BENEMERITA UNIVERSIDAD AUTONOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACION

"DISEÑO E IMPLEMENTACION DE UNA UNIDAD DE MEMORIA VIRTUAL COMPARTIDA CON MONITOREO (MEMVIR)"

TESIS

PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS DE LA COMPUTACION

PRESENTA

IRENE GARCIA ORTEGA

ASESOR

M.C. GRACIANO CRUZ ALMANZA

DICIEMBRE 2004

AGRADECIMIENTOS

A Dios por darme la vida y la capacidad para poder realizar y culminar este proyecto.

A mis padres César García Caballero y Soledad Ortega Lara por su apoyo, confianza y por estar siempre pendientes de mis logros, gracias por siempre.

A las autoridades del Instituto Tecnológico de Tehuacán, por todas las facilidades otorgadas para la realización de este proyecto.

A todas las personas amigos y compañeros que de alguna manera contribuyeron para la realización de este proyecto, en particular a mis amigos Lili, Paco, Ramón, Adrián y al Ing. Serafín por su apoyo constante, su preocupación y su paciencia.

A mis hermanas Paty, Gaby, Vero y Moni por su apoyo, confianza y por estar siempre pendientes de mi. Gracias.

A mis hijos, que son mi razón de ser: a Felipe y a Nestor, que sin saberlo han sido parte muy importante para poder realizar y concluir este proyecto y a mi hija Iris con todo mi amor y mi admiración, gracias por todo tu apoyo. Los amo.

Con gran agradecimiento a la Dra. Lourdes Sandoval Solis y al M.C. Graciano Cruz Almanza por todo el apoyo y facilidades brindadas que fueron muy importantes para realizar y culminar este proyecto.

A COSNET por el apoyo económico brindado para la realización de este proyecto.

A mi esposo Felipe con todo mi amor y mi admiración, por todo tu apoyo, confianza y tu paciencia, por ser parte de este proyecto. Gracias te amo.

INDICE

CAPITULO IV IMPLEMENTACION DE UNA UNIDAD DE MEMORIA VIRTUAL COMPARTIDA CON MONITOREO (MEMVIR)

INTRODUCCION

El paralelismo es una técnica que resuelve las necesidades que actualmente se tienen en sistemas que requieren alto poder de procesamiento. Esto se hace al descomponer una tarea, en varias subtareas las cuales puedan ser asignadas a diferentes procesadores.

Las ventajas que ofrece el paralelismo son varias entre las que se pueden mencionar: alta capacidad de cómputo al repartir las tareas y lograr un menor tiempo de ejecución y alta capacidad de memoria.

Debido a que el procesamiento paralelo cobra cada día mayor importancia en distintas áreas y en consecuencia para muchos problemas es indispensable el pensar en una arquitectura paralela para darles solución; en la Facultad de Ciencias de la Computación de la Benemérita Universidad Autónoma de Puebla desde hace ya algunos años se ha estado trabajando en algunos proyectos en el área de paralelismo, con lo cuales cada día se va consolidando y sentando las bases para la generación de nuevos proyectos de investigación así como trabajos y propuestas de tesis tanto a nivel de hardware como software que pueden ser dirigidos a alumnos egresados de la licenciatura o de posgrado.

Para realizar el procesamiento paralelo se necesita algún tipo de arquitectura paralela en la cual una característica importante es como se accesa a la memoria, por lo tanto se tiene la necesidad de contar con un software que imite el comportamiento de una unidad de memoria, pensando en una arquitectura tipo Von Neumann, el cual una vez instalado en una computadora personal, sea visto por los usuarios como una unidad de almacenamiento temporal en la que puedan almacenar datos e incluso programas en ella, y pueda ser utilizada por sistemas que requieran este tipo de memoria.

El sistema MEMVIR ofrece a los usuarios una unidad de memoria virtual compartida basada en el modelo espacio de tuplas, que pueda ser utilizada de manera sencilla para poder ejecutar aplicaciones que demanden solicitudes de memoria en forma de tuplas de una manera más sencilla y didáctica. Ofrece también un sistema de monitoreo superior al proporcionado por el modelo Linda el cual sea de mucha ayuda al usuario para poder depurar sus aplicaciones de manera más rápida y sencilla.

Se logró la creación de una memoria virtual compartida con las siguientes características:

- Se basa en el modelo espacio de tuplas a partir del cual se realiza la comunicación y sincronización de los procesos que accesan a dicha memoria.
- El espacio de memoria está ubicado en una sola máquina, es decir se tiene un modelo de memoria centralizada.

- La recuperación de datos del espacio compartido se realiza por asociación y no por acceso a direcciones físicas.
- Integra un sistema de monitoreo, el cual se diseñó pensando en el usuario para poder proporcionarle la mayor información posible al momento de diseñar sus aplicaciones que hagan uso de esta memoria.

Además se lograron diversas aportaciones con respecto al modelo Linda, como son: contar con un servidor dinámico que todo el tiempo este disponible para atender solicitudes de memoria sin tener que reiniciarlo cada vez que un usuario desea utilizarlo para correr sus aplicaciones, además de aceptar solicitudes de memoria el servidor atiende solicitudes de conexión lo cual hace escalable el sistema, por el otro lado también se puede en cualquier momento dar de baja a cualquier computadora que se encuentre en la red sin que eso ocasione que el servidor de memoria se tenga que reiniciar, se tiene un sistema tolerante a fallas sencillo ya que en varios casos el servidor puede corregir situaciones que le pueden causar problemas, cada vez que una nueva conexión es atendida en ese momento se considera para la asignación de trabajo, El servidor de memoria maneja una serie de tuplas de control que son muy útiles para su buen funcionamiento y mejor entendimiento en el sistema de monitoreo, el servidor puede ser suspendido pero antes de salir envía una tupla de control a todas las computadoras que se encuentren conectadas para que terminen de manera normal sin dejar basura o descriptores abiertos.

El sistema MEMVIR se desarrollo en el lenguaje ‘C’, bajo la plataforma Linux.

La presentación del trabajo tiene la siguiente distribución:

En el capítulo I se presenta la Taxonomía de Flynn, la cual es importante conocer ya que ayuda a entender las diferentes arquitecturas modernas. Posteriormente se describe el modelo Linda ya que fue la guía para el diseño e implementación de este trabajo.

En el capítulo II se describe la memoria y como esta estructurada la jerarquía de memoria, lo cual es muy importante conocer para poder entender el esquema espacio de tuplas y que bondades ofrece con respecto a otros esquemas, así este capitulo se termina presentando este esquema de memoria espacio de tuplas y sus características para tomarlas en consideración durante el diseño del trabajo.

En el capítulo III se presenta el diseño del sistema MEMVIR y se da una breve explicación de los diagramas principales que se utilizaron para la implementación de MEMVIR.

Finalmente en el capítulo IV se presenta los detalles de la implementación de MEMVIR, mostrando los fragmentos de código más relevantes y su explicación para poder entender su funcionamiento.

CAPITULO I

PROCESAMIENTO PARALELO

1.1 INTRODUCCION

El procesamiento paralelo es una tecnología que se está desarrollando muy rápidamente; consiste en dividir una tarea computacional en varias tareas pequeñas, se lleva a cabo con una serie de procesadores encargados cada uno de determinadas labores, cuyos resultados se conjuntan para obtener un resultado final, con el fin de aumentar la velocidad computacional de un sistema de cómputo. Así en lugar de procesar cada instrucción en forma secuencial como se haría en una computadora secuencial, en un sistema de procesamiento paralelo se puede ejecutar el procesamiento en diversos procesadores para conseguir un menor tiempo de ejecución.

Por ejemplo, el sistema puede tener 2 o más ALU's y ser capaz de ejecutar dos o más operaciones al mismo tiempo. Además, el sistema puede tener dos o más procesadores operando en forma concurrente. El propósito del procesamiento paralelo es acelerar las posibilidades de procesamiento de la computadora y aumentar su eficiencia, es decir, la capacidad de procesamiento que puede lograrse durante un cierto intervalo de tiempo. La cantidad de circuitos aumenta con el procesamiento paralelo y, con él, también el costo del sistema. Sin embargo, los descubrimientos tecnológicos han reducido el costo de los circuitos a un punto en donde las técnicas de procesamiento paralelo son económicamente factibles.

Una arquitectura paralela es una colección de elementos de procesamiento que se comunican y cooperan para resolver rápido problemas grandes [1].

Existen diversas arquitecturas de computadoras paralelas. Estas se diferencian en como se organizan sus procesadores, memoria e interconexiones. Los sistemas más comunes son:

- MPP: Massive Parallel Processors
- SMP: Symmetric Multiprocessors
- CC-NUMA: Cache-Coherent Nonuniform Memory Access
- Sistemas distribuidos
- Clusters

El principal problema al que se enfrenta el modelo para las arquitecturas paralelas es cómo organizar múltiples procesadores para ejecutar al mismo tiempo. El estudio en el área del paralelismo deberá ser más fácil si es posible identificar un modelo de máquina paralela que sea general y útil como el modelo secuencial de Von Neumann. Este modelo de máquina deberá ser tanto simple como real: simple para facilitar la comprensión y la programación, y real para asegurar que los programas desarrollados en el modelo ejecuten con una eficiencia razonable sobre computadoras reales [2].

1.2 TAXONOMÍA DE FLYNN

Existen varias maneras de clasificar el procesamiento paralelo. Puede considerarse a partir de la organización interna de los procesadores, desde la estructura de interconexión entre los procesadores o desde del flujo de la información a través del sistema. Una clasificación presentada por M. J. Flynn considera la organización de un sistema de computadoras, mediante la cantidad de instrucciones y unidades de datos que se manipulan en forma simultánea. La operación normal de una computadora es recuperar instrucciones de la memoria y ejecutarlas en el procesador. La secuencia de instrucciones leídas de la memoria constituye un flujo de instrucciones. Las operaciones ejecutadas sobre los datos en el procesador constituyen un flujo de datos [3].

El procesamiento paralelo puede ocurrir en el flujo de instrucciones, en el flujo de datos o en ambos. La clasificación de Flynn divide a las computadoras en cuatro grupo principales de la siguiente manera:

1.2.1 **SISD** (Single Instruction Single Data)

Este es el modelo tradicional de procesamiento secuencial, representa la organización de una computadora que contiene una unidad de control, una unidad de procesador y una unidad de memoria (las máquinas Von Neumann pertenecen a la clase SISD). Tienen un flujo de instrucciones, uno de datos y realizan una operación a la vez (Fig. 1.1). Las instrucciones se ejecutan en forma secuencial [3].

Fig. 1.1 Arquitectura SISD.

1.2.2 **SIMD** (Single Instruction Multiple Data)

El modelo SIMD consiste de *n* procesadores idénticos, una memoria, una red de interconexión y una unidad de control. Todos los procesadores operan bajo el control de una sola secuencia de instrucciones emitida por una unidad de control y cada procesador deberá ejecutar la misma instrucción al mismo tiempo, pero sobre conjuntos distintos de datos [4]. Ejemplo de este tipo son las máquinas vectoriales de Cray.

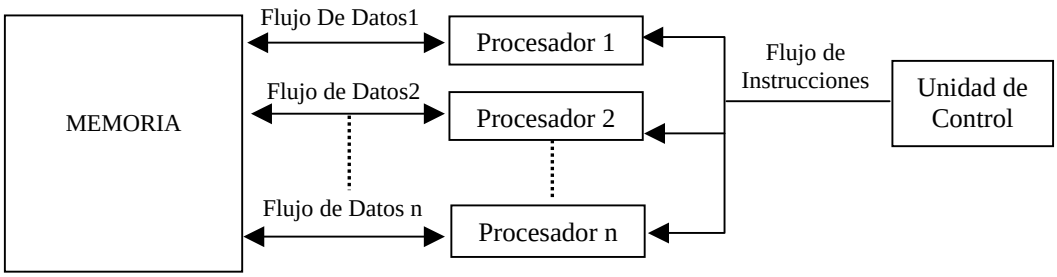


Fig. 1.2 Arquitectura SIMD.

1.2.3 **MIMD** (Multiple Instruction Multiple Data)

Este tipo de computadora es paralela al igual que las SIMD, la diferencia con estos sistemas es que MIMD es asíncrono o en términos simples, pueden estar haciendo diferentes cosas en diferentes datos al mismo tiempo. No tiene un reloj central. Cada procesador en un sistema MIMD puede ejecutar su propia secuencia de instrucciones y tener sus propios datos. Este tipo de arquitectura es la más general y poderosa de esta clasificación [5].

Los sistemas MIMD se clasifican en:

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

Sistemas MIMD de Memoria Compartida

En este tipo de sistemas cada procesador tiene acceso a toda la memoria, es decir hay un espacio de direccionamiento compartido. Se tienen tiempos de acceso a memoria uniformes ya que todos los procesadores se encuentran igualmente comunicados con la memoria principal y las lecturas y escrituras de todos los procesadores tienen exactamente las mismas latencias; y además el acceso a memoria es por medio de un ducto común (fig. 1.3). En esta configuración, debe asegurarse que los procesadores no tengan acceso simultáneamente a regiones de memoria de una manera en la que pueda ocurrir algún error [5].

Desventajas:

- ✓ El acceso simultáneo a memoria.
- ✓ Poca escalabilidad de procesadores, debido a que se puede generar un cuello de botella al incrementar el número de CPU's.
- ✓ Hay interferencia entre CPUs.
- ✓ La razón principal por el alto precio es la memoria.

Ventajas:

- ✓ La facilidad de la programación. Es mucho más fácil programar en estos sistemas que en sistemas de memoria distribuida.
- ✓ Bajo costo ya que ya que sólo hay un módulo de memoria.
- ✓ Fácil crecimiento, ya que es relativamente fácil incrementar procesadores.

