problemas y sus soluciones.

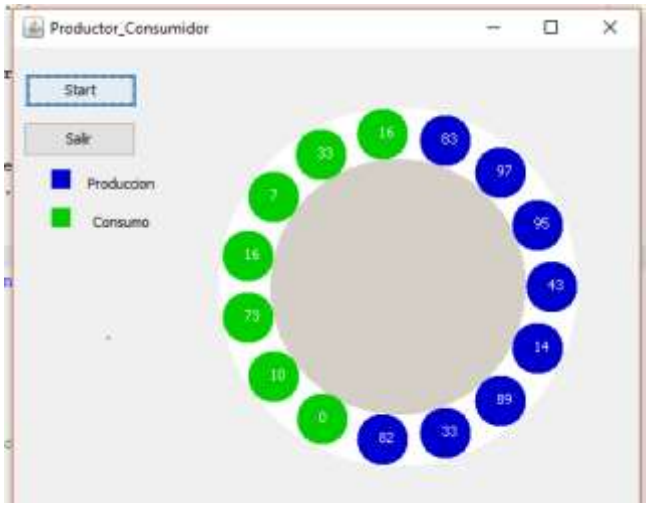


Figura 4.5: Productor y consumidor en plena sincronización.

En la figura 4.5 se puede apreciar que conforme avanza el tiempo de ejecución, el consumidor absorbe las producciones o productos que genera el procesos productor. Se puede pensar que en este tipo de aplicaciones los procesos se ejecutan al mismo tiempo, como si fuera paralelismo,

pero en realidad no es paralelismo, esto es concurrencia, puesto que los procesos que ejecuta la aplicación está en una sola CPU y los recursos compartidos son las localidades de buffer circular, así como, el mismo buffer. A este tipo de aplicaciones se les llaman aplicaciones concurrentes.

Iniciando procesos

```
public void actionPerformed (ActionEvent event) {

        Object source = event.getSource();

        if ((source == iniciar)&&(listo == false)) {

                productor.start();
                consumidor.start();
                listo = true;
                estado = true;

                System.out.println("\nIniciando Precesos...\n");

        }
```

Figura 4.6: Iniciando procesos.

En la figura 4.6 se muestra el código del evento donde se inician los procesos productor y consumidor, como se ve en la línea 4, el primer procesos a ejecutará es el productor, esto por el caso de uso visto anteriormente, donde indica que si no se ejecuta primero el productor, entonces no se puede consumir, por tal razón un línea después se ejecuta el consumidor. Así también, se puede observar que los procesos no se ejecutan si no es por un evento asignado a un boton en Java. Este botón es un componente gráfico en las librerías AWT del SDk del a JVM.

```
            else if ((source == detener)&&(estado == true)) {

                productor.suspend();
                consumidor.suspend();
                estado = false;

                System.out.println("\nPrecesos detenidos...\n");
            }

            else if ((source == reanudar)&&(estado == false)) {

                productor.resume();
                consumidor.resume();
                estado = true;

                System.out.println("\nPrecesos reanudados...\n");
            }

            else if (source == salir) {

                System.out.println("\nSaliendo...\n");
                System.exit(0);
            }
        }
```

Figura 4.7: Manejo de procesos.

En la figura 4.7 se muestra el segmento de código donde se aplican métodos exclusivos del API de Threads, que como se comentó en el capítulo 3, estos son procesos ligeros que pueden un proceso normal, solo que corren en el espacio usuario y los procesos trabajan en el espacio kernel. los métodos del API son de suspend(), este método vino a sustituir al método stop(), por que este último fue degradado por que hace daño al sistema base, es decir al sistema operativo y al CPU por que utiliza llamadas nativas a recursos reales de la computadora. El otro método es el resume, este hace que se activen nuevamente los procesos, es decir, hace el efecto de pausa.

4.2. Cena de filósofos

Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a co- mer. Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer. Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina interbloqueo o deadlock. El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

Interfaz gráfica

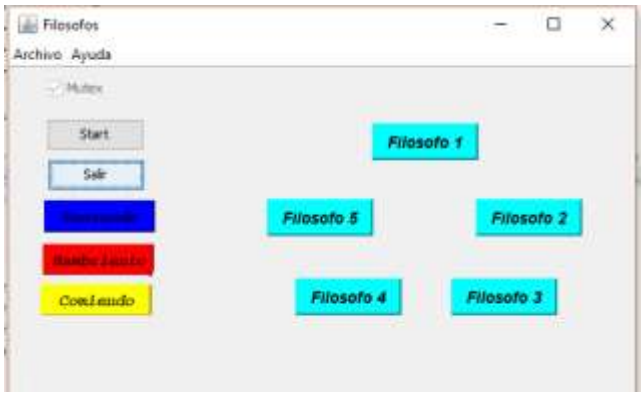


Figura 4.8: Vista de la corrida.

En la figura 4.8 Se muestra el estado de cada uno de los filósofos cuando ellos no estan ejecutandose.

Fase experimental

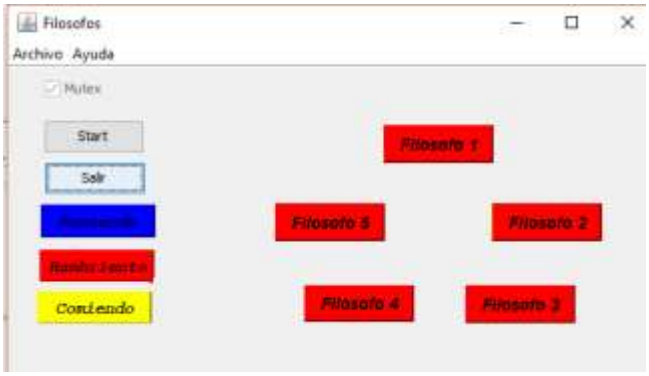


Figura 4.9: Filósofos hambrientos.

En la figura 4.9 se inicializa la ejecución del programa y notamos que las etiquetas llamadas Filósofo 1 al Filósofo 5 cambian de color pues comienzan con el estado de hambriento y la concurrencia.

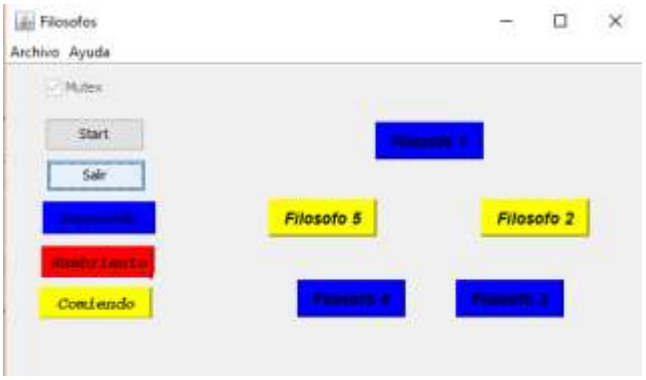


Figura 4.10: Filósofos en turno.

En la figura 4.10 se puede apreciar que conforme avanza el tiempo de ejecución, los estados de cada filósofo van variando pues en esta concurrencia los 5 filósofos en curso no pueden estar al mismo tiempo ejecutados en el mismo estado.

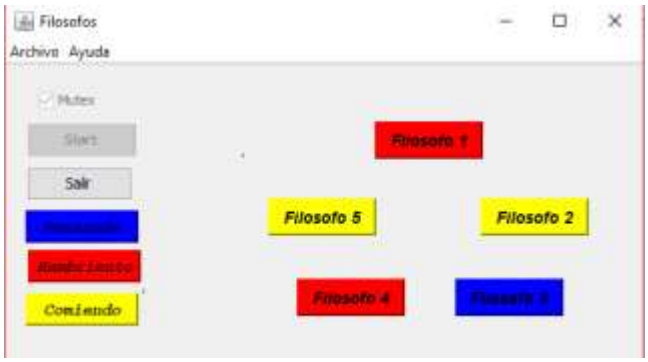


Figura 4.11: Filósofos cambiando estados.

En la figura 4.11 se puede apreciar que conforme avanza el tiempo de ejecución, los estados de cada filósofo van variando pues en esta concurrencia los 5 filósofos en curso no pueden estar al mismo tiempo en el mismo estado. Cuando un filósofo quiere comer se pone en la cola de los dos tenedores que necesita. Cuando un tenedor está libre lo toma. Cuando toma los dos tenedores, come y deja libre los tenedores.

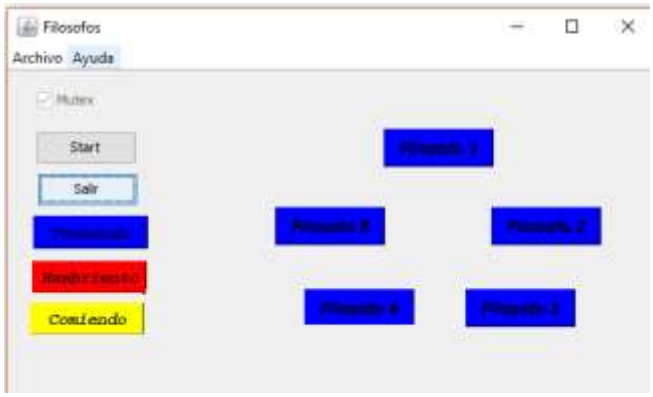


Figura 4.12: Ejemplo de deadlock.

En la figura 4.12 solo podemos observar un ejemplo sobre algún problema de sincronizacion llamdo deadlock.

Iniciando procesos.

```
            pintaFilo c5=new pintaFilo();
            Filosofo filosofo5=new Filosofo(mostrar5,t5,t1,c5);
            getContentPane().add(c5);
            c5.setBounds(305,150,50,50);
            c5.repaint();

            filosofo1.start();
            filosofo2.start();
            filosofo3.start();
            filosofo4.start();
            filosofo5.start();

        }

        public static void main(String args[])
        {
        Filosofos f=new Filosofos();
        f.setVisible(true);


        }
}
```

Figura 4.13: Código de filósofos.