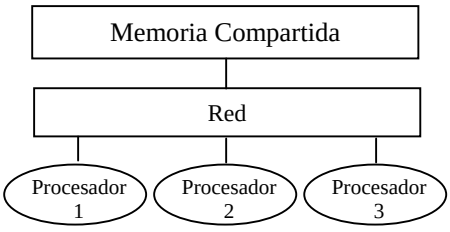


Fig. 1.3 Arquitectura MIMD de Memoria Compartida.

Las computadoras MIMD con memoria compartida más elementales son sistemas conocidos como de multiprocesamiento simétrico (SMP) donde múltiples procesadores comparten un mismo sistema operativo y memoria.

Sistemas MIMD de Memoria Distribuida

En estos sistemas cada procesador tiene su propia memoria local. Los procesadores pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos (fig. 1.4). Esta comunicación se le conoce como Paso de Mensajes [5].

Ventajas:
- ✓ La escalabilidad. Las computadoras con sistemas de memoria distribuida son fáciles de escalar, mientras que la demanda de los recursos crece, se puede agregar más memoria y procesadores.

Desventajas:
- ✓ El acceso remoto a memoria es lento.
- ✓ La programación puede ser complicada.

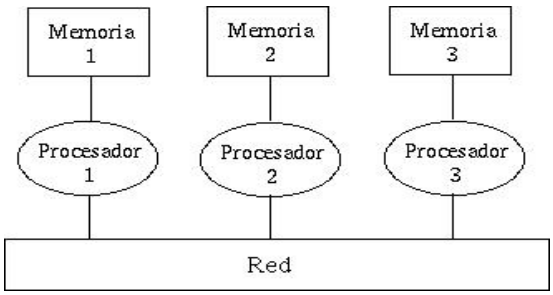


Fig. 1.4 Arquitectura MIMD de Memoria Distribuida.

Las computadoras MIMD de memoria distribuida son conocidas como sistemas de procesamiento en paralelo masivo (MPP) donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria. Además se les llama multicomputadoras, máquinas libremente juntas o cluster. Algunos ejemplos de este tipo de máquinas son IBM SP2 y SGI/Cray T3D/T3E [5].

Sistemas MIMD de Memoria Compartida Distribuida

Es un cluster o un grupo de procesadores que tienen acceso a una memoria compartida común pero sin un canal compartido. Esto es, físicamente cada procesador posee su memoria local y se interconecta con otros procesadores por medio de un dispositivo de alta velocidad, y todos ven las memorias de cada uno como un espacio de direcciones globales [5] (fig. 1.5).

Ventajas:

- ✓ Presenta escalabilidad como en los sistemas de memoria distribuida.
- ✓ Es fácil de programar como en los sistemas de memoria compartida.
- ✓ No existe el cuello de botella que se puede dar en máquinas de sólo memoria compartida.

Desventaja:

- ✓ El acceso remoto es lento.

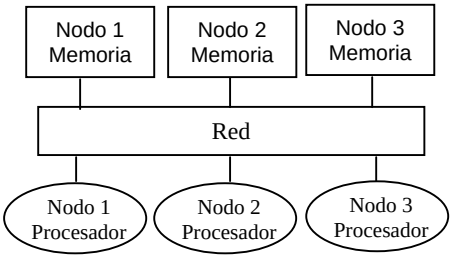


Fig. 1.5 Arquitectura MIMD de Memoria Compartida Distribuida.

Algunos ejemplos de este tipo de sistemas son HP/Convex SPP-2000 y SGI/Cray Origin2000.

1.2.4 MISD (Multiple Instrucion Single Data)

En este modelo, secuencias de instrucciones pasan a través de múltiples procesadores. Diferentes operaciones son realizadas en diversos procesadores. *N* procesadores, cada uno con su propia unidad de control comparten una memoria común (fig. 1.6).

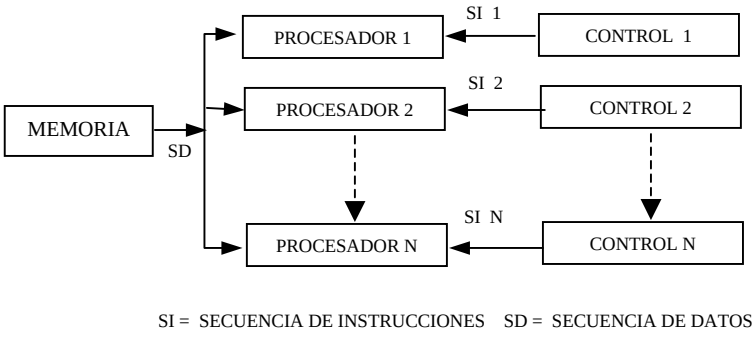


Fig. 1.6 Arquitectura MISD.

Hay *N* secuencias de instrucciones (algoritmos/programas) y una secuencia de datos. El paralelismo es alcanzado dejando que los procesadores realicen diferentes cosas al mismo tiempo en el mismo dato. Las máquinas MISD son útiles en cómputos donde la misma entrada esta sujeta a diferentes operaciones [5].

1.4 EL MODELO LINDA

Linda es una clase de lenguaje de programación que opera en arquitecturas paralelas, es decir es una herramienta que nos permite poder escribir programas paralelos. Desarrollado en la Universidad de Yale, su primera descripción data de 1982 a cargo de David Gelernter y su primera versión fue realizada por Nicholas Carriero en 1984 sobre una máquina S/Net de los Laboratorios Bell [6].

El lenguaje Linda está basado en el paradigma de la coordinación, el cual se fundamenta en la separación de los aspectos de computación y de interacción de los componentes que integran un sistema [7]. Linda es un lenguaje de coordinación que proporciona comunicación generativa [8].

La comunicación generativa toma su nombre del hecho de que los procesos no se envían mensajes para comunicarse, en su lugar generan estructuras de datos pasivas (datos) o activas (código) en un espacio compartido, en el cual otros procesos pueden leer, escribir o borrar.

El lenguaje de coordinación Linda está basado en el uso de estructuras de datos para realizar la comunicación y sincronización, en donde una estructura contiene grupos de datos que pueden manipularse simultáneamente por varios procesos. Para la comunicación entre procesos se necesita acceder al contenido de la estructura para leer, depositar, o extraer componentes. El paralelismo se introduce permitiendo la realización de tales acciones a múltiples procesos sobre diferentes componentes de la estructura de datos.

Este lenguaje permite al desarrollador crear programas paralelos que se ejecuten sobre una variedad de arquitecturas de computadoras, debido a que está basado en lenguajes que son muy conocidos para la mayoría de los programadores; es fácil de usar ya que permite utilizar la potencialidad del paralelismo con un pequeño número de simples operaciones sobre algo denominado espacio de tuplas, para crear y coordinar los procesos paralelos [9].

El lenguaje de coordinación Linda opera bajo el paradigma de programación "maestro/trabajador", en donde un proceso maestro recibe tareas (código) o datos. Cada proceso trabajador obtiene una de estas tareas, la ejecuta y envía el resultado de regreso al proceso maestro, obteniendo otra tarea y así sucesivamente. Cuando no hay más tareas por realizar, el trabajo está terminado y se envían los resultados al desarrollador.

En la figura 1.7 se muestra de forma grafica el modelo Linda.

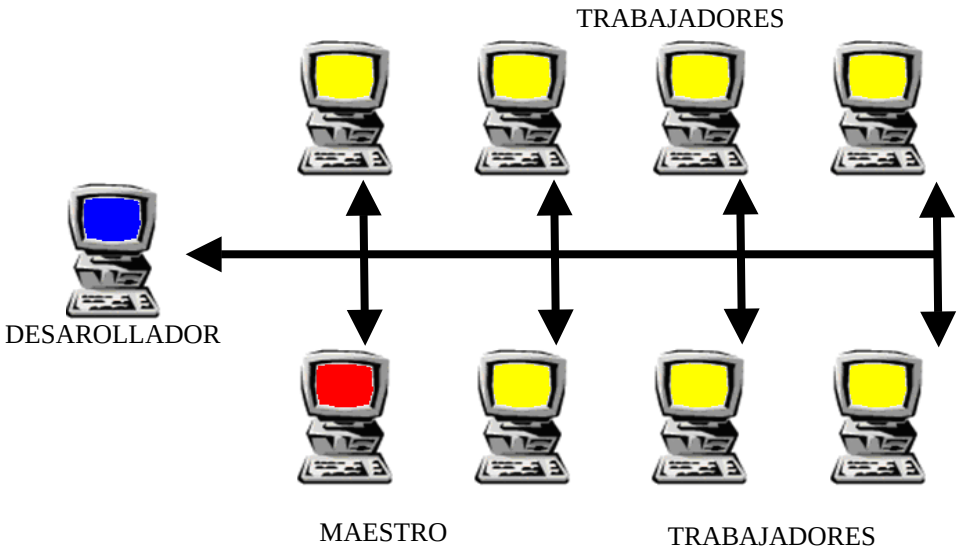


Fig. 1.7 Modelo Linda.

Aunque el modelo Linda puede parecerse al modelo de paso de mensajes comúnmente usado en multiprocesadores, éste es más potente. El paso de mensajes requiere que el proceso que envía información conozca la identidad del receptor; los datos no existen independientemente de los procesos en ejecución. En contraste, los datos producidos por un programa Linda, son completamente autónomos. Los procesos pueden crear datos sin conocer qué proceso los usará, ni cuándo podría ocurrir este uso. Esta persistencia de los datos, independientemente de los procesos que los crean, incrementa el desacoplo entre ellos, haciendo los programas en Linda mucho más fáciles de escribir y entender que sus contrapartes en paso de mensajes [10].

CAPITULO II

ESQUEMAS DE MEMORIA

2.1 INTRODUCCION

Aunque actualmente la mayoría de los sistemas de cómputo cuentan con una alta capacidad de memoria, de igual manera las aplicaciones actuales tienen también altos requerimientos de memoria, lo que sigue generando escasez de memoria en los sistemas multitarea y/o multiusuario.

Hoy en día cada vez se requiere más memoria para poder utilizar complejos programas y para gestionar complejas redes de computadoras.

La memoria es uno de los principales recursos de la computadora, se puede definir como los circuitos que permiten almacenar y recuperar la información. En un sentido más amplio, puede referirse también a sistemas externos de almacenamiento.

2.2 Elementos de la memoria

Una memoria vista desde el exterior, tiene la estructura mostrada en la figura 2-1. Para efectuar una lectura se deposita en el bus de direcciones la dirección de la palabra de memoria que se desea leer y entonces se activa la señal de lectura (R); después de cierto tiempo (tiempo de latencia de la memoria), en el bus de datos aparecerá el contenido de la dirección buscada. Por otra parte, para realizar una escritura se deposita en el bus de datos la información que se desea escribir y en el bus de direcciones la dirección donde deseamos escribirla, entonces se activa la señal de escritura (W), pasado el tiempo de latencia, la memoria escribirá la información en la dirección deseada. Internamente la memoria tiene un registro de dirección (MAR, memory address register), un registro buffer de memoria o registro de datos (MB, memory buffer, o MDR, memory data register) y, un decodificador. Esta forma de estructurar la memoria se llama organización lineal o de una dimensión [11].

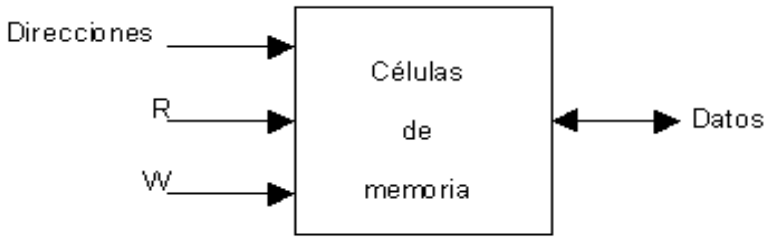


Fig. 2.1 Estructura externa de la memoria.

2.3 JERARQUIA DE MEMORIA

En una computadora hay una jerarquía de memorias atendiendo al tiempo de acceso y a la capacidad que normalmente son factores contrapuestos por razones económicas y en muchos casos también físicas.

Debe existir un mecanismo que permita al procesador saber cuándo una dirección Se encuentra en el nivel superior de la jerarquía.

Comenzando desde el procesador y hacia el exterior, es decir en orden creciente de tiempo de acceso y capacidad, se puede establecer la siguiente jerarquía [12].

2.3.1 REGISTROS DE PROCESADOR

Estos registros interaccionan continuamente con la CPU (porque forman parte de ella). Los registros tienen un tiempo de acceso muy pequeño que depende directamente de la tecnología usada en la fabricación del CPU. Estos registros tienen una capacidad mínima, normalmente igual a la palabra del procesador, es decir entre 1 y 8 bytes.

2.3.2 REGISTROS INTERMEDIOS

Constituyen el paso intermedio entre el procesador y la memoria, tienen un tiempo de acceso muy breve ya que están dentro del CPU. Dichos registros tiene poca capacidad y normalmente corresponde al doble de los registros del procesador, es decir entre 2 y 16 bytes [12].

2.3.3 MEMORIAS CACHE

Son memorias de capacidad pequeña. Normalmente una pequeña fracción de la memoria principal y un reducido tiempo de acceso. Este nivel de memoria se coloca entre la CPU y la memoria central. Hace algunos años este nivel era exclusivo de las computadoras grandes pero actualmente todas las computadoras lo incorporan. Dentro de la memoria caché puede haber, a su vez, dos niveles denominados caché on chip, memoria caché dentro del circuito integrado, y caché on board, memoria caché en la placa de circuito impreso pero fuera del circuito integrado. Evidentemente, por razones físicas, la primera es mucho más rápida que la segunda, existe también una técnica, denominada Arquitectura Harvard, que utiliza memorias caché separadas para código y datos [12]. La capacidad de las memorias caché se mide en términos de miles de bytes.

2.3.4 MEMORIA PRINCIPAL

La memoria principal corresponde a uno de los niveles más importantes ya que en ella residen tanto los programas como los datos. Las características de la memoria principal son un elevado número de localidades de almacenamiento, que normalmente se miden en megabytes y puede llegar incluso hasta gigabytes. Con respecto a la velocidad es reducida en comparación con los registros ya que en este nivel se lee y se escribe con menos frecuencia que en los niveles anteriores [12].

2.3.5 MEMORIA VIRTUAL

La necesidad cada vez más imperiosa de ejecutar programas grandes y el crecimiento en poder de las unidades centrales de procesamiento empujaron a los diseñadores de los sistemas operativos a implantar un mecanismo para ejecutar automáticamente programas más grandes que la memoria real disponible, esto es, de ofrecer “memoria virtual”..

La memoria virtual se llama así porque el programador ve una cantidad de memoria mucho mayor que la real, y en realidad se trata de la suma de la memoria de almacenamiento primario y una cantidad determinada de almacenamiento secundario [13].

La memoria virtual es una técnica para proporcionar la simulación de un espacio de memoria mucho mayor que la memoria física de una máquina. Esta "ilusión" permite que los programas se hagan sin tener en cuenta el tamaño exacto de la memoria física. La ilusión de la memoria virtual está soportada por el mecanismo de traducción de memoria, junto con una gran cantidad de almacenamiento rápido en disco duro. Así en cualquier momento el espacio de direcciones virtual hace un seguimiento de tal forma que una pequeña parte de él, está en memoria real y el resto almacenado en el disco, y puede ser referenciado fácilmente [14].

Características:

- Divide el espacio de memoria disponible en bloques, y los asigna a diferentes procesos.
- Permite ejecutar programas de mayor tamaño que la memoria física disponible.
- Gestiona automáticamente los niveles de memoria principal (DRAM) y secundaria (discos).

La traducción de direcciones se hace mediante una combinación de hardware y software, las direcciones virtuales se traducen a direcciones físicas que permiten acceder a memoria principal.

2.4 MEMORIA VIRTUAL COMPARTIDA

Aunque la memoria virtual permite que cada proceso tenga un espacio de memoria separado, hay veces que es necesario que varios procesos compartan memoria. Por ejemplo pueden haber varios procesos del sistema ejecutando el intérprete de comados bash. En lugar de tener varias copias del bash, una en cada memoria virtual de cada proceso, es mejor sólo tener una sola copia en memoria física y que todos los procesos que ejecuten bash la compartan. Las bibliotecas dinámicas son otro ejemplo típico de código ejecutable compartido por varios procesos.

Por otra parte, la memoria compartida se puede utilizar como un mecanismo de comunicación entre procesos (Inter Process Communication IPC), donde dos o más procesos intercambian información vía memoria común a todos ellos.

Conforme el procesador va ejecutando un programa lee instrucciones de la memoria y las decodifica. Durante la decodificación de la instrucción puede necesitar cargar o guardar el contenido de una posición de memoria. El procesador ejecuta la instrucción y pasa a la siguiente instrucción del programa. De esta forma el procesador está siempre accediendo a memoria tanto para leer instrucciones como para cargar o guardar datos.

Gracias a los mecanismos de memoria virtual se puede conseguir fácilmente que varios procesos compartan memoria. Todos los accesos a memoria se realizan a través de las tablas de páginas y cada proceso tiene su propia tabla de páginas. Para que dos procesos compartan una misma página de memoria física, el número de marco de está página ha de aparecer en las dos tablas de página [15].

2.5. ESPACIO DE TUPLAS

El Espacio de Tuplas es un modelo de memoria virtual compartida, que provee una comunicación y sincronización entre procesos, que es independiente de la plataforma. El espacio de tuplas es el lugar en donde todos los procesos pueden insertar y extraer tuplas (siempre de forma asociativa).

Accesible Globalmente:

El espacio de tuplas puede ser implementado de maneras diferentes, tales como dividir la memoria entre las memorias convencionales de cada procesador dentro

de un sistema distribuido, duplicar la memoria en cada procesador o proporcionar un módulo de memoria física compartida.

Asociativa

Se le denomina memoria asociativa, debido a que las tuplas no son direccionadas utilizando una localidad de memoria, sino por el contenido mismo de la tupla, esto ayuda a la implementación del espacio de tuplas, ya que la localización de la tupla no necesita ser conocida y puede por lo tanto estar en procesadores alejados.

Al espacio de Tuplas se le asocia con el modelo de memoria virtual compartida, debido a que los procesadores accesan un área común de memoria, y se diferencia de ella en que la información se recupera por contenido (asociativa) y no por su dirección en memoria.

Este modelo introduce sólo un nuevo tipo de objeto, llamado *tupla*. Una tupla es un conjunto ordenado de datos con tipo que puede ser utilizado por cualquier proceso. Estas tuplas pueden agregarse y/o removerse de la memoria compartida.

El espacio de tuplas es administrado utilizando un número pequeño de operaciones, éstas aseguran que las tuplas no sean removidas por más de un proceso al mismo tiempo. El espacio de tuplas implícitamente resuelve exclusión mutua y puntos de sincronización.

La sincronización entre procesos se realiza bloqueando a aquellos procesos que requieren de una tupla hasta que ella este disponible.

Hay dos tipos básicos de tuplas: las tuplas de proceso llamadas también tuplas activas, y las tuplas de datos llamadas tuplas pasivas. Las tuplas activas intercambian datos mediante la generación, lectura y consumo de las pasivas.

Un esquema general del espacio de tuplas se muestra en la figura. 2.2.

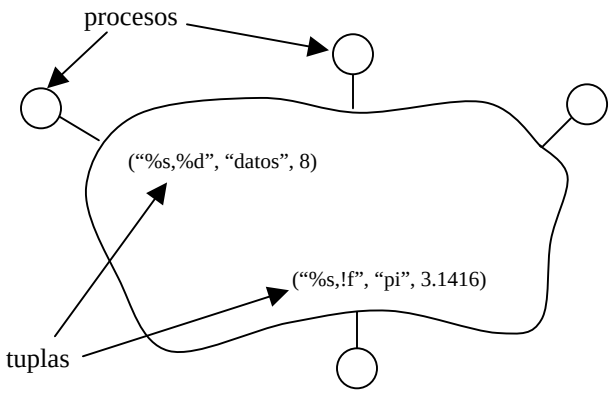


Fig. 2.2 Esquema del espacio de tuplas.

En algunas versiones del modelo Linda, el espacio de tuplas está localizado en una máquina denominada maestro. El maestro es el responsable de administrar las tuplas para poder dar atención a las solicitudes que reciba de los procesos trabajadores. Cuando las tuplas son colocadas en el espacio, el maestro no verifica por la repetición de ellas, cualquier número de tuplas duplicadas puede existir en el espacio.

Un espacio de tuplas tiene cuatro propiedades importantes:

1. **Es compartido**. Esta propiedad indica que todos los procesos trabajan con el mismo espacio de tuplas y el espacio puede ser utilizado por cualquier proceso.

2. **Es como una bolsa no como un conjunto**. Lo cual quiere decir que en el espacio puede haber muchas tuplas idénticas.

3. **Es asociativo**. Las tuplas, son removidas del espacio de tuplas utilizando reglas de comparación con respecto al contenido, más que el ser direccionada.

4. **Es anónimo**. Una vez que una tupla ha sido colocada en el espacio de tuplas, el sistema no mantiene la pista de quien la creó o cuando.

COMPARACION DE TUPLAS

Para saber si una tupla se encuentra en el espacio de tuplas, se utiliza una relación asociativa: las tuplas no tienen una dirección así que para localizar la que se esta solicitando, se tiene que buscar por cualquier combinación de valores de los campos correspondientes.

Cada tupla puede tener de cero a siete elementos, en donde el primero corresponde a el nombre de la tupla, y lo seis restantes deberán ser un valor o variable que puede ser formal o actual. Si se requiere un elemento para almacenar un valor, entonces el elemento es formal, por otro lado si lo que importa es el valor de él, entonces es un elemento actual.

Una tupla solicitada es igual a la que se encuentra en el espacio de tuplas si se cumple lo siguiente:

1. Los nombres de ambas son iguales.
2. Las dos tiene el mismo número de campos.
3. Los tipos de los campos correspondientes son los mismos.
4. Los valores de los campos actuales correspondientes son los mismos.
5. Los elementos actuales son iguales a los elementos formales si ellos son del mismo tipo y longitud.

CAPITULO III

DISEÑO DE UNA UNIDAD DE MEMORIA VIRTUAL COMPARTIDA MEMVIR

3.1 INTRODUCCION

Debido a que el procesamiento paralelo cobra cada día mayor importancia en distintas áreas y en consecuencia para muchos problemas es indispensable el pensar en una arquitectura paralela para darles solución; en la Facultad de Ciencias de la Computación de la Benemérita Universidad Autónoma de Puebla desde hace ya algunos años se ha estado trabajando en algunos proyectos en el área de paralelismo, con lo cuales cada día se va consolidando y sentando las bases para la generación de nuevos proyectos de investigación así como trabajos y propuestas de tesis tanto a nivel de hardware como software que pueden ser dirigidos a alumnos egresados de la licenciatura o de posgrado.

3.2 DESCRIPCION GENERAL

Una importante característica de las arquitecturas paralelas es el sentido en el cual la memoria es accesada, de ahí que se tenga la necesidad de contar con software que imite el comportamiento de una unidad de memoria, esto es pensando en una arquitectura tipo Von Neumann, la cual una vez instalada en una computadora personal, sea vista por los usuarios como una unidad de almacenamiento temporal en la que puedan almacenar datos e incluso programas en ella, y pueda ser utilizada por sistemas que requieran este tipo de memoria.

El objetivo de este proyecto es la implementación de una memoria sobre una PC la cual estará conectada en red y permitirá que los usuarios a través de la red se conecten y puedan hacer uso de la memoria. Formalmente puede decirse que será un servidor que atienda solicitudes de memoria virtual compartida a los usuarios de la red.

Es importante mencionar que para el desarrollo de este proyecto se tomó como referencia el modelo de memoria “Espacio de Tuplas”, el cual es el modelo que se utiliza en el lenguaje de coordinación Linda.

Como en muchos sistemas de comunicación el monitoreo es importante ya que permite visualizar las transacciones que se están realizando y de esta forma ayudar a depurar los sistemas con problemas, de ahí que se plantee reforzar este proyecto con un sistema de monitoreo de mensajes.

3.3 ESQUEMA GENERAL DE MEMVIR

El diseño del sistema se basa en tener una computadora personal que estará conectada en red con otro conjunto de computadoras, a las cuales les prestará servicio de memoria (datos o código, lo que en el modelo de memoria espacio de tuplas se conoce como tuplas), la comunicación entre ellas se realizará por medio de sockets.

Existe una máquina operada por el usuario la cual podrá solicitar datos, o enviar datos a la memoria (lo que en el modelo LINDA se conoce como Desarrollador), y otro conjunto de computadoras las cuales podrán solicitar datos, o enviar datos a la memoria (en el modelo Linda se denominan máquinas trabajadoras o workers).

Es importante destacar las partes de que costa el modelo espacio de tuplas de Linda, ya que será la base para el diseño de MEMVIR.

En este modelo podemos identificar las siguientes partes:

El maestro (master**).** Se le denomina así a la computadora donde se instalará el administrador de memoria (unidad de memoria). Su función será mantener las tuplas que son usadas como medio de comunicación entre los trabajadores y el desarrollador. También será el encargado de atender las solicitudes de memoria que le lleguen y darles seguimiento.

El trabajador (worker). Se le denomina así, al conjunto de computadoras que se utilizan solamente para realizar cálculos (unidad de procesamiento). Cada una de ellas obtendrá el trabajo a ejecutar mediante tuplas activas y las solicitudes de datos se realizaran por medio de tuplas pasivas; en ambos casos es importante la trabajo del master como puente de comunicación.

El desarrollador. Es la computadora donde se realizarán las aplicaciones paralelas que se desean ejecutar. Aquí se debe repartir el trabajo mediante la creación de tuplas activas que serán enviadas al master, para posteriormente recuperar los resultados mediante tuplas pasivas.

En la Fig. 3.1 se muestra el esquema general de MEMVIR y los módulos con los que interactúa.