En la figura 4.13 se muestra el segmento de código donde se aplican los diferentes estados para cada folósofo al momento de iniciar con la ejecucion.

4.3. Semafóros

El problema del productor-consumidor es un ejemplo clásico de sincronización de multiprocesos. El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío. La idea para la solución es la siguiente, ambos procesos (productor y consumidor) se ejecutan simultáneamente y se despiertan o duermen según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a dormir. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario si el buffer se vac´ıa, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Se puede encontrar una solución usando mecanismos de comunicación interprocesos, generalmente se usan semáforos. Una inadecuada implementación del problema puede terminar en un deadlock donde ambos procesos queden en espera de ser despertados. Este problema pude ser generalizado para múltiples consumidores y productores.

Interfaz gráfica

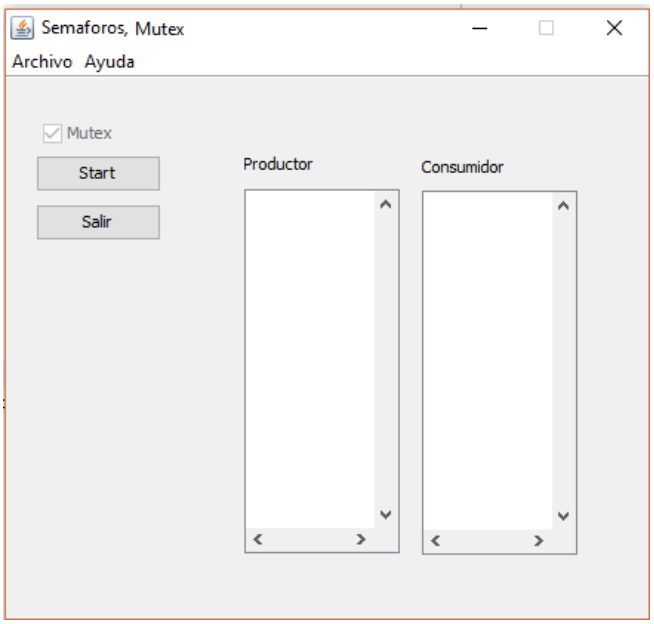


Figura 4.14: Primera vista.

En la figura 4.14 se presenta una aplicación concurrente donde se muestra la ejecución de dos procesos con el método de sincronización synchronized, que no es otra cosa que el metodo de sincronizacion monitor, encapsulado en un objeto que se encuentra en la clase Java.lang.

Fase experimental iniciando ejecución

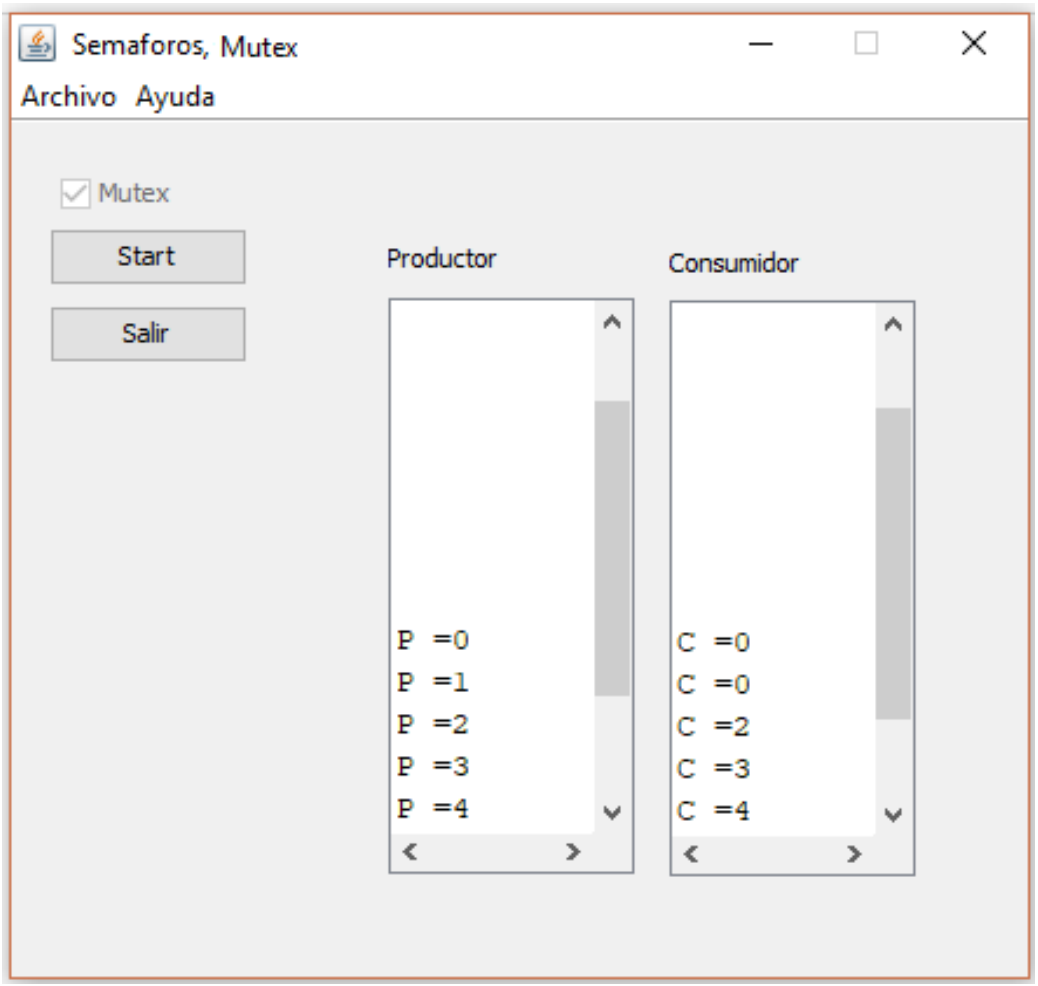


Figura 4.15: Iniciando ejecución.

En la figura 4.15 podemos notar la ejecución del programa y la manera en la que se van almacenando los productos en una área ilimitada o también conocido como un segmento de memoria la cual implica que solamente un proceso puede tener acceso a este recurso compartido, gracias al bloqueo que se realiza con el mutex.

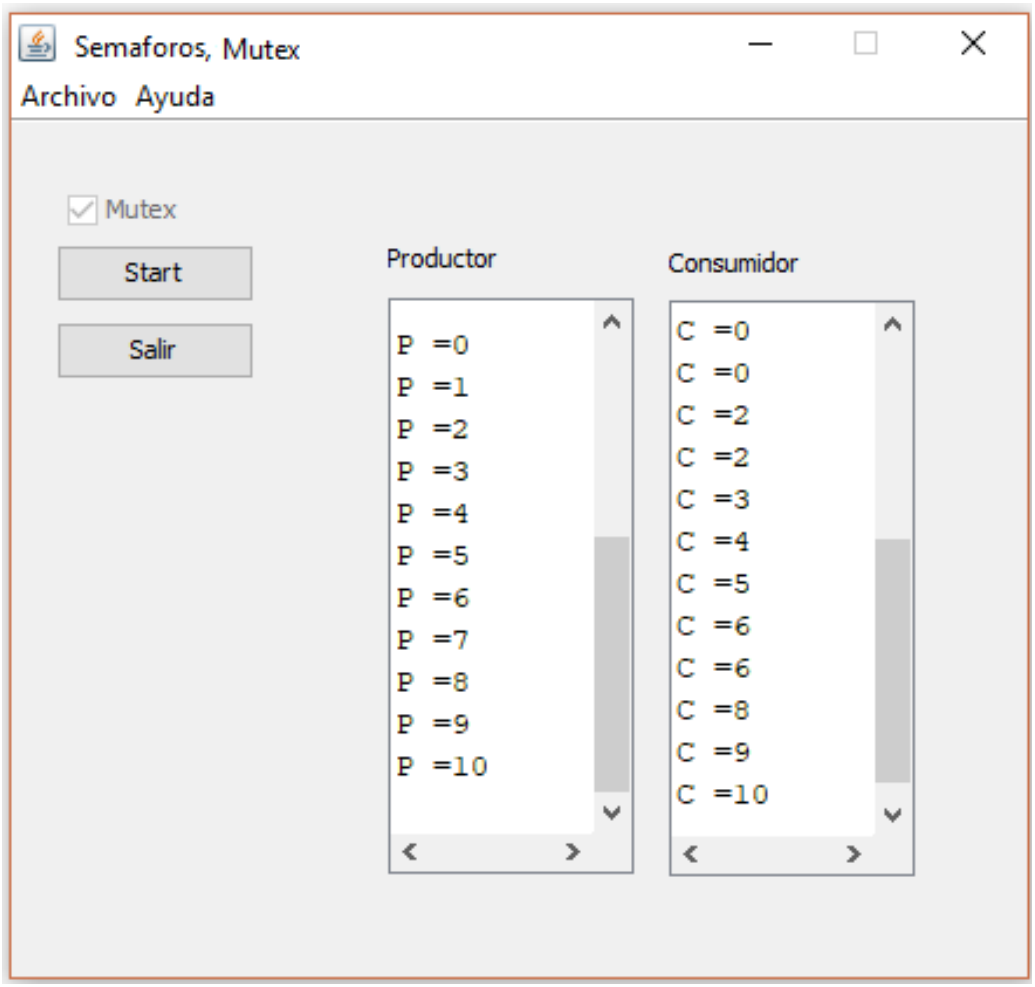


Figura 4.16: Ejemplo de funcionamiento.

En la figura 4.16 podemos ver la diferencia en lo resultados de almacenamiento tanto de productor como de consumidor el cual sucede por medio del método de mutex el cual evita las condiciones de competencia, por tal motivo este toma en cuenta el proceso que ya esté listo para poder ejercitarse sin importar si existe un orden o no.

```
final class Semaforo
{
        private int valor;
        private int esperando=0;
        public Semaforo(int i)
        {
                valor=i;

        }


        synchronized void DOWN()
        {
                if(valor>0)

                valor--;
                else
                {
                        esperando++;
                        try{wait();}catch(InterruptedException e){};

        esperando--;


                                        };
        }
```

Figura 4.17: Sincronización con semáforos.

En la figura 4.17 podemos notar que exister una clase llamda semaforo de la cual esperamos un valor y una sincronización para una posible interrupción.

```
public Consumidor(ObjetoBuffer dado,Semaforo sem1,Semaforo sem2,TextArea salida)
{

        um_Buffer=dado;
        s1=sem1;
        s2=sem2;
        this.salida=salida;

}
```

Figura 4.18: Procesos consumidos.

En la figura 4.18 conforme el programa se va describiendo podemos notar los procesos consumidos.

Estas instrucciones pueden modificarse para evitar la espera activa, haciendo que la operación P duerma al mismo proceso que la ejecuta si no puede decrementar el valor, mientras que la operación V despierta a un proceso que no es quien la ejecuta. En un pseudolenguaje más entendible, la operación P suele denominarse wait ó espera y la operación V signal ó señal.

El porqué de los nombres de estas funciones, V y P, tiene su origen en el idioma holandés. ”Verhogen”significa incrementar y ”proberen”probar, aunque Dijkstra usó la palabra inventada prolaag [1], que es una combinación de probeer te verlagen (intentar decrementar). El valor del semáforo es el número de unidades del recurso que están disponibles (si sólo hay un recurso, se utiliza un ”semáforo binarioçuyo valor inicial es 1).

La verificación y modificación del valor, así como la posibilidad de irse a dormir (bloquerse) se realiza en conjunto, como una sola e indivisible acción atómica. El sistema operativo garantiza que al iniciar una operación con un semáforo, ningún otro proceso puede tener acceso al semáforo hasta que la operación termine o se bloquee. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar condiciones de competencia.

Si hay n recursos, se inicializará el semáforo al número n. Así, cada proceso, al ir solicitando un recurso, verificará que el valor del semáforo sea mayor de 0; si es así es que existen recursos libres, seguidamente acaparará el recurso y decrementará el valor del semáforo.

Cuando el semáforo alcance el valor 0, significará que todos los recursos están siendo utilizados, y los procesos que quieran solicitar un recurso deberán esperar a que el semáforo sea positivo, esto es: alguno de los procesos que están usando los recursos habrá terminado con él e incrementará el semáforo con un signal o V(s).

4.4. Peer to peer

- Se debe usar Multicast para la comunicación entre Peers.
- Se debe contar con una interfaz sencilla para enviar el mensaje y mostrar los mensajes recibidos.
- Todos los miembros unidos al grupo Multicast pueden leer los mensajes.
- No se necesita estar unido obligatoriamente al grupo Multicast para enviar mensajes.

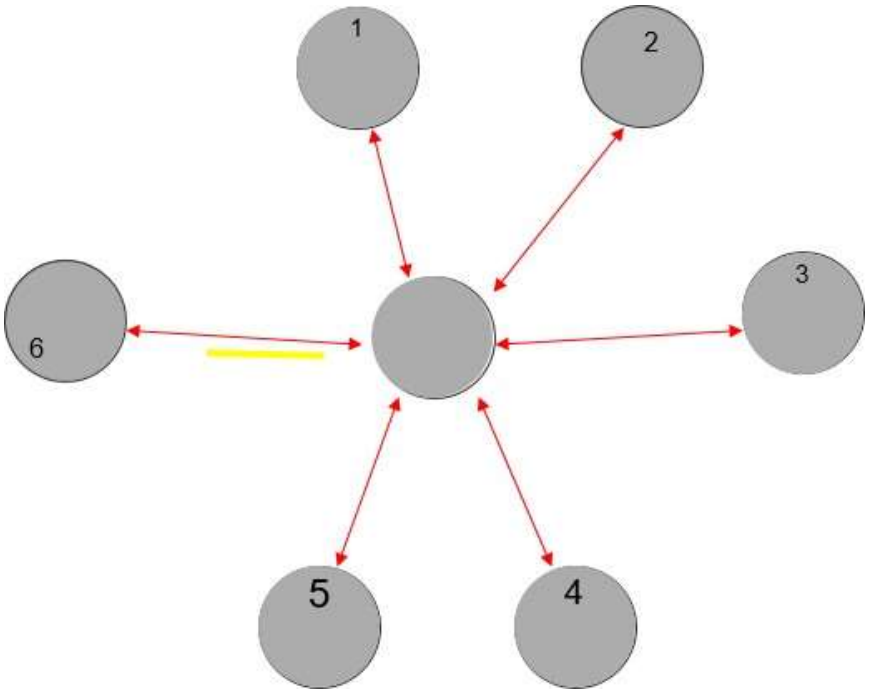


Figura 4.19: Ejemplo de un peer.

Demostración NetBeans 8.2 / Java jdk 1.8.

```
public void enviar_Mensaje(String msj){
    this.mensaje = msj.getBytes();
    this.paquete = new DatagramPacket(this.mensaje,this.mensaje.length,host,port);
    try {
        this.socket.send(paquete);
        System.out.println("Mensaje enviado");
    } catch (IOException ex) {
    }
}
```

Figura 4.20: Cadena de caracteres grupo Multicast.

En la figura 4.20 recibimos una cadena de caracteres y la enviamos al grupo Multicast.

```
@Override
public void run() {
    String mensaje_Recibido="";
    try {
            this.socket.joinGroup(host);

            while(true){

                this.paquete = new DatagramPacket(this.msj_Enviado,this.msj_Enviado.length);
                synchronized(this){
                    this.socket.receive(paquete);
                }
                mensaje_Recibido+= new String(paquete.getData()) + "\n";
                //System.out.println("Recibio: "+mensaje_Recibido);
                this.area.setText(mensaje_Recibido);

            }
    } catch (IOException ex) {

    }
}
```

Figura 4.21: El monitoreo.

En la figura 4.21 se mantiene un monitoreo constante de los mensajes transmitidos en el Multicast sin tiempo de espera definido, se espera por la recepción de un mensaje.

Interfaz gráfica.

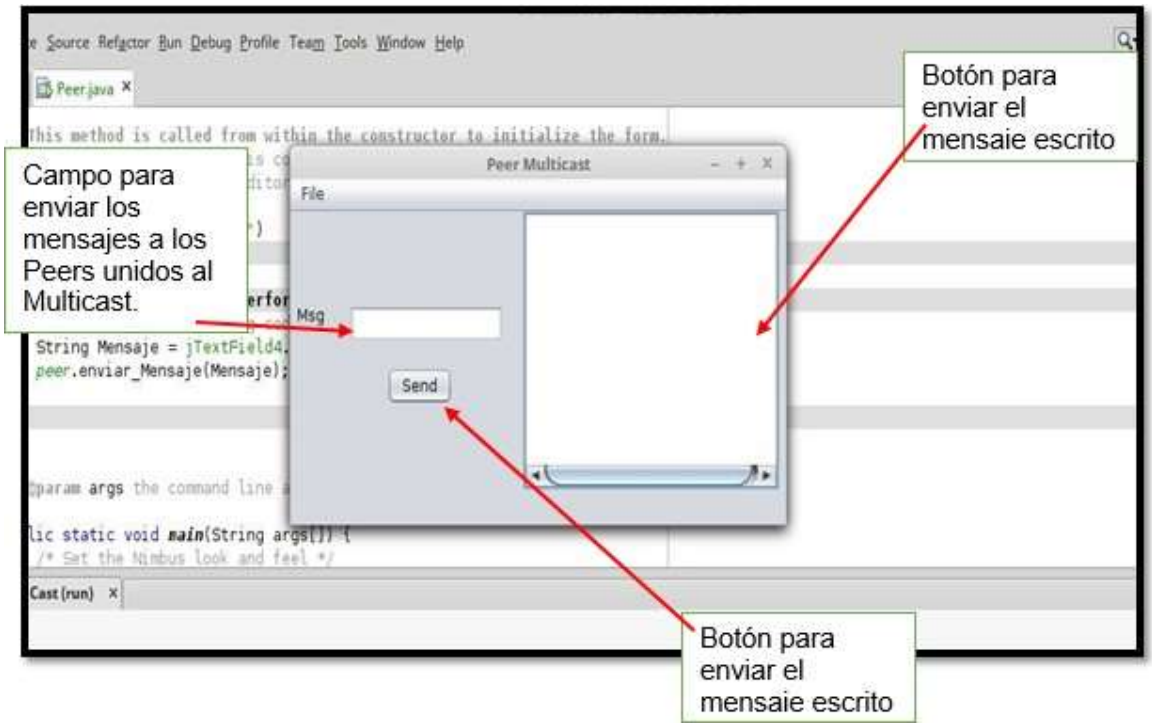


Figura 4.22: Caracteres enviados al grupo Multicast.

En la figura 4.22 recibimos una cadena de caracteres y la enviamos al grupo Multicast. Podemos notar las partes que conforman la interfaz, campo para enviar los mensajes a los peers unidos en el multicast, la ventana para escribir el mensaje a enviar y por último el botón para poder enviar el mensaje.

Fase experimental.