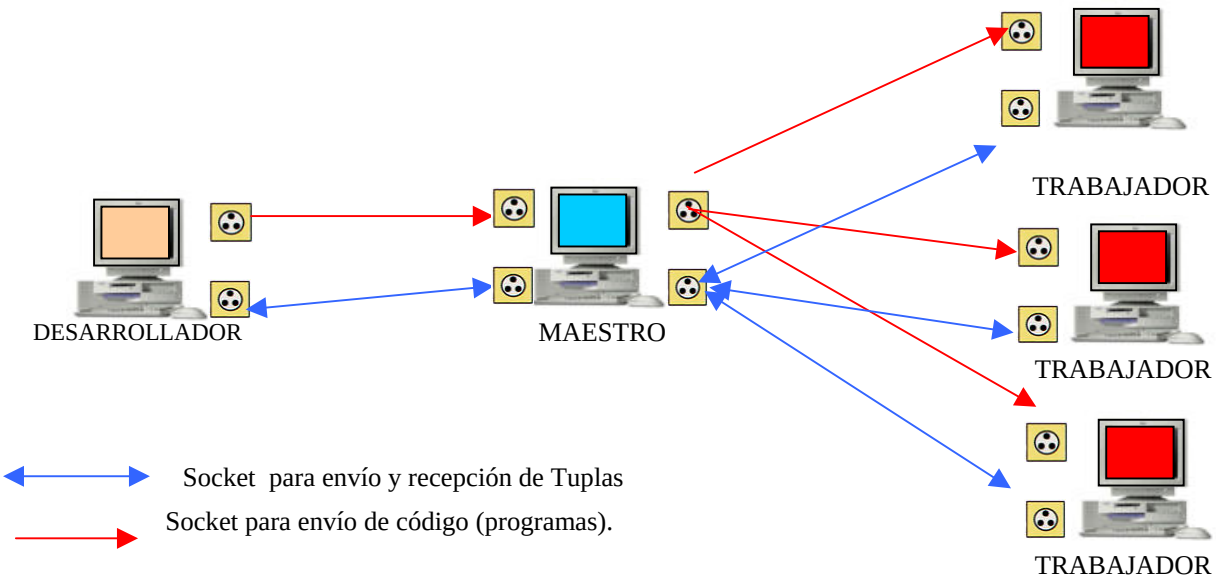


Fig. 3.1 Esquema General del Proyecto MEMVIR y los módulos con los cuales interactúa.

En las secciones siguientes se dará una descripción detallada de los módulos principales, con los que contará este servidor de memoria MEMVIR.

3.4 DIAGRAMA PRINCIPAL

La tarea principal del servidor de memoria MEMVIR (master) es estar observando un grupo de descriptores los cuales serán utilizados para envío y recepción de datos, para dar la atención a cada una de las solicitudes que le lleguen.

El comportamiento de este servidor es dinámico, lo cual quiere decir que siempre estará atendiendo solicitudes de conexión a la par de estar atendiendo solicitudes de datos (ya sea para enviar ó recibir), esto sin afectar su comportamiento. Esto representa una aportación a la forma en como opera el master en el modelo Linda, debido a que en Linda primero debían de realizarse las conexiones con las cuales se tenía que trabajar, sin tener la alternativa de agregar mas componentes al sistema.

En la fig. 3.2 se muestra el diagrama general de este servidor de memoria (master).

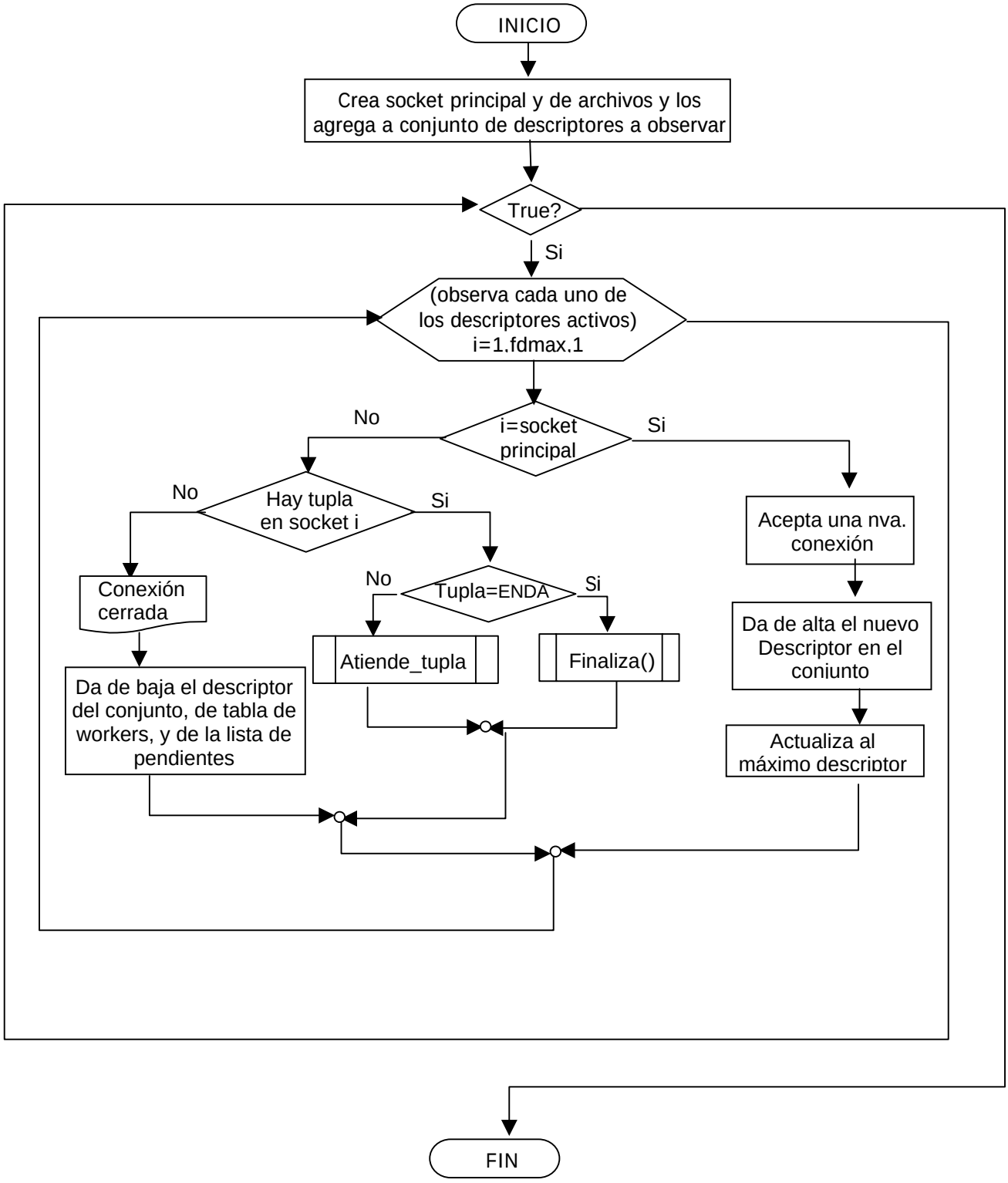


Fig. 3.2 Diagrama principal del servidor de memoria MEMVIR.

El sistema servidor de memoria inicia con la creación de los sockets que necesita para llevar a cabo la atención de las diferentes solicitudes, específicamente se crearán tres sockets principales antes de iniciar con la atención de solicitudes; los sockets que se crean son:

- Un socket a través del cual se dará atención a la solicitud de conexión que existan desde otras computadoras, y a partir del cual se crearán sockets para cada una de las conexiones aceptadas.
- Un segundo socket para atender la solicitud de recepción de código a la cual se le denomina tupla activa en el modelo Linda.
- Un tercer socket para el envío de código (tuplas activas) al worker que se encuentre disponible.

Una vez que estos sockets se han creado, al socket que estará atendiendo solicitudes de conexión se le insertará en un conjunto de descriptores y se le asignará como máximo descriptor, con el fin de estar censando continuamente este conjunto para detectar por cual esta llegando alguna solicitud y de esta manera estar dando atención a las diferentes solicitudes.

Este servidor iniciará un ciclo infinito, en el que se empezará por censar uno a uno cada uno de los descriptores que estén dados de alta en el conjunto a observar. Si el socket que esta listo es el asignado a atender conexiones, entonces se aceptará la nueva conexión asignándole un nuevo descriptor de socket para atender las solicitudes del worker que se acaba de conectar; dicho descriptor se comparará para determinar nuevamente el máximo descriptor y se le incluirá en el conjunto de sockets a observar. La finalidad de obtener el máximo descriptor es únicamente para controlar el ciclo que tiene la tarea de observar todos los sockets.

Si la solicitud que llegó no es de conexión entonces es de solicitud de tupla, y puede ser una tupla de control, una tupla activa o una tupla pasiva. Lo primero que realiza entonces es verificar que exista la tupla en el socket, ya que puede suceder que la tupla no llegue por que se cerró la conexión con el worker que hace la solicitud o puede ocurrir un error en la llamada a la función recv(), por lo tanto para evitar posibles errores se verifica por el valor que regresa la función recv() si este es igual a cero entonces es un indicador de que la conexión ha sido cerrada, si el valor es menor que cero entonces ocurrió algún error en la recepción de la tupla, en cualquiera de los casos entonces dará de baja el descriptor correspondiente del conjunto de descriptores y de las tablas correspondientes, para que ya no sea considerado.

Si el valor que regresa la función recv() es mayo a cero, entonces atenderá la solicitud de tupla deseada.

Si la tupla es de fin de aplicación (tupla de control ENDA), entonces limpiará el espacio de tuplas, reinicializará el sistema y regresará nuevamente al ciclo

principal. Si la tupla no fue ENDA entonces atenderá la solicitud, después de lo cual regresará nuevamente a censar a cada uno de los descriptores.

3.5 ALGORITMOS IMPORTANTES

3.5.1 Algoritmo atiende tuplas

Dentro de los algoritmos importantes tenemos a la función que dará atención a cada una de las tuplas que lleguen al master, esta función recibe como parámetro la tupla que llega al master y empieza por clasificar el tipo de tupla que es, para que sea atendida de manera correcta, aquí se hace la separación de tuplas activas (EVAL), tuplas pasivas (IN, OUT, RD, INP, RDP) y tuplas de control (ACK, ACK1) útiles para el modelo Linda. En el siguiente diagrama (fig. 3.3) se muestra cada caso.

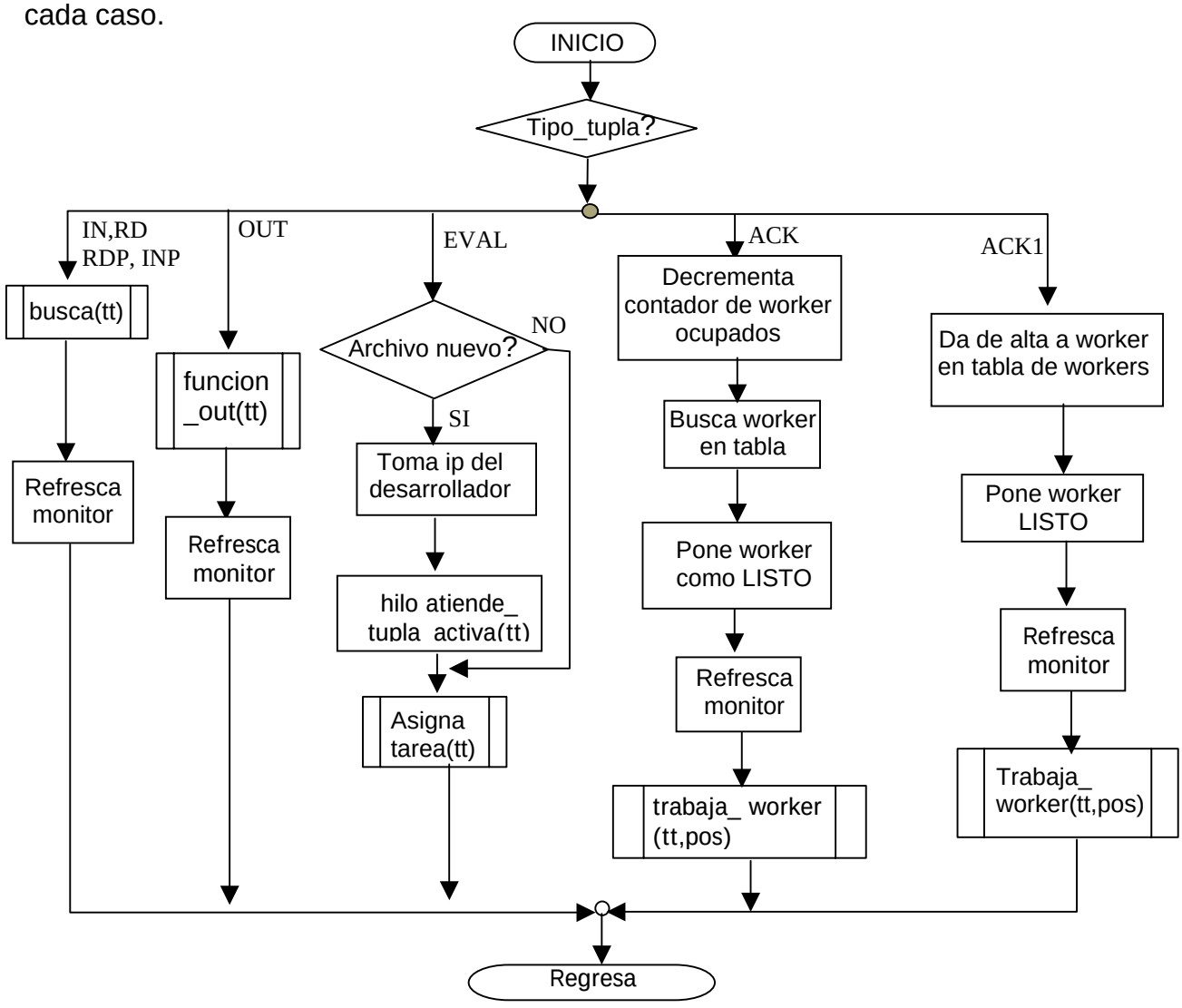


Fig. 3.3 Diagrama de la función atiende_tupla.

La función atiende tuplas recibe como parámetro la tupla que llego al servidor de memoria (master) y analiza el tipo de tupla que tiene para darle el tratamiento que corresponda.