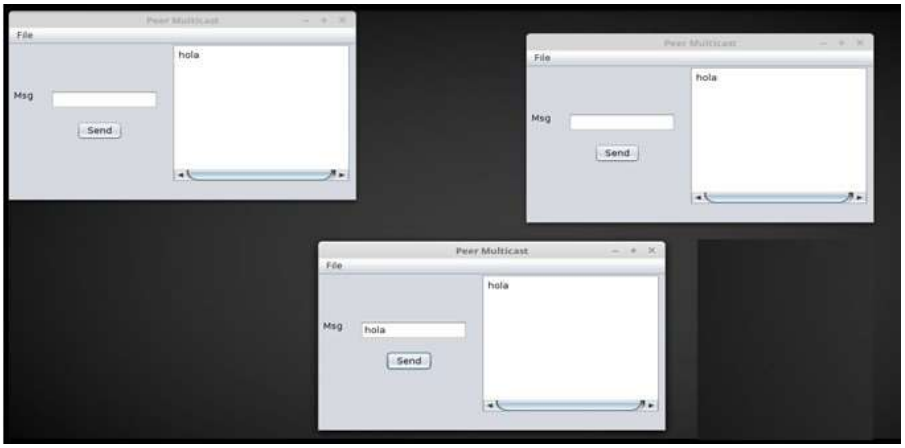


Figura 4.23: Ventanas para iniciar chat.

En la figura 4.23 podemos notar que el programa ya esta en ejecucion en ella vemos tres ventanas abiertas listas para enviar y recibir mensajes, la primera prueba es que una de las ventanas fue la que envió el primer mensaje y las otras dos disponibles solo recibieron dicho mensaje.

Ejemplo de funcionamiento

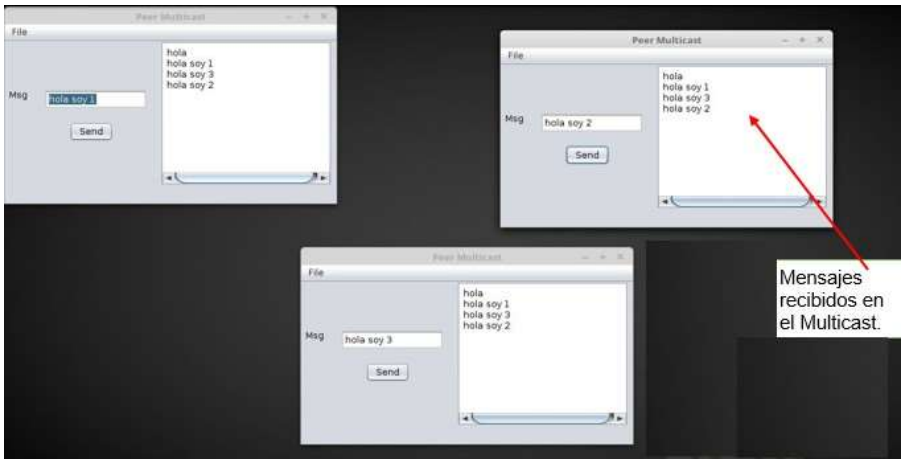


Figura 4.24: Paso de mensajes.

En la figura 4.24 podemos notar que conforme avanza la ejecución del programa los valores en las ventanas van cambiando, pues ahora todas las ventanas mandan mensaje y reciben mensajes. Los mensajes los podemos ver en la parte blanca de la ventana, pues ahi se nota claramente la información enviada.

4.5. Multicast - lectores / escritores

- Se tienen varios Peer en acceso a una estructura compartida, de la cual necesitan recuperar información o modificarla bajo el siguiente principio.
- Varios lectores pueden acceder de forma concurrente al recurso.
- Un Escritor puede acceder al recurso solo si no hay lectores u escritor usando el recurso.

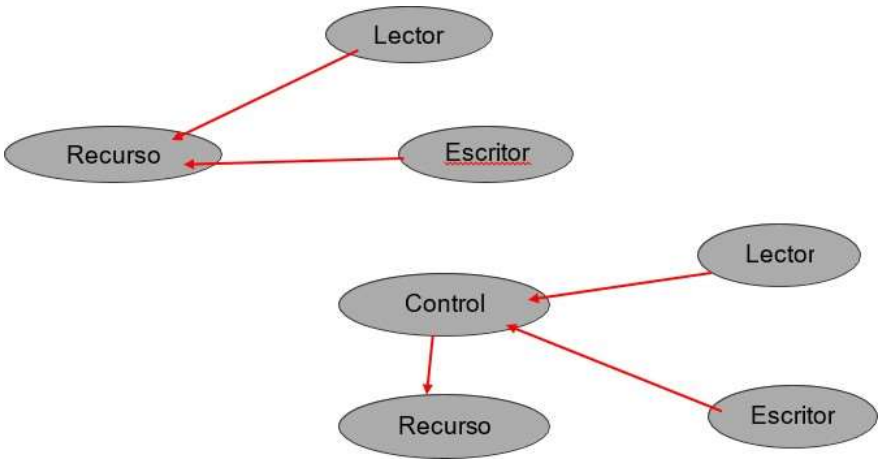


Figura 4.25: Ejemplo1.

- Las peticiones de entrada y salida se env´ıan al grupo Multicast.
- Cada Peer rescata las peticiones y las clasifica en la respectiva lista.
- Se extrae una petición de la lista y analiza el equipo de acuerdo al algoritmo principal.
- Se extrae una petición de la lista de salidas para liberar el peer del recurso previamente autorizado.

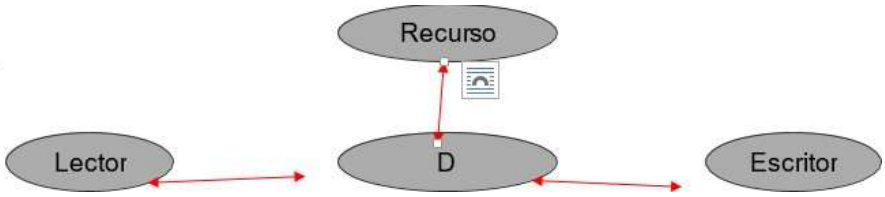


Figura 4.26: Ejemplo2.

```
if(tipo.equalsIgnoreCase("Lector")){
    System.out.println("Es un Lector");
    this.entrada="ENTERREAD "+id;
    this.salida="EXITREAD "+id;
    this.Lector=true;
}
if(tipo.equalsIgnoreCase("Escritor")){
    System.out.println("Es un Escritor");
    this.entrada="ENTERWRITE "+id;
    this.salida="EXITWRITE "+id;
    this.Escritor=true;
}
if(tipo.equalsIgnoreCase("Coordinador")){
    System.out.println("Es un coordinador");
    this.Coordinador=true;
}
```

Figura 4.27: Tipos de peer.

En la figura 4.27 y 4.28 el constructor de la clase definimos el tipo de Peer que se manejará. También se instancia el tipo de mensaje para petición de entrada o salida. Recibimos las peticiones, clasificamos el tipo de mensaje de acuerdo a si es una petición de entrada o petición de salida en 2 estructuras distintas.

```
if(this.Coordinador)
{
    while(this.Coordinador){
        buffer = new byte[20];
        paquete = new DatagramPacket(buffer,buffer.length);
        try{
        socket.receive(paquete);
        p = new Peticion(paquete);
        if(p.is_In()){
            this.s_Entrada.add(p);
        }
        if (p.is_Out()){
            this.s_Salida.add(p);
        }
        }catch(IOException ex){

        }
        Thread.sleep(sleep);
    }
```

Figura 4.28: Recepción de peticiones y clasificación.

Algoritmo principal lectores - escritores.

En la figura 4.29 se nota como extraemos un elemento de las solicitudes de entrada. Si es un lector y no tenemos Escritores usando el recurso autorizamos el ingreso. Si es un Escritor y no existen lectores como escritores usando actualmente el recurso autorizamos el ingreso. De otra forma Mantenemos la petición en espera. Cuando la lista de las peticiones que usan el recurso no este vac´ıa mostramos lo elementos actuales en el recurso.

En la figura 4.30 extraemos una petición de las soluciones de salida, corroboramos que se encuentre ya autorizada en el recurso. Buscamos y retiramos la petición de la estructura que manifieste las peticiones en espera.

```
if(!this.s_Entrada.isEmpty()){

    a=(Peticion)this.s_Entrada.get(0);
    do{
       this.Recurso.cont_Recurso();
       if(a.is_In_L() && Recurso.get_Escritores()==0){
            this.Recurso.add(a);
            this.s_Entrada.remove(0);
            this.notificacion=false;
            mensaje+=("Entra: "+a+"\n");
            this.ar1.setText(mensaje);
            a=null;
        }
        else if(a.is_In_E() && Recurso.get_Escritores()==0 && Recurso.get_Lectores()==0){
            this.Recurso.add(a);
            this.s_Entrada.remove(0);
            this.notificacion=false;
            mensaje+=("Entra: "+a+"\n");
            this.ar1.setText(mensaje);
            a=null;
        }
        if(a!=null)
        {
            if(!this.Recurso.isEmpty()){
                mensaje+=("Elementos usando el recurso\n");
                mensaje+=(this.view_list(this.Recurso));
            }
            if(!this.s_Entrada.isEmpty()){
            mensaje+=("Elementos en espera del Recurso\n");
            mensaje+=this.view_list(s_Entrada);
            }
            if(this.Recurso.isEmpty() && this.s_Entrada.isEmpty()){
                mensaje="";
            }
            this.ar1.setText(mensaje);
        }
```

Figura 4.29: Algoritmo principal.

```
if(!this.s_Salida.isEmpty()){
    s = (Peticion)this.s_Salida.removeFirst();
    int b=(this.Recurso.busqueda(s));

    if(b!=-1){
        mensaje+=("Sale: "+s+"\n");
        System.out.println("Sale: "+s);
        this.Recurso.remove(b);
        this.ar1.setText(mensaje);
    }


}
```

Figura 4.30: Peticiones de salida.