Si el tipo de tupla corresponde a una tupla IN, RD, INP, o RDP (tuplas pasivas que solicitan alguna tupla) llama a la función busca (tt) enviándole como parámetro la tupla recibida, la tarea de esta función será precisamente buscar en el espacio de tuplas si se encuentra y realizar la acción correspondiente al tipo de tupla, al regreso del llamado hace un refresco del monitor para actualizar el contador de las tuplas pasivas.

Si el tipo de tupla corresponde a una tupla OUT (tupla pasiva que deposita tuplas en el espacio de tuplas), se llama a la función_out (tt) enviándole como parámetro la tupla recibida, la tarea de esta función será depositar la tupla en el espacio de tuplas, al regreso de esta función se hace un refresco del monitor para actualizar las tuplas activas.

Si el tipo de tupla es EVAL (tupla activa), esta tupla se utiliza para enviar código (programa) al espacio de tuplas, lo primero que realizará será verificar si el archivo es nuevo o si ya fue enviado con anterioridad, si es la primera vez que se solicita el envío de la tupla, entonces se toma la IP de la máquina (desarrollador) que hizo la solicitud y se refresca monitor, enseguida se lanza un hilo cuya tarea será precisamente recibir el archivo. Después de esto se llama a la función asigna_tarea (tt) a la que se le manda la tupla recibida y cuya tarea será asignar la tupla activa al primer worker que encuentre desocupado.

Si el tipo de tupla es ACK (tupla de control), esa tupla se envía cada vez que cada worker ha finalizado la ejecución del código correspondiente a la tupla activa que le fué asignada para que el master pueda actualizar su estado y si existe más tuplas activas pendientes le vuelva a asignar trabajo (tupla activa).

Lo primero que realiza es decrementar su contador de workers ocupados, actualizar la tabla de workers y ponerlo como LISTO, refresca monitor y manda a llamar a la función trabaja_worker (tt, pos), cuya tarea será asignarle una nueva tupla activa o agregarlo a la lista de workers pendientes.

Si el tipo de tupla es ACK1 (tupla de control), esa tupla se envía cada vez que un worker se conecta al master por primera vez, para solicitarle trabajo (tupla activa); lo primero que realiza es darlo de alta en la tabla de workers, marcarlo como LISTO, refrescar monitor y manda a llamar a la función trabaja_worker(tt, pos), para que sea puesto a trabajar si hay tuplas activas, o agregarlo a la lista de workers pendientes.

3.5.2 Algoritmo busca tupla

Este algoritmo tiene como tarea buscar la tupla solicitada en el Espacio de tuplas, si la tupla existe se le manda al worker que la solicito y dependiendo del tipo de solicitud será borrada o no del espacio de tuplas; si la tupla no existe entonces dependiendo del tipo de tupla se marcará como bloqueado al worker que la solicitó o se enviará la respuesta de tupla no encontrada.

En la fig. 3.4 se muestra el diagrama correspondiente a esta función.

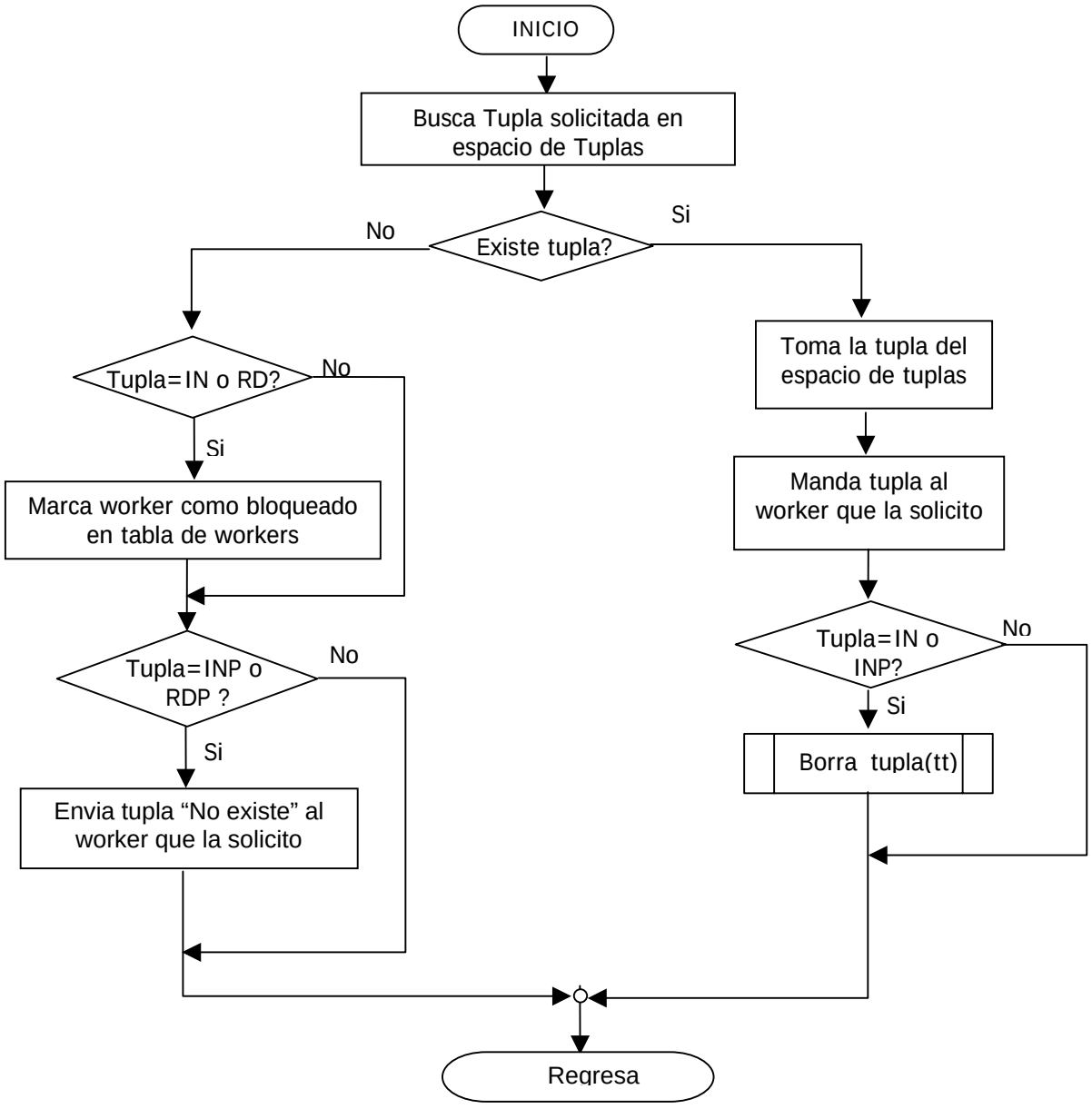


Fig. 3.4 Diagrama de la función busca (tt).

Esta función se invoca cada vez que al master llega una tupla pasiva del tipo IN. Recibe como parámetro la tupla que llegó al master y lo primero que realizará será buscar la tupla solicitada en el espacio de tuplas, para ver si existe ya y de acuerdo al resultado de la búsqueda y del tipo de tupla se dará respuesta a la solicitud.

Si la tupla se encuentra en el espacio de tuplas, entonces se manda al worker que la solicitó; y si la tupla fue del tipo IN o INP entonces además de enviar la tupla, ésta se elimina del espacio de tuplas.

Si la tupla solicitada no fue encontrada en el espacio de tuplas, se verifica si el tipo de tupla fue IN O RD, entonces el worker que hizo la solicitud se marca como bloqueado en la tabla de workers.

Si la tupla solicitada fue del tipo INP o RDP, y no fue encontrada entonces se envía la tupla “No existe” al worker que hizo la solicitud para indicarle que la tupla no fue encontrada en el espacio de tuplas, sin tener que marcarlo como bloqueado; este tipo de tuplas se emplean simplemente para consulta sin necesidad de bloquear al worker que haga la solicitud para que pueda continuar con su ejecución.

3.5.3 Algoritmo función out

Este algoritmo tiene como tarea depositar tuplas en el espacio de tuplas.
En la figura (3.5) se muestra el diagrama de esta función.

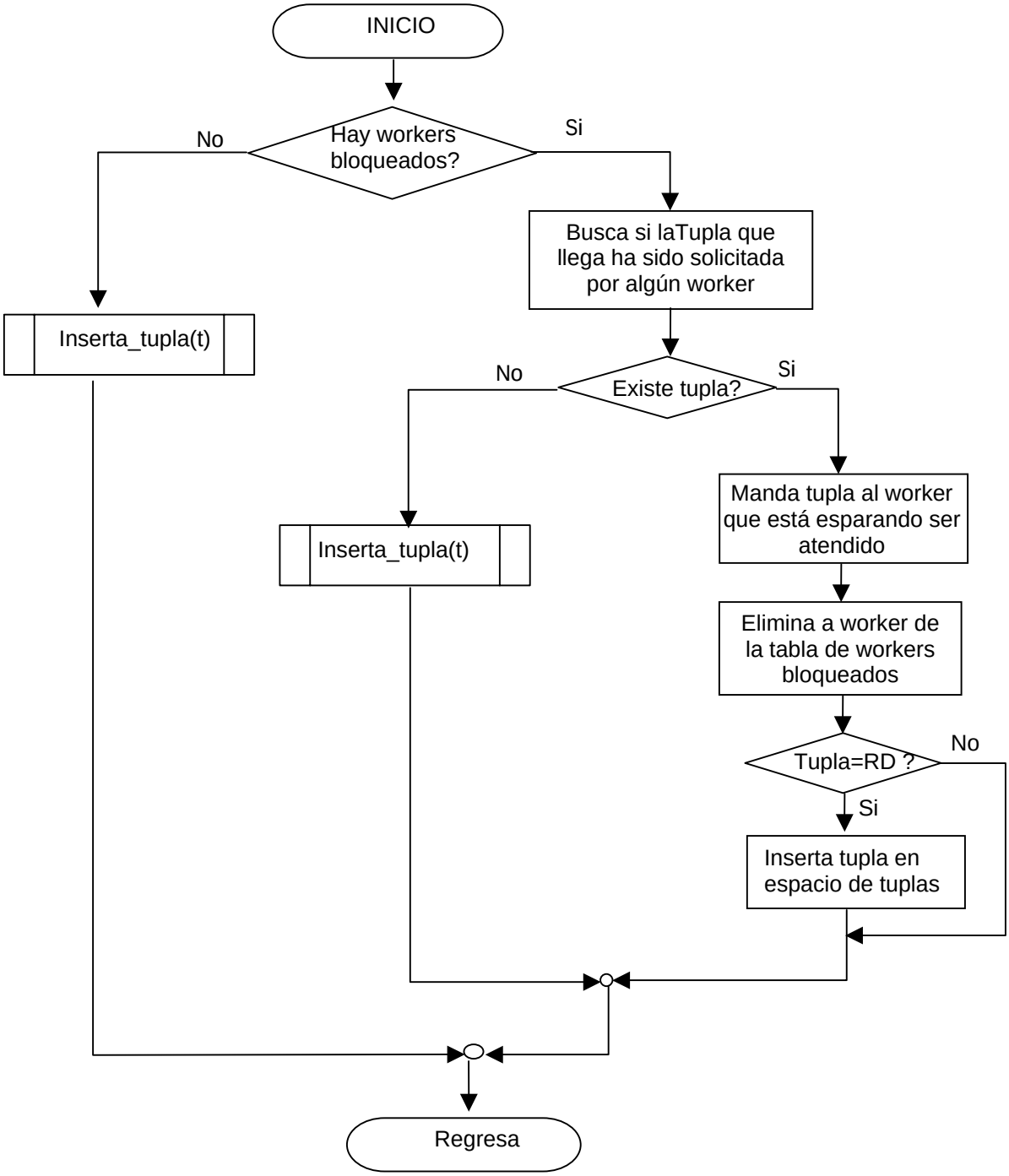


Fig. 3.5 Diagrama de la funcion_out

Esta función recibe como parámetro la tupla que llego al master. Se invoca cada vez que llega una tupla pasiva del tipo OUT, cuya tarea es almacenar tuplas en el espacio de tuplas.

Su primera tarea de esta función será ir a la tabla de worker pendientes para saber si existe algún worker bloqueado. Si encuentra que en la tabla de worker pendientes hay workers registrados entonces empezará a buscar si alguna de las tuplas solicitadas por estos workers corresponde con la que acaba de llegar, de ser así, entonces enviará la tupla al worker que estaba esperando (bloqueado) para ser atendido.

Si en la tabla de bloqueados no encuentra ninguna tupla que coincida con la que llegó, entonces esta última será depositada en el espacio de tuplas.

Si al analizar la tabla de bloqueados, esta se encuentra vacía, entonces directamente se depositará la tupla en el espacio de tuplas.

3.5.4 HILO ATIENDE_TUPLA_ACTIVA

Este hilo se crea una vez que llega una tupla activa al master (tupla EVAL) después de comprobar que el archivo que contiene el código aún no se ha recibido.

El hilo que se crea será el encargado de aceptar conexión (a través del socket destinado a recibir archivos), del desarrollador cada vez que se le solicite atender una tupla activa (código), en donde se tendrá que recibir al archivo que contiene el código que se desea sea asignado a algún worker si existen disponibles o para que sea depositado en el espacio de tuplas.

Una vez que este hilo termina de recibir el archivo correspondiente cerrará los descriptores correspondientes al archivo y al socket.

En la figura 3.6 se muestra el diagrama correspondiente del hilo atiende_tupla_activa.