Fase experimental

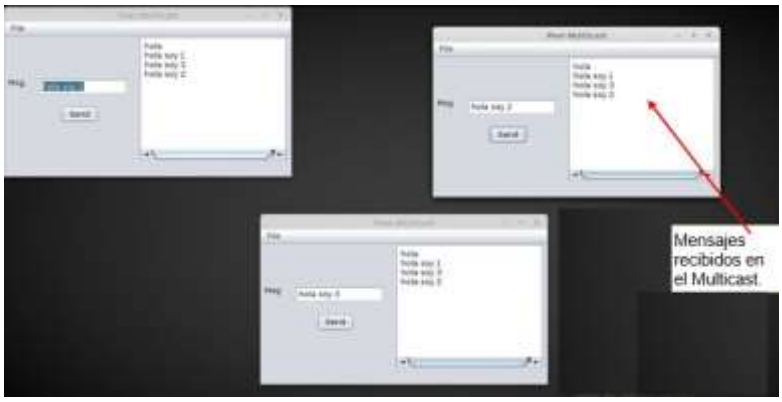


Figura 4.31: Interfaz gráfica.

Por fines estéticos tomaremos exclusivamente un Peer como se nota en la figura 4.31 un coordinador para mostrar más limpiamente el funcionamiento adecuado. Esta Sección puede ser Mostrada al activar las variables booleanas respectivas en el código para mostrar el proceso en cada Peer. Las estructuras de datos usadas para almacenar las peticiones también cumplen la funciona de una cola de datos (FIFO) para no tener preferencia sobre las entradas y atender las peticiones en el orden de ingreso.

Figura 4.32: Petición enviada.

En la figura 4.32 tenemos tres ventanas diferentes de las cuales sólo notamos el trabajo de dos de ellas. una manda una petición y otra de acepta la petición.

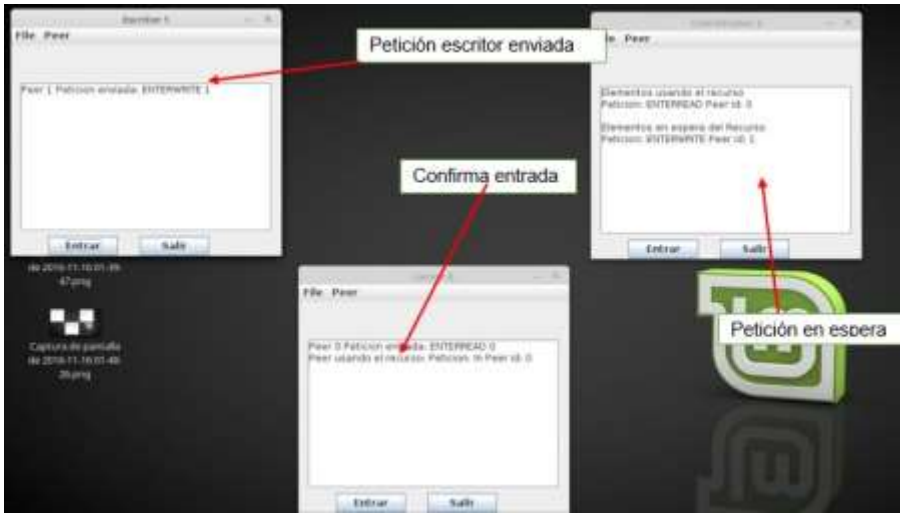


Figura 4.33: Confirmación de entrada.

En la figura 4.33 a comparación de la imagen anterior aquí sí notaremos el trabajo de ejecución de las tres ventanas. Partiendo de la ejecución anterior aquí notamos cuando la ventana tres la faltante comienza a ejecutarse iniciando con una petición a las dos ventanas extras las cuales confirman la petición.

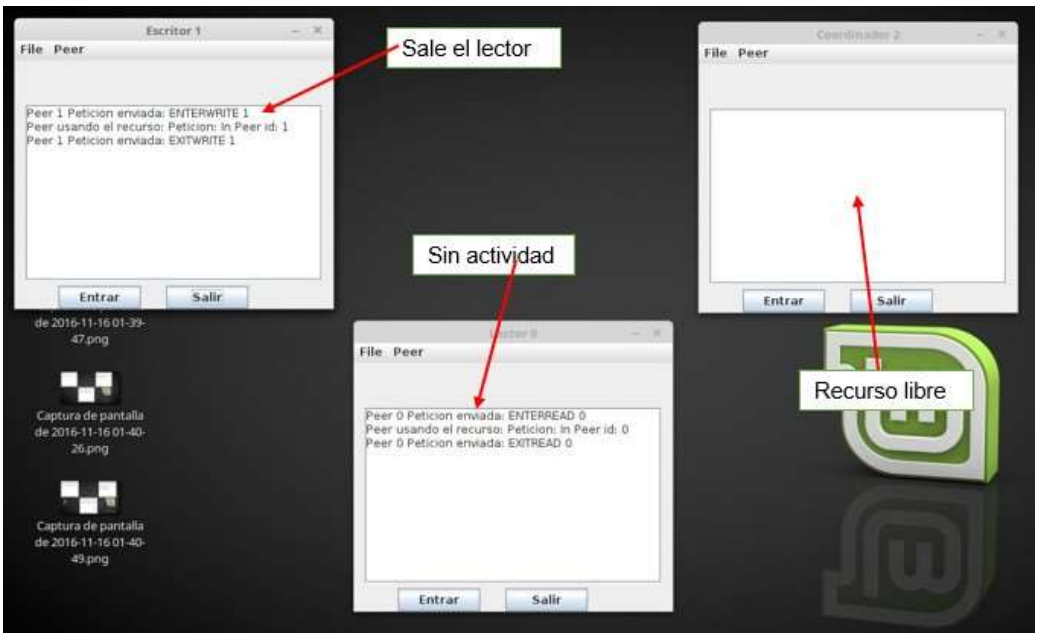


Figura 4.34: Finaliza petición.

4.6. Multicast - bully

4.6.1. Bully monitoreo.

En una red de varios ordenadores conectados entre sí, donde a cada uno denominaremos Peer. Un Peer de la red tiene la función de ser coordinador enviando un mensaje cada cierto tiempo al resto de Peers que no son coordinadores. Si un Peer no es Coordinador y detecta que el Coordinador actual fallo inicia un proceso para seleccionar a un nuevo Coordinador.

4.6.2. Bully elección

Un Peer emite un mensaje para iniciar el proceso de elección esperando como respuesta un ok confirmando. Si el Peer no recibe algún mensaje de confirmación por un tiempo determinado entonces se declara como Coordinador e inicia el proceso de enviar los mensajes cada cierto tiempo notificando el resultado de la elección. Si el Peer recibe un mensaje de eleccion de un Peer con un id mayor este se queda a la espera del nuevo coordinador. Si recibe un mensaje de elección de un Peer con un id menor, envía un mensaje de respuesta ok e inicia de nuevo el proceso de elección.

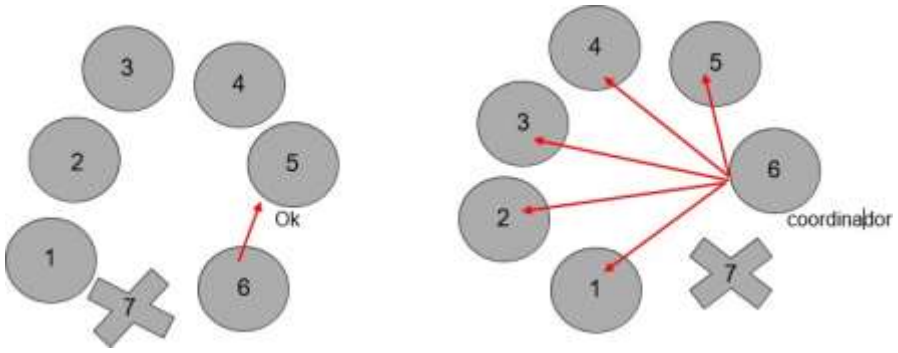


Figura 4.35: Ejemplo.

4.6.3. Bully multicast

Método para la difusión de mensajes a múltiples destinos, se reservan las redes de tipo D para la multidifusión en el rango 224.0.0.0 a 239.255.255.255.

Peers monitorean al coordinador, en caso de fallo se inicia la elección.

Los Peers leen el mensaje de elección con el id que detectó la caída.

Si leen un mensaje de elección que proviene de un peer con id mayor se quedan a la espera del nuevo coordinador.

si lee un mensaje de elección con su mismo id este incrementa un contador, para simular tiempo de espera.

cuando el id recibe más de 2 mensajes con su mismo id, simulando un tiempo de espera de 3 segundos, se declara Coordinador.

Tipo de mensaje - coordinador

```
/**
 * Se envia el mensaje para informar a todos los peer que es el coordinador actual
 * @param id identificador del peer que se declara coordinador.
 */
private void msg_Coordinador(int id){
    byte buffer []= ("Coordinador "+id).getBytes();
    DatagramPacket paquete = new DatagramPacket(buffer,buffer.length,host,port);
    try {
        socket.send(paquete);
    } catch (IOException ex) {
        Logger.getLogger(Peer.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Figura 4.36: Mensaje de coordinador.

En la figura 4.36 se nota que el Mensaje Coordinador, notifica que el coordinador está activo. Entonces se envía la cadena de caracteres Coordinador + id del mismo.

Tipo de mensaje - elección

- Se envía una cadena de caracteres Elección + id del Peer.