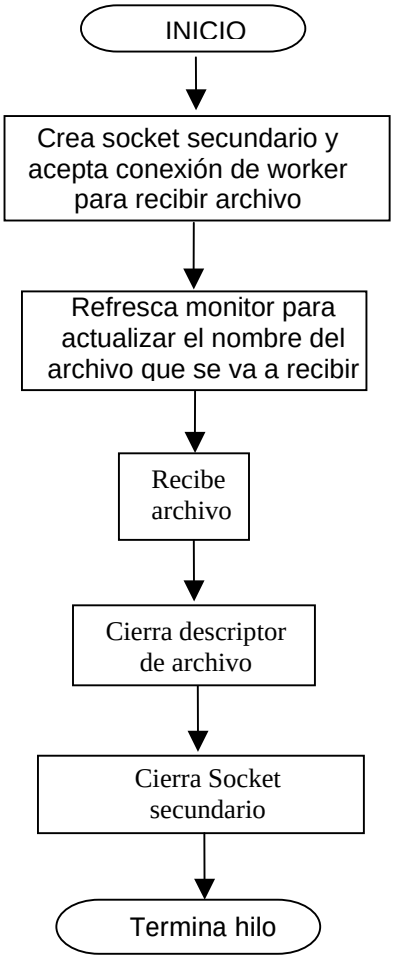


Fig. 3.6 Diagrama del hilo atiende_tupla_activa.

3.5.5 Algoritmo de la función asigna_tarea (tt)

Esta función recibe como parámetro la tupla que llega al master, y es invocada cuando el tipo de tupla corresponde a una tupla activa EVAL. Su función consistirá en asignar la tupla a algún worker que este disponible, o de otra forma insertar la tupla activa en el espacio de tuplas.
En la figura 3.7 se muestra el diagrama de la función asigna tarea.

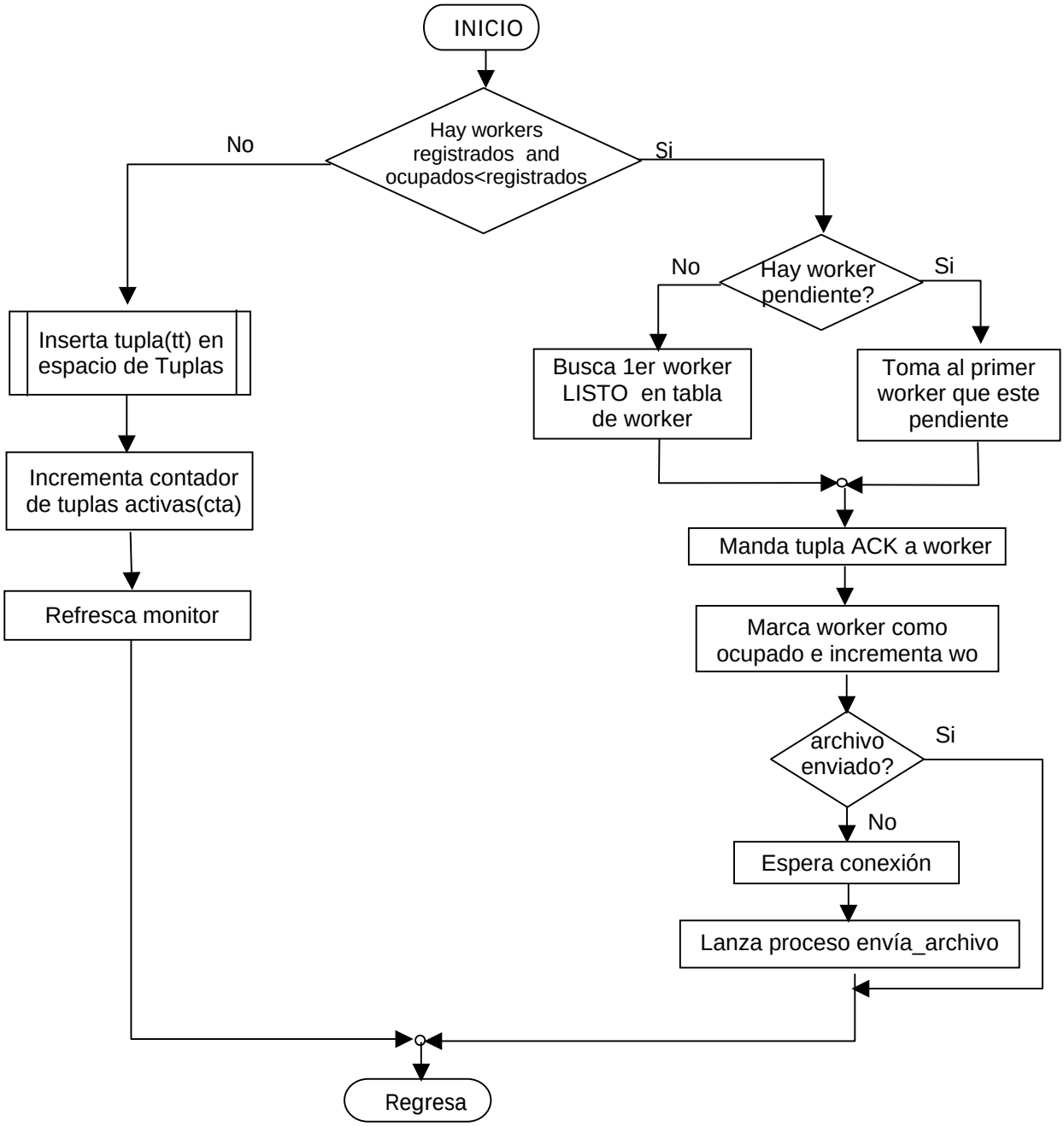


Fig. 3.7 Diagrama de la función asigna_tarea.

Esta función se invoca cada vez que el master recibe una tupla activa (EVAL), y después de que el archivo que contiene el código de la tupla activa ya se ha recibido, esto se realiza con la finalidad de que el master no dedique tiempo para recibir un archivo que ya tiene.

Lo primero que se realiza en este algoritmo es comprobar que existan workers conectados y disponibles, es decir que su estado sea LISTO, si esto se cumple entonces primero buscará en la tabla de workers pendientes (por si existe registrado algún worker que ya terminó su trabajo y al no haber tuplas activas se le registro como pendiente sin que se le haya asignado ninguna tupla activa), de esta forma se busca que a todos los workers se le ponga a trabajar sin dejar workers ociosos, es decir se busca un balance de carga.

Si se encuentra que en la lista de workers pendientes existen workers registrados, se tomará al que haya sido insertado primero, si no hay alguno, entonces se buscará en la tabla de workers y tomará al primero que encuentre en estado de LISTO.

Una vez que se ha localizado a que worker se le va a asignar la tupla activa, se manda una tupla de control tipo ACK al worker indicado (el cual hasta el momento permanece bloqueado en espera de trabajo), esto se realiza como control para que el worker se informe de que ya se le va asignar trabajo y se prepare para recibir el código de la tupla activa que se le va a enviar.

Una vez que se mandó la tupla ACK, se registra al worker con estado de OCUPADO, se incrementa el contador que lleva el total de workers ocupados (wo) y enseguida el master se prepara para aceptar la conexión con el worker y una vez establecida la comunicación se lanza un proceso el cual será el encargado de enviar el código (archivo) al worker correspondiente, para su ejecución.

Si no existen workers disponibles (con estado LISTO) para asignar la tupla activa, entonces se inserta la tupla activa en el espacio de tuplas y el contador de tuplas activas se incrementa.

3.5.6 Algoritmo de la función trabaja_worker (tt, pos)

Este algoritmo se invoca cada vez que llega una tupla ACK de algún worker que ha terminado de atender a la tupla activa que le fue asignada y por lo tanto ha quedado disponible nuevamente (LISTO).

La tarea de esta función es asignarle al worker una nueva tupla activa (si hay disponibles), si no hay tuplas activas disponibles, entonces se debe colocar al worker en la tabla de workers pendientes para que se le tome en cuenta nuevamente en el momento que llegue la siguiente tupla activa y así los workers siempre sean considerados en el momento de asignarles tuplas activas.

En la figura 3.8 se muestra el diagrama de la función trabaja_worker(tt,pos) a la que se le envían como parámetros la tupla recibida por el master y la posición del worker en la tabla de workers.

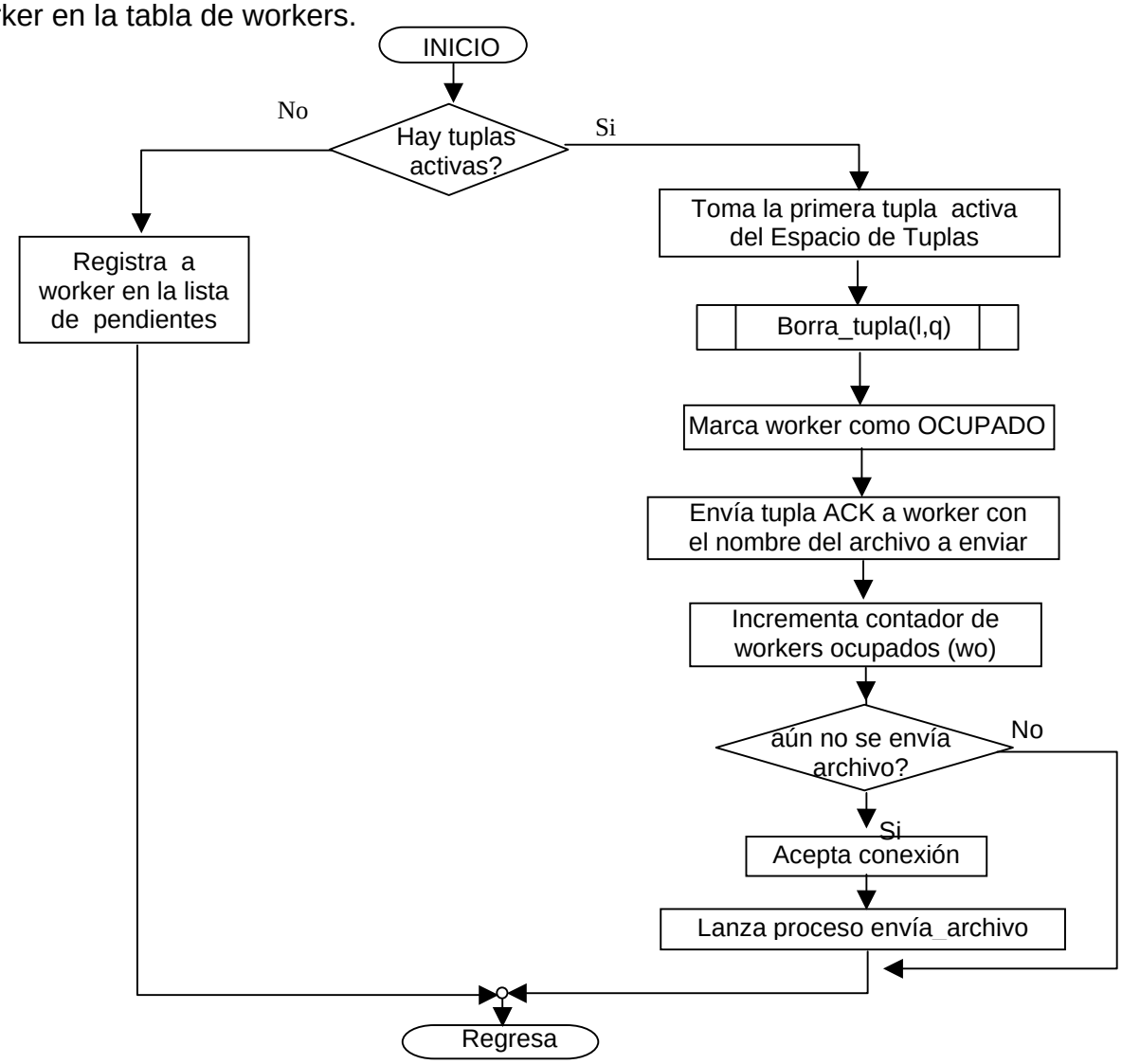


Fig. 3.8 Diagrama de la función trabaja_worker(tt,pos).

Esta función se invoca cada vez que el master recibe una tupla de control ACK o ACK1 por parte de algún worker que se acaba de conectar o bien que se vuelve a presentar como disponible para que se le asigne otra tupla activa (más trabajo).

La función trabaja_worker, empieza por verificar si existen tuplas activas en el espacio de tuplas, que esten pendientes para ser asignadas a algún worker.

Si hay tuplas activas, entonces toma la primera tupla activa que encuentre en el espacio de tuplas, envía una tupla ACK con el nombre de la tupla activa, al worker que envío la petición de trabajo, cambia el estado del worker de LISTO a OCUPADO, y borra la tupla del espacio de tuplas.

Si el código de la tupla activa ya se había enviado anteriormente al worker solicitante, entonces ya no se le envía, simplemente se envía el nombre de la tupla activa para que sea ejecutada nuevamente en el worker.

Si en el espacio de tuplas no hay tuplas activas por el momento, entonces al worker solicitante se le agrega a una lista que contiene información de todos los workers que están terminando y quedan disponibles, para que en cuanto llegue la siguiente tupla activa sean considerados antes de que se consulte a la tabla de workers, y así de esta manera ninguno quede sin trabajar.

3.5.7 Algoritmo de la función Finaliza

Esta función se invoca cada vez que el master recibe una tupla de control ENDA.
La tupla ENDA se utiliza para que desde la máquina desarrollador se informe al master que la aplicación que estaba corriendo ya ha concluido y se han recibido todos los resultados esperados.

Cuando una tupla tipo ENDA llega al master, se inicializa el servidor y las estructuras utilizadas con la finalidad de seguir proporcionando servicio de memoria a quien se lo solicite. También se envían tuplas de control tipo END a cada uno de los workers que se hayan conectado, para informarles de la terminación de la aplicación que se estaba atendiendo, y de esta manera, los workers puedan terminar su sesión de trabajo.

Es importante mencionar de las aportaciones que se tienen con respecto al espacio de tuplas del modelo Linda.

- La computadora en donde reside el master no necesita reiniciarse cada vez que se desee ejecutar una nueva aplicación.
- Los workers al recibir una tupla END también terminan de forma normal, sin tener que dejarlos “zombies”.

En la figura 3.9 se muestra el diagrama de la función finaliza().

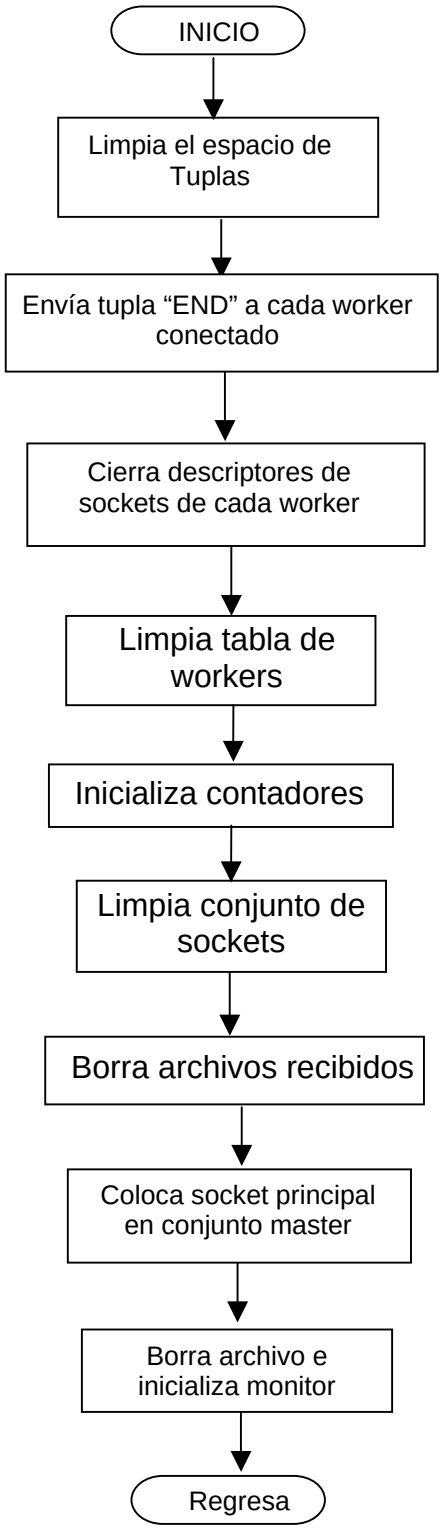


Fig. 3.9 Diagrama de la función finaliza().

Esta función se invoca cada vez que llegue una tupla de control tipo ENDA.
Lo primero que se realiza cada vez que llega este tipo de tupla es ir al espacio de tuplas y limpiarlo para que no tenga ninguna tupla, esto ya sea por descuido del programador o por el uso de las tuplas RD o RDP las cuales no borran las tuplas del espacio de tuplas, y así cada vez que se desee correr una aplicación que haga uso de esta memoria este espacio se encuentre vacío, lo cual ayudará a la corrección de posibles errores de una manera más sencilla.

Una vez que el espacio de tuplas queda vacío se podrá volver a disponer de la memoria.

Consulta el conjunto de descriptores para poder cerrar todos aquellos que se hayan asociado a los sockets de los workers para darles atención a sus solicitudes, enviándoles antes una tupla tipo END a cada uno de los workers que se encuentren conectados.

Se limpian los conjuntos de descriptores utilizados por el servidor para dar atención a cada solicitud, específicamente se limpian los conjuntos master y read_fds.

Coloca como único socket a observar el socket principal asociado al master por el que se atiende solicitudes de conexión y se asigna como el máximo descriptor.

Las listas ligadas utilizadas se inicializan a NULL. Borra el archivo que se envió con las tuplas activas.

Cierra sockets para manipulación de archivos e inicializa el monitor con todos los valores iniciales.

CAPITULO IV

IMPLEMENTACION DE UNA UNIDAD DE MEMORIA VIRTUAL COMPARTIDA CON MONITOREO (MEMVIR)

4.1 INTRODUCCION

El desarrollo de este proyecto se realizó en el lenguaje de programación C por ser un lenguaje de los más utilizados para el desarrollo de sistemas y como plataforma de desarrollo el sistema operativo Linux Mandrake 7.2 y 9.2, el cual se basa fundamentalmente en tecnología abierta con enormes capacidades para trabajar sistemas de computación paralela.

Para probar este proyecto se utilizaron computadoras conectadas en una red, con procesador Pentium IV 1.6 GHz, con 128 Megabytes en memoria RAM, disco duro de 80 GB, una unidad de 3 1/2, dos Pentium III 450 MHz, con 64 Megabytes en memoria RAM, disco duro de 40 GB, una unidad de 3 ½, 3 Pentium I 133 MHz, con 80 Megabytes en memoria RAM, disco duro de 2 GB, una unidad de 3 ½ , tarjetas de red Ethernet 10/100 Mbps, y un Hub Ethernet a 10 Mbps (para realizar la conexión).

Como se mencionó anteriormente este proyecto está enfocado básicamente al desarrollo de una memoria virtual compartida, la cual ofrece atender solicitudes de memoria bajo el modelo espacio de tuplas, es decir, específicamente estamos hablando de un servidor de tuplas (Una tupla es un simplemente un conjunto ordenado de datos con tipo).

Entre las alternativas para el desarrollo del servidor se optó por diseñar un servidor basado en sockets no bloqueantes o también conocidos como servidores dirigidos por eventos, los cuales basan su funcionamiento en la utilización de lecturas y escrituras asíncronas sobre descriptores –normalmente sockets. A este método de lectura y escritura asíncrona sobre descriptores es a lo que se le conoce como entrada/salida no bloqueante.

Dado que los servicios que presta este servidor no son demasiado complicados se implementó un servidor interactivo el cual se encuentra a la espera de solicitudes por parte de los clientes, y en cuanto llega una petición las atiende el mismo. Para la comunicación en la red se utilizaron sockets orientados a conexión en el dominio de Internet, para la transmisión de protocolos TCP, ya que los procesos a comunicar no están en la misma computadora.

4.2 ESTRUCTURAS DE DATOS UTILIZADAS

En la implementación de este proyecto se utilizaron las siguientes estructuras de datos:

- Para el espacio de tuplas se utilizó una lista ligada en donde cada nodo de la lista corresponde a una tupla que tiene la siguiente distribución:

```
struct tupla
{
   int tipo;
   char ip[15];
   struct datos argumentos[7];
   int tot_dat;
  struct tupla *sig;
};
```

En donde el campo ***tipo*** se utilizó para indicar el tipo de tupla que se va a atender. Entre los tipos que se manejan en el servidor tenemos los siguientes

- Tuplas de control: **ACK1** (se envía cuando se conectan cada uno de los trabajadores para ser registrados), Tupla de control **ACK** (se envía cuando los workers terminan su tarea y solicitan mas trabajo), Tupla de control **ENDA** (se envía por el desarrollador para indicar que su aplicación ha terminado y no requiere mas tuplas), estas tuplas se utilizan solamente para control del servidor, para poder reiniciar y dar atención a nuevas solicitudes.

- Tuplas pasivas: **IN,RD,INP,RDP** (se utilizan para solicitar al servidor tuplas del espacio de tuplas), y por el otro lado se tiene a la tupla pasiva **OUT** (se utiliza para enviar tuplas al espacio de tuplas).

- Tuplas activas: **EVAL**, es la tupla que se utiliza para enviar código al servidor, y asignarselo a algún trabajador disponible

El campo ***ip*** se utiliza para tener un seguimiento de la IP de la máquina de donde esta llegando la tupla.

El campo ***argumentos***, contiene los datos a solicitar o enviar dentro de la tupla al espacio de tuplas.

El campo ***tot_dat***, se utiliza para conocer cuantos argumentos tiene la tupla para poder realizar la búsqueda y comparaciones correspondientes, a la hora de que se hace una solicitud de tupla del espacio de tuplas.

El campo **sig**, se utiliza para ir ligando las tuplas en el espacio de tuplas.

Para tener acceso al espacio de tuplas el cual se implementó como una lista, se utilizó la variable: struct tupla ***TS***

- Para ir registrando la información relacionada con cada una de las computadoras que se conectaban para solicitar tuplas, específicamente para registrar a los workers que se van conectando al servidor, se utilizó una tabla (arreglo de estructuras), en la cual se registra la siguiente información:

```
struct worker
{
    char ipw[15];
    int edo,ds,env,pp;
 } twork[100];
```

En el campo ***ipw***, se regitra la IP desde donde está llegando la tupla, esta estructura se actualiza cada vez que los workers mandan su primera tupla de control ACK1, para ser registrados y en caso de no haber tuplas activas poder tenerlos disponibles para asignarles la primera tupla activa en el momento que llegue.

El campo ***edo*** se utiliza para ir actualizando los diferentes estados por los que pasa un worker y que son: OCUPADO(O) , LISTO (L) o BAJA(X).

Los criterios considerados para que un worker vaya cambiando de estado son los siguientes:

- El worker inicialmente está en estado de LISTO al registrarse, lo cual se realiza cuando el master recibe una tupla de control tipo ACK1.
- Si al momento de registrarse el worker hay tuplas activas inmediatamente se le asigna la primera que se encuentre en el espacio de tuplas y cambiando enseguida su estado a OCUPADO.
- Si por algún motivo un worker rompe su conexión con el master entonces se da de BAJA del sistema y se marca con una X en el monitor.

El campo ***ds*** se utiliza para registrar el identificador de socket por el cual el master mantiene comunicación con el worker registrado.

El campo ***env*** se utiliza para controlar que el envío del archivo correspondiente a la tupla activa no se envíe más de una vez.

El campo ***pp*** se utiliza simplemente para control en el monitor de la información relacionada con el worker.

La tabla que tiene toda la información se declaró como: ***twork***.

- Para registrar la información de los workers que enviaron su tupla de solicitud de trabajo y no fueron atendidos por no haber tuplas activas, se utilizó una estructura de workers pendientes con la siguiente información:

```
struct workerp
{
    char ip[15];
    int s;
    struct workerp *sig;
}*lwp=NULL, *fin;
```

El campo ***ip***, se utiliza para conocer la IP del worker que hizo la solicitud de trabajo.

El campo ***s*** se utiliza para conocer el socket por el cual llegó la solicitud de trabajo.

lpw es el apuntador a la lista de workers pendientes.

4.3 DISEÑO FINAL DEL PROYECTO

Se desarrolló un programa llamado ***master*** el cual es un servidor interactivo con entrada/salida no bloqueante, que atiende solicitudes de memoria bajo el esquema espacio de tuplas. Este servidor basa su funcionamiento en la utilización de lecturas y escrituras asíncronas sobre descriptores asociados a sockets, esto con la finalidad de que el servidor pueda dar atención a cada solicitud que le llegue de cada uno de los workers que se encuentren conectados al servidor.

Además del socket principal por el cual se van a estar atendiendo solicitudes de conexión se crean otros 2 sockets, uno para recibir el código asociado a una tupla activa del desarrollador y otro para que se envíe este código del master hacia el worker al cual se le haya asignado la tupla activa. En el siguiente código se tiene la creación de los tres descriptores de sockets que inicialmente se crean.

```
//********** CREA SOCKET PARA ENVIAR ARCHIVO**********
s1 = socket(AF_INET, SOCK_STREAM,IPPROTO_TCP);
  if(s1!=-1)
  { bs1.sin_family=AF_INET;
    bs1.sin_port=htons(1850);
    bs1.sin_addr.s_addr=htonl(INADDR_ANY);
    if(bind(s1,(struct sockaddr*)&bs1,sizeof(bs1))!= -1)
      listen(s1,MAX_CONN);
  }
```

Este socket al igual que los otros, son orientados a conexión en el dominio de Internet, para la transmisión por protocolo TCP; el puerto asociado al socket para el envío de archivos hacia los workers es el **1850**.

```
//************ CREA SOCKET PARA RECIBIR ARCHIVO **************
  socket2=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
  if(socket2!=-1)
  { bs2.sin_family=AF_INET;
    bs2.sin_port=htons(1234);
    bs2.sin_addr.s_addr=htonl(INADDR_ANY);
    if(bind(socket2,(struct sockaddr*)&bs2,sizeof(bs2))!= -1)
      listen(socket2,MAX_CONN);
    }
```

El puerto asociado al socket para recibir archivos del desarrollador es el **1234**

```
//*************** CREA SOCKET PRINCIPAL *************
  s = socket(AF_INET, SOCK_STREAM,IPPROTO_TCP);
  if (s != -1)
  {
    bs.sin_family = AF_INET;
    bs.sin_port = htons(8080);
    bs.sin_addr.s_addr = htonl(INADDR_ANY);
   if( bind(s,(struct sockaddr*)&bs, sizeof(bs)) != -1)
      listen(s, MAX_CONN);
```

El puerto asociado al socket para atender solicitudes de conexión es el **8080**.

Una vez que los sockets han sido creados, el socket principal ***s*** se agrega a un conjunto que será el que continuamente se esta actualizando con todos los descriptores de sockets a observar y se asigna como el máximo descriptor abierto.

Posteriormente a esta acción el servidor entra en un ciclo infinito en el que continuamente estará observando el conjunto de descriptores, utilizando la función *select()* la cual comprueba "conjuntos" de descriptores; en concreto el conjunto *read_fds* listos para lectura, el cual de manera inicial solo contiene el descriptor del socket principal por el que escucha solicitudes el servidor.