```
/**
 * Se envia el mensaje para el proceso de eleccion a todos los peer informando
 * que es un posible candidato a coordinador
 * @param id identificador del peer que pretende ser coordinador
 */
private void msg_Eleccion(int id){
    byte buffer[] = ("Eleccion "+id).getBytes();
    DatagramPacket paquete = new DatagramPacket(buffer,buffer.length,host,port);
    try {
        socket.send(paquete);
    } catch (IOException ex) {
        Logger.getLogger(Peer.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Figura 4.37: Mensaje de elección.

Envio - coordinador

```
if(Coordinador){
    //System.out.println("Soy el coordinador "+id);
    msg_Coordinador(id);
        mensaje+=("\nSoy el coordinador "+id+"\n");
        if(ar1!=null)
            ar1.setText(mensaje);
}
```

Figura 4.38: Mensaje de eleccion.

En la figura 4.38 Se utiliza el método para enviar el mensaje coordinador a los peer que no son coordinadores.

Recepción - no coordinador

en la figura 4.39 Recibimos los mensajes del grupo Multicast al que estamos anexados. El conjunto de bytes recibidos serán convertidos a una cadena que dividiremos en su valor entero y el contexto del mensaje que analizaremos mensaje Coordinador/Elección.

```
if(!Coordinador){
    try {

        byte buffer[]= new byte[20];
        paquete=new DatagramPacket(buffer,buffer.length);
        socket.setSoTimeout(3000);
        socket.receive(paquete);

        pack = to_Split_Datagram(paquete.getData());

        String msg = String.valueOf(pack.get(0));
        int id_rec = Integer.parseInt(String.valueOf(pack.get(1)));
```

Figura 4.39: Recepción no coordinador.

Coordinador activo

```
if(msg.equalsIgnoreCase("Coordinador")){
    Elector_lock=false;
    //System.out.println("El coordinador actual es: "+id_rec+" soy: "+id);
        mensaje+=("El coordinador actual es: "+id_rec+" soy: "+id+"\n");
        if(ar1!=null)
            ar1.setText(mensaje);

}
```

Figura 4.40: Mensaje coordinador.

En la figura 4.40 Si el tipo de mensaje es la cadena Coordinador simplemente mostramos el id del Peer que es coordinador y el id del peer que lo recibe.

4.6.4. Preparando el entorno experimental (Scrip Bash)

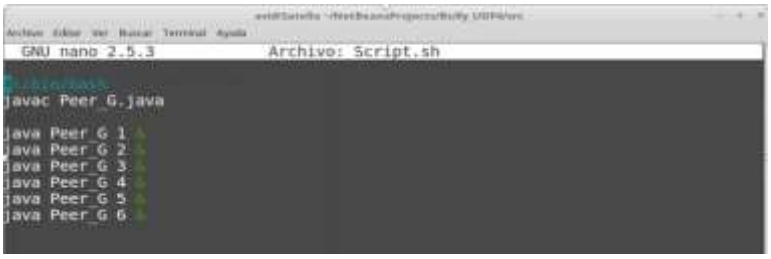


Figura 4.41: Primera corrida.

En la figura 4.41 se crea un archivo en el entorno GNU/Linux mediante el intérprete de comandos bash. El Script estará encargado de compilar las clases .java y ejecutar el archivo Peer G agregando un argumento del tipo entero para el id del peer. Consecutivamente será acompañado de un símbolo & ampersand, en entornos Linux este símbolo se usa para que un proceso, al momento de ser lanzado se vuelvan daremos y sean independientes del proceso padre que los crea, de esta forma podemos cerrar cada peer una vez ejecutado para simular la caída del coordinador. Deshabilitar Firewall y Conectarse a una red con un Router para el direccionamiento Multicast.

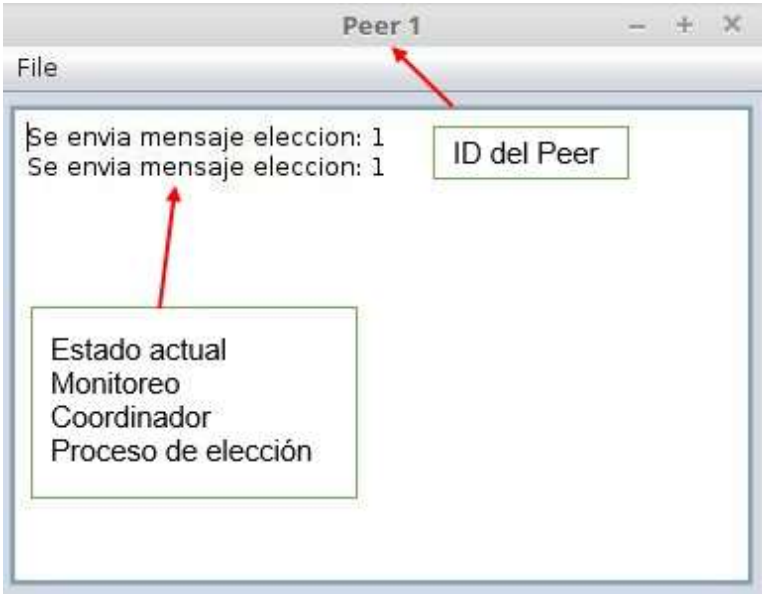


Figura 4.42: Vista de ventana.

En la figura 4.42 es un ejemplo de la vista de la ventana al ejecutarse el programa, en la cual podemos notar que como es enviado el mensaje y tambien el estado actual del mismo, como informacion estra el ID del peer.

Probando 6 PEER´S

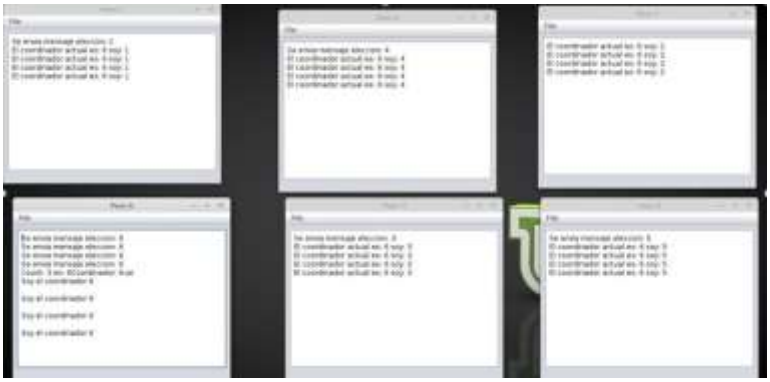


Figura 4.43: Corrida.

En la figura 4.43 podemos notar la ejecucion del programa con 6 peer´s teniedo una conexion entre ellos, enviandose mensajes, piendo permisos y siendo coordinador.

Finalizando fase experimental

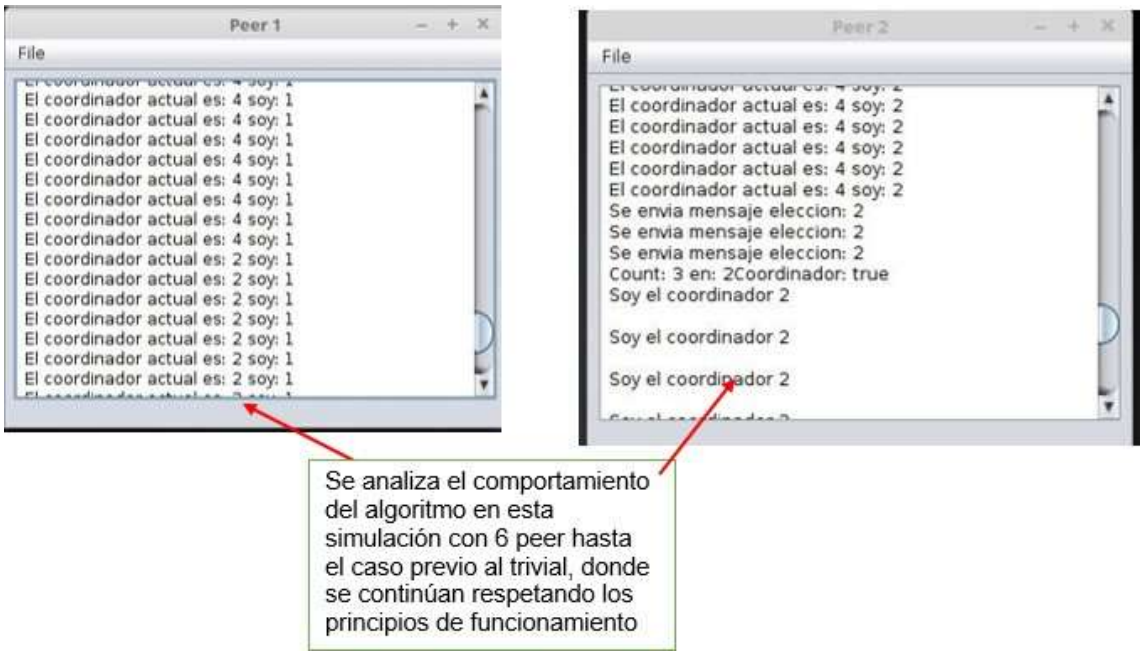


Figura 4.44: Comportamiento de la simulación hasta con 6 peer´s.

En la figura 4.44 podemos notar la ejecucion del programa con 6 peer´s teniedo una conexion entre ellos, en este se analiza el comportamiento del algorimo con 6 peer´s hasta el caso previo al trivial, donde se continuan respetando los principios de funcionamiento.

Resultados

Portada del libro ya existente ”Programación Concurrente y Paralela”.

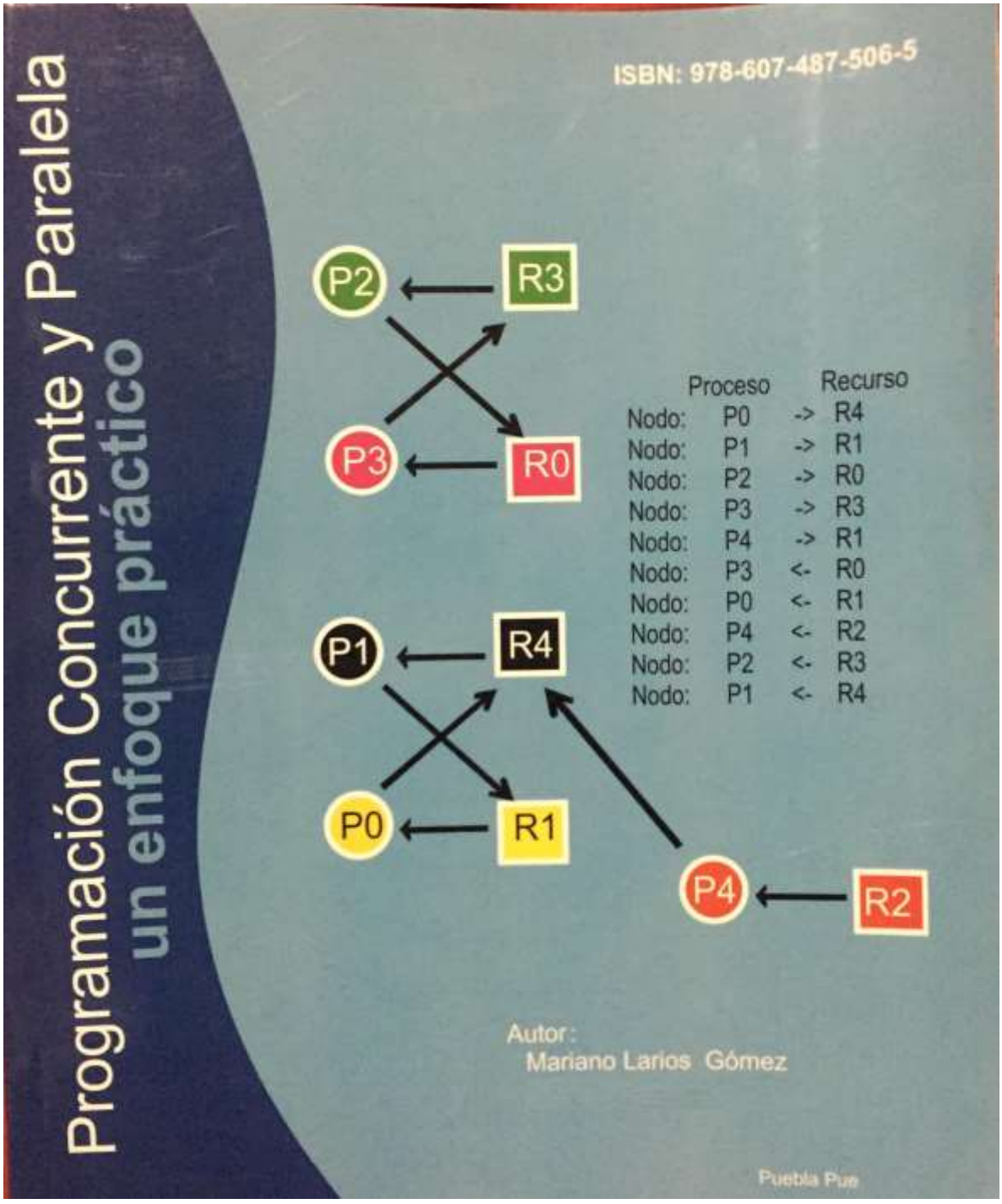


Figura 4.45: Portada simulada del libro base.

Portada del posible libro ”Programación Paralela: Un enfoque práctico.”

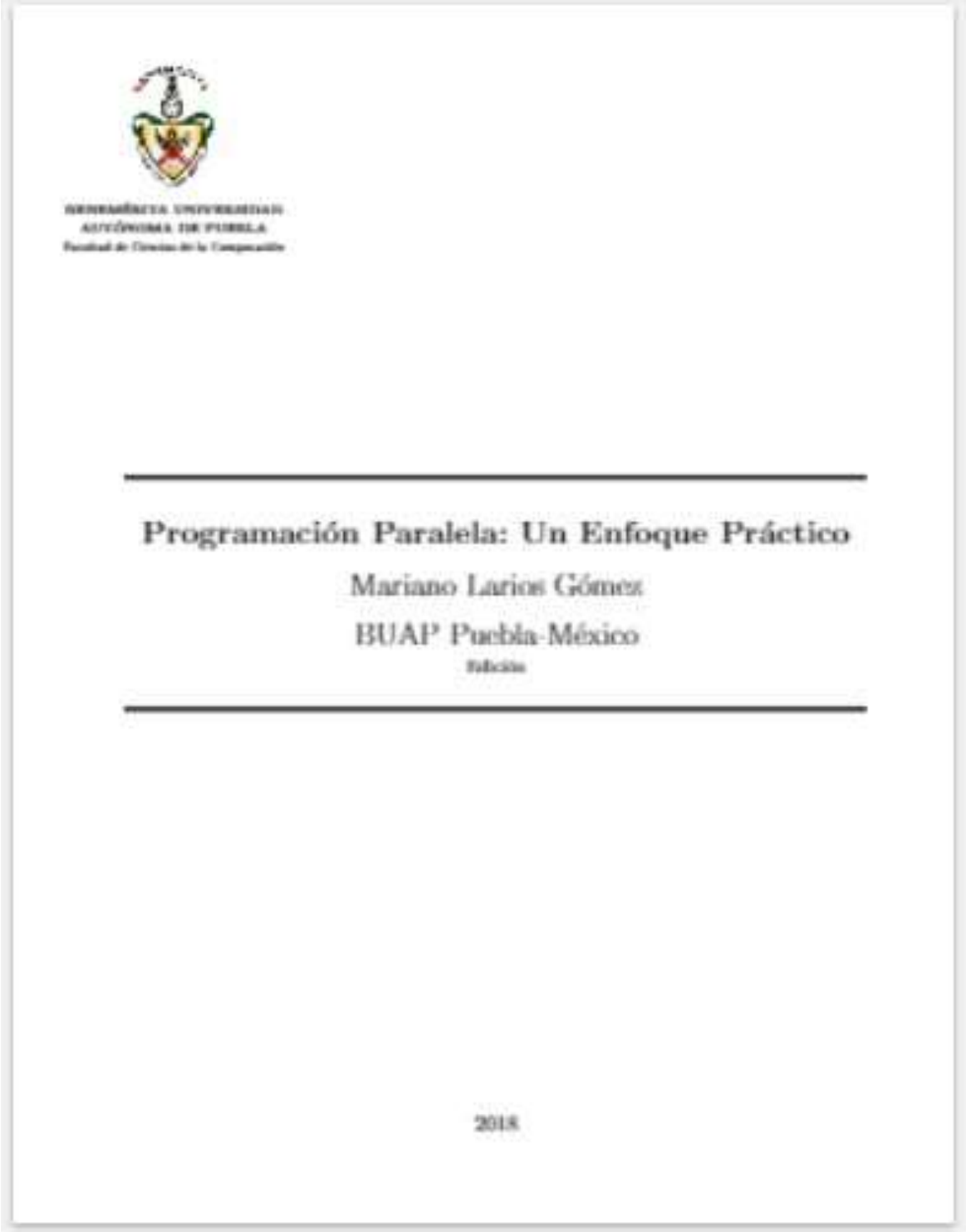


Figura 4.46: Primera portada simulada.

Portada del posible libro ”Programación Concurrente: Un enfoque práctico.”

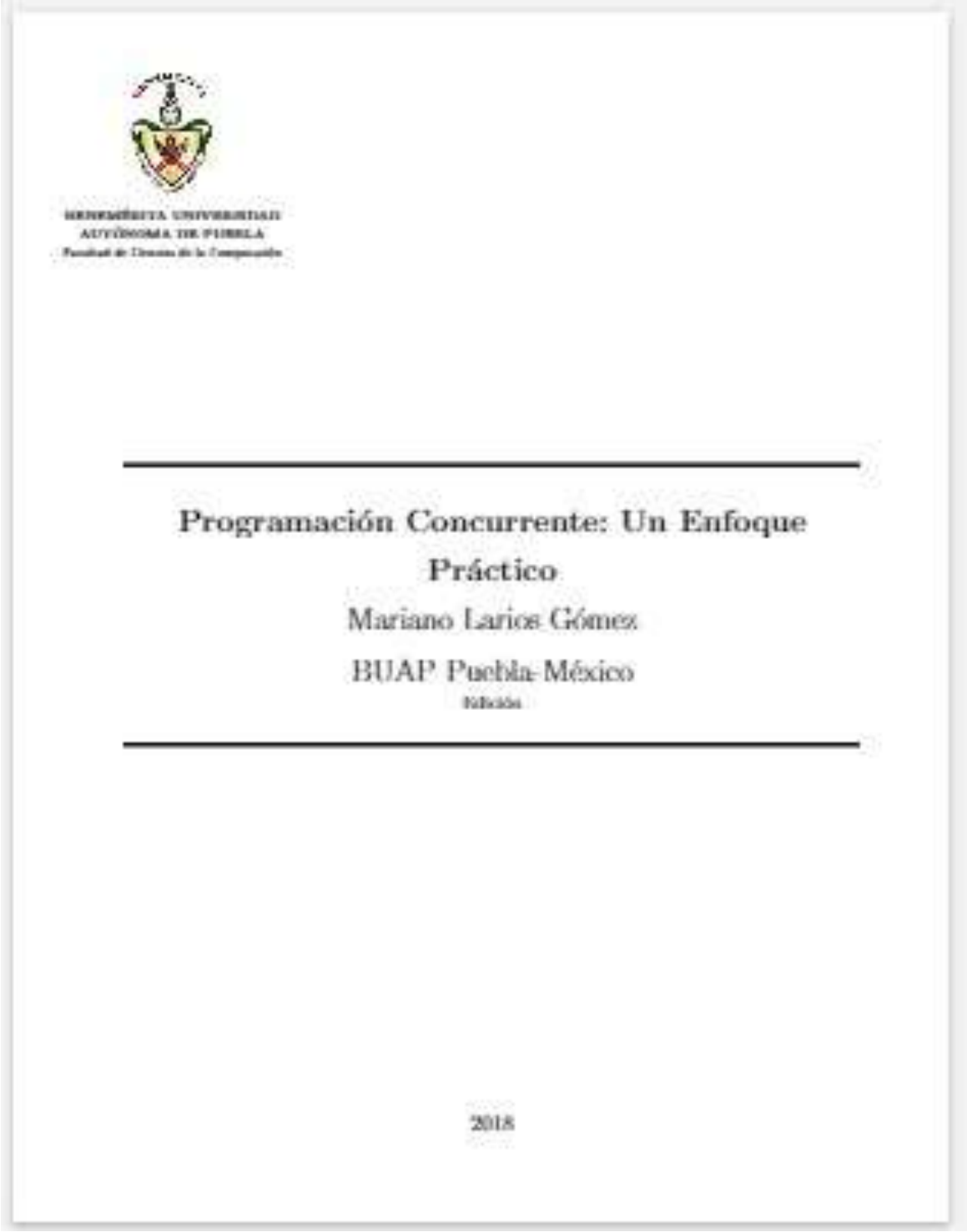


Figura 4.47: Segunda portada simulada.

Conclusión

En este trabajo de tesis de investigación concluimos en que aplicar el paradigma de cómputo distribuido y cómputo concurrente se necesita de altos conocimientos de cómputo, y de ciencias dela computación. Esto debido a que se utilizan lenguajes de alto nivel orientados a objetos y eventos, sistemas operativos multicapas o máquinas virtuales embebidas en los S.O. os conceptos como peer-to-peer cambia por completo la forma de pensar, porque se nace con un concepto centralizado en los sistemas computacionales y en los sistemas de la vida diaria, por ejemplo el sistema de política, el sistema educativo y de trabajo. los algoritmos distribuidos como el del grandulón, avestruz, lamport, consenso, y los bizantinos son muy difícil de aplicarlos en redes tipos grids y más aún con supercómputo HPC, entre otros. este trabajo propone otro forma de explicar cómo es que funciona las aplicaciones distribuidas y concurrentes, hace más entendible a estudiantes de ciencias, de ingenier´ıa y de posgrado. La aportacion esta dirigida a dar material entendible e ilustrado para aprender estos dos paradigmas complejos y abstractos.

Bibliografía

[1]Larios G. M., Migliolo C. J., Anzures G. M., Aldama D. A., Trinidad G. G., *"A Schediluing Algorithm for a platform in real time",* 9th International Supercomputing Conference in Mexico. Springer International Publishing AG.(2018).

[2]M. Larios-Gómez, L. A. Zamarripa Almazan, A. Hernández B., E. A. Martinez-Mirón. *"Implementation of a visual interface for a platform that applies processes planning algorithms",*Bill Lewis, Daniel J. Berg

[3]Laboratorio Nacional de Supercómputo del Sureste de México *"Laboratorio Nacional de Supercómputo del Sureste de México.",* 2018, de BUAP Sitio web:http://lns.org.mx/

*[4]*Courtois P. J. and Parnas D. L., *"Concurrent Control with Readers.ª nd "Writers".",* Communications of the ACM Vol. 14, No. 10. October 1971, pp. 667-668.

[5]L. Besnard, A. Bouakaz, T. Gautier, P. Le Guernic, Y. Ma, J.-P. Talpin, H. Yu. *"Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony",* In Science of Computer Programming (SCP, core rank A). Elsevier, 2014.

[6]Bouakaz and J.-P. Talpin. International Workshop on Software and Compilers for Embedded Systems *"Design of Safety-Critical Java Level 1 Applications Using Affine Abstract Clocks",*(SCOPES'13, core rank A). ACM, June 2013.

[7]A. Bouakaz and J.-P. Talpin. Conference on Languages, Compilers and Tools for Embedded Systems *"Buffer minimization in earliest-deadline first scheduling of dataflow graphs",* (LCTES'13, core rank A). ACM, June 2013

[8]. Ma, H. Yu, T. Gautier, L. Besnard, P. Le Guernic, J.-P. Talpin and Maurice Heitz.Design Analysis and Test in Europe *"Toward polychronous analysis and validation for timed software architectures in AADL",* (DATE'13, core rank A). IEEE, April 2013.

[9]A. Bouakaz, J.-P. Talpin, and J. Vitek. Application of Concurrency to System Design (ACSD'12). EEE Press, June 2012. *"Affine data-flow graphs for the synthesis of hard real-time applications",*

[10]Albert M. K. Cheng. Real-Time Systems: Scheduling, Analysis, and Verification. August 2002.

[11]J.A. Stankovic; M. Spuri; M. Di Natale; G.C. Buttazzo. *"Implications of Classical Scheduling Results for Real-Time Systems.",* Browse Journals Magazines Computer Volume: 28 Issue: 6. August 2002.

[12] Sengul Cigdem and Robin Kravets. Bypass routing: An on-deman local recovery protocol for Ad Hoc netwoks. Volume 4 Issue 3, May, 2006 Pages 380-397 Elsevier Science Publishers B. V. Amsterdam, The Netherlands, The Netherlands.

[13] Ramasubramanian, V., Haas, Z. J., Sirer, E. G. (2003, June). SHARP: A hybrid adaptive routing protocol for mobile ad hoc networks. *"In Proceedings of the 4th ACM international symposium on Mobile ad hoc networking computing .",* (pp. 303-314). ACM.

[14] Director de Tesis: Dr. Héctor Benítez Pérez, Estudiante: Magali Arellano Vázquez, Posgrado en Ciencias e Ingeniería de la Computación, UNAM, *"Estudio de Sistemas Adaptables de Encaminamiento para Sistemas Distribuidos Móviles.",* Fecha de graduación 21 de Septiembre de 2015.

[15]A. S. Tanenbaum *"Modern Operiting System (3ra Edition)",* Pretinces Hall (1995)

[16]E. W. Dijkstra. *"Hierarchical ordering of sequential processes",* Acta Informática 1(2): 115-138. 1971.

[17]E. W. Dijkstra. *"Solution of a Problem in Concurrente Programming Control",* Communications of the ACM Vol. 5 No. 9 September, 1965.

[18]Patil S. S. *"Limitation and capabilities of Dijkstra's semaphore primitives for coordination among processes",* Proj. MAC, Computational Structures Group Memo 57, Feb 1971.

[19]C.A.R Hoare *"Monitors: An Operating System Structuring Concept",* Communication of the ACM Vol. 17, No. 10. October 1974, pp. 549-557.

[20] Holt R. C. *"Some Deadlock Properties of Computer System",* Computing Surveys,Vol. 4, pp. 179-196, September 1972

[21] Naiqi Wu. *"Necessary and Sufficient Conditions for Deadlock-Free Operation in Flexible Manufacturing Systems Using a Colored Petri Net Model",* IEEETRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICSPART C: APPLICATIONS AND REVIEWS, VOL. 29, NO. 2, MAY 1999.

[22] ZhiJun Ding, ChangJun Jiang, and MengChu Zhou. *"Deadlock Checking for One-Place Unbounded Petri Nets Based on Modified Reachability Trees",* IEEE TRANSACTIONS ON SYSTEMS, AN, AND CYBERNETICSPART B: CYBERNETICS, VOL. 38, NO. 3, JUNE 2008.

[23] Coffman E. G. *"System Deadlocks",* Commputing Surveys. Vol. 3, No. 2 June 1971, pp. 68-78.

[24] Parnas D.L. *"On Solution the Cigarette Smoker's Problem (without conditional statements)",* Communication of the ACM Vol. 18, No. 3. March 1975, pp. 181-183.

[25] Larios G. M., Salazar M. H., Vázquez Z. A., David Z. C., Italo José Cortéz. *"Deadlock Detection Algorithm with Coloring Graphs",* Congreso Nacional de Ciencias de la Computación. CONCIC 2008. ISBN: EN TRAMITE. pp. 68-73 2008

[26] Keir Fraser, Tim Harris. *"Concurrent Programming Without Locks",* ACM Transactions on Computer Systems, Vol. 25, No. 2, Article 5, Publication date: May 2007.

[27]M. L. Liu. *"Distributed Computing: Principles and Applications ",* Addison Wesley 2004.

[28] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar *"Introduccion to Parallel Computing",*Pearson, Addison Wesley (2003)

[29] Flynn *"Taxonompia",*

[30] Andrew S. Tanenbaum, *"Organización de computadoras",*Person (2000).

[31] Cay S. Horstman, Gary Cornell. *"Core Java 2. Volumen I Fundamentos",* Sun Microsystem Pearson Pretinces-Hall(2005).

[32] Selim G. AKL. *"Parallel Computation",* Pretinces-Hall 1997.

[33] Vijay K. Garg *"Concurrent and Distributed Computing in Java",*Wiley Inter-Science (2004).

[34] Doug Lea *"Concurrent Programming in Java. Design Principes and Patterns",* Sun Microsystem, Addison Wesley (2003).

[35] Oracle *"Oracle ".*

TESISTA

ALEJANDRA RUFINO ALONSO

PARA OBTENER EL GRADO DE

INGENIERO EN CIENCIAS DE LA COMPUTACIÓN

ASESOR

M.C. MARIANO LARIOS GÓMEZ

BUAP, FCC