Después de la llamada a select el conjunto read_fds contendrá los descriptores que están listos para lectura.

En la llamada a la función select(), se pueden indicar 3 conjuntos de descriptores de sockets, en este caso para el diseño del servidor solo se utiliza uno, el que contiene los descriptores para lectura, por lo tanto en los dos argumentos siguientes se coloca NULL, el último NULL corresponde al temporizador, esto con

la finalidad de que el servidor espere hasta que algún descriptor de archivo este listo con alguna solicitud.

En el siguiente fragmento de código se realizan estas acciones.

```
FD_SET(s,&master);
fdmax=s;
while (1)
{
    read_fds=master;
    if(n=select(fdmax+1,&read_fds,NULL,NULL,NULL)==-1)
      {  perror("select"); exit(1) }
```

Cada vez que se regresa de la llamada a la función select() el conjunto read_fds se modifica para reflejar que sockets están listos para lectura, dada esta situación para no perder el conjunto de descriptores de lectura, antes de realizar el llamado a la función select(), se copia este conjunto *read_fds* en otro llamado *master*, el cual siempre contendrá todos los descriptores de archivo que están actualmente conectados, incluyendo el descriptor del socket que esta escuchando para nuevas conexiones.

Una vez que en el conjunto *read_fds* tenemos a los descriptores que están listos para lectura, el master entra en un ciclo a través del cual atenderá a cada una de las solicitudes. Esto se muestra en el siguiente código.

```
for(i=0;i<=fdmax;i++)
{
   if(FD_ISSET(i,&read_fds))
    {
        if(i==s)
         {
            if((s_aux = accept (s,(struct sockaddr*) &ino,&sd))==-1)
                perror("ACEPT");
            else
            {
               FD_SET(s_aux,&master);
                if(s_aux>fdmax)
               fdmax=s_aux;
            }//ELSE
          }//if
```

Aquí se observa como la variable *fdmax* (la cual contiene al máximo descriptor abierto), se utiliza para controlar el ciclo.

Debido a que de los descriptores que haya abiertos, solo nos interesan los que tiene el conjunto read_fds, se utiliza la macro FD_ISSET con este conjunto, la cual pregunta si el descriptor *i* está en este conjunto.

Si *i* es un descriptor del conjunto read_fds, lo que sigue es diferenciar si se trata de una solicitud de conexión al master o si es una tupla que llega de algún worker y esto se realiza comparando el descriptor *i* (el cual está listo para lectura) con el socket principal del servidor, si son iguales, entonces se acepta la conexión y se crea el socket (s_aux), por el que se va estar atendiendo las solicitudes del worker recién conectado, además también se agrega al conjunto master para que sea incluido en la siguiente comprobación de sockets listos para lectura.

Si el socket listo no corresponde al socket principal, esto indica que es de algún worker, y lo primero que se comprueba es que el worker envíe su solicitud.

```
else
{
   // datos de un worker
  if((num=recv(i,&t, sizeof(struct tupla),0))<=0)
  {
     if(num==0)                                    //conexión cerrada
     {
       f=0;noencontrado=1;
       while(f<tw && noencontrado)
       {
           if(twork[f].ds==i)
               noencontrado=0;
           else f++;
        }//WHILE
```

Si recv() devuelve 0,indica que el worker ha cerrado la conexión. Esto implica que la información del worker se debe de eliminar de la tabla de workers conectados (*twork*), de la lista de pendientes (*lwp*) si es que se encuentra ahí cerrar el descriptor asociado a él para liberarlo y eliminarlo del conjunto master esto se realiza con la macro FD_CLR. El siguiente fragmento de código muestra esto.

```
for(k=f;k<(tw-1);k++)
  twork[k]=twork[k+1];
```

Aquí a partir de la posición que tenía el worker en la tabla de worker, se realiza un recorrido para eliminarlo.

También se busca si el worker está en la lista de pendientes y de ser así se elimina de la lista.

```
qq=lwp;l=NULL; tw--;
while(qq!=NULL && qq->s != i)
{  l=qq; qq=qq->sig; }
if(qq!=NULL)
{  if(l==NULL) lwp=qq->sig;
    else l->sig=qq->sig;
  free(qq);
}
```

Si lo que devuelve recv() es menor que cero entonces se despliega un mensaje de error de recv(), se cierra el descriptor asociado al socket y se elimina del conjunto master con la macro FD_CLR, como se muestra en el siguiente fragmento de código.

```
    }//if num=0
else {printf("Error de recv\n"); }
close(i);
FD_CLR(i,&master);
```

Ahora si lo que devuelve recv() es mayor que cero, entonces se recupera la tupla (t) del socket y se atiende, como se muestra en el código siguiente.

```
if(t.tipo==ENDA)
{    finaliza();
      break;
 }
else
    atiende_tupla(t);
```

En este caso primero se comprueba si la tupla que llega es del tipo ENDA entonces llama a la función finaliza() para reiniciar al master, de otra forma llama a la función atiende_tupla donde se dará atención a cada una de las tuplas que llegan al master.

De esta manera es como va a estar operando este servidor master para dar servicio a las solicitudes de tuplas (memoria) que le lleguen de cualquier computadora que se conecte a él.

En seguida se muestran partes del código desarrollado para las funciones más importantes que forman parte del servidor y se da una breve descripción de su funcionamiento.

Función atiende_tupla

En esta función se analiza el tipo de tupla que llegó para atender el servicio solicitado.

Si el tipo de la tupla corresponde a una tupla pasiva IN, RD, INP o RDP, se enviará a una función que realizará la búsqueda en el espacio de tuplas.

Si el tipo de la tupla tupla es OUT, entonces se envía al espacio de tuplas con la llamada a la función_out().

Enseguida se muestra el código de la función atiende_tupla()

```
void atiende_tupla(struct tupla tt)
{  struct tupla t2,tpla;
   int num,f,noencontrado,pos,sk,fp;
   char *valor=NULL;
  switch(tt.tipo)
  { case OUT:  funcion_out(tt);  ctp++;
                  //refesca monitor
                  break;
    case IN: case RD:  case RDP:
    case INP:   funcion_busca(tt);
                  //refresca monitor
                  break;
    case EVAL:
              if(fp=open(tt.argumentos[0].valor.s,O_RDONLY)<0)
              {   if(strcmp(tt.ip,ipd)!=0)
                    { ipd=tt.ip;
                      //refresca monitor
                    }
                 pthread_create(&tid,NULL,(void*)&atiende_tupla_activa,(void*)&tpla);
                pthread_join(tid,(void **)&valor);
                }
           asigna_tarea(tt);
          break;
      case ACK1:
                  strcpy(twork[tw].ipw,tt.ip);
                  twork[tw].ds=i;
                  twork[tw].env=0;
                  pos=tw;
                  twork[tw].edo=LISTO;
                //refresca monitor
                  trabaja_worker(tt,tw);
                  tw++;
                  break;
      case ACK:     wo--;
                  f=0;noencontrado=1;
                  while(f<tw && noencontrado)
                  {   if(strcmp(tt.ip,twork[f].ipw)==0)
                      {   noencontrado=0;      pos=f;
                          twork[pos].edo=LISTO;
                         //refresca monitor
                       }
                      else f++;
                   }//WHILE
                   if(noencontrado==0)
                   trabaja_worker(tt,pos);
                  break;
      }
}
```

La función atiende_tupla(tt) recibe como parámetro la tupla que es solicitada por el worker, y se analiza para dar la atención correspondiente.

Si el tipo de la tupla corresponde a una tupla pasiva OUT, se hace un llamado a *funcion_out( ),* la cual se ocupa de almacenar las tuplas que llegan dentro del espacio de tuplas..

Si el tipo de la tupla corresponde a una tupla pasiva IN, INP, RD, RDP, entonces se llama a la función *busca( ),* la cual se encarga de buscar la tupla solicitada en el espacio de tuplas.

Si la tupla es una tupla activa EVAL, y el archivo con el código asociado a la tupla aún no se ha enviado al master, entonces se toma la IP del desarrollador (para refresco del monitor) y se lanza un hilo que se encargará de recibir el archivo. Esta comprobación se realiza con la finalidad de no recibir más de una vez el mismo archivo.

Una vez que el archivo está en el servidor se llama a la función *asigna_tarea( ),* la cual es la encargada de asignar la tupla activa que llega, al primer worker disponible que encuentre.

Si la tupla es una tupla de control ACK1, como esta tupla se utiliza cuando cada worker se acaba de conectar para solicitar trabajo, en este momento el master tomará todos los datos que necesita y los registra en la tabla de worker (twork), una vez que se ha registrado al worker, se llama a la función trabaja_worker( ), la cual se encarga de asignarle la primera tupla activa que encuentre en el espacio de tuplas.

Si la tupla es una tupla de control tipo ACK, la cual se utiliza cada vez que un worker termina de ejecutar el código de la tupla activa, y vuelve a solicitar nuevamente trabajo, se actualiza su estado como LISTO y se llama a la función trabaja_worker.

función_out( )

Como se mencionó esta función se utiliza para almacenar tuplas en el espacio de tuplas, o enviar la tupla que llega hacia algún worker, siempre y cuando se encuentre registrado en la tabla de bloqueados (tpb) y además la tupla por la que se bloqueó sea la tupla que acaba de llegar.

Para saber si las tuplas son iguales se procede a realizar una serie de comparaciones empezando por el número de argumentos.

Para mayor ilustración se muestra parte del código de esta función.

```
void funcion_out(struct tupla tt)
{ int j,x, encontrado,v,k,enc,y;
 struct tupla t2,*q;
 if(npb)
 { // buscar el proceso bloqueado por la tupla que se acaba de insertar
  j=0; encontrado=0; enc=1;
  while(j<npb && encontrado==0)
  {  if(strcmp(tpb[j].tt.argumentos[0].valor.s,tt.argumentos[0].valor.s)==0)
     {  if(tpb[j].tt.tot_dat==tt.tot_dat)
        {  x=1; enc=1;
         while(x<tt.tot_dat && enc)
         {  if(tpb[j].tt.argumentos[x].tacceso==VALOR)
            {
               switch(tt.argumentos[x].tvar)
               {  case CHAR_TYPE:
                        if(tpb[j].tt.argumentos[x].valor.c!=tt.argumentos[x].valor.c)
                        enc=0; break;
                  case INT_TYPE:
                        if(tpb[j].tt.argumentos[x].valor.i!=tt.argumentos[x].valor.i)
                        enc=0; break;
                  case FLOAT_TYPE:
                         if(tpb[j].tt.argumentos[x].valor.f!=tt.argumentos[x].valor.f)
                         enc=0; break;
                  case STRING_TYPE:
                        if(strcmp(tpb[j].tt.argumentos[x].valor.s,tt.argumentos[x].valor.s)!=0)
                        enc=0; break;
                  case TOT_ARR:
                        if(tpb[j].tt.argumentos[x].valor.td!=tt.argumentos[x].valor.td)
                        enc=0; break;
               }//switch
             }//if  //por valor
            else
            {  if(tpb[j].tt.argumentos[x].tvar!=tt.argumentos[x].tvar)
               enc=0;
             }
             x++;
          }//while x<tot_dat
       if(enc)
       {  encontrado=1;
          if(tpb[j].tt.tipo==RD)
          inserta_tupla(tt);
          v=tpb[j].ds;
        }
        else j++;
      }//if de tot_dat
      else
        j++;
    }//if del nombre
    else
      j++;
}//while
```

Hasta este punto lo que se hace si hay procesos bloqueados, comparar las 2 tuplas: la del worker bloqueado y la que le llegó al master; para decir que la tupla corresponde a la que el worker esta esperando se deben cumplir los puntos siguientes:

- Igual nombre, lo primero que se verifica es que sea la tupla deseada, si no corresponde con el nombre, sale de la comparación.
- Igual número de argumentos, si no se cumple, entonces la tupla no es la que se espera y sale de la comparación.
- Se comparan uno a uno cada argumento y verificar que igualen con respecto a lo siguiente:
    - Si el argumento es por valor, entonces deben tener ambas el mismo valor.
    - Si el argumento es por referencia, entonces debe ser del mismo tipo de dato.

  Si alguno de estos puntos no se cumple entonces sale de la comparación.

Si al salir de la comparación, el resultado de la búsqueda fue satisfactorio, se analiza si la tupla con la que se hizo la solicitud fue del tipo RD, entonces en ese caso la tupla que llegó además de que va a dar respuesta al worker que espera por ella, también se va insertar al espacio de tuplas. Recordemos que RD sólo solicita una copia de la tupla, por lo tanto la tupla debe estar en el espacio de tuplas.

Además de enviar la tupla, se debe borrar de la tabla de bloqueados al worker que acaba de ser atendido, lo cual se muestra en el siguiente código.

```
for(k=j;k<(npb-1);k++)
   tpb[k]=tpb[k+1];
npb--;
send(v,&t2,sizeof(struct tupla),0);
```

Si el resultado de la búsqueda no fue satisfactorio, entonces la tupla aún no ha sido solicitada y por lo tanto es depositada en el espacio de tuplas.

```
else
  inserta_tupla(tt);
```

Función busca( )

Esta función se encargará de buscar en el espacio de tuplas la tupla solicitada.

Para esto se realizan una serie de comparaciones como en el caso de la función_out, pero en este caso con las tuplas que se encuentran en el espacio de tuplas. Si la comparación es satisfactoria entonces se obtiene la tupla del espacio de tuplas, para tener la tupla que se va a mandar al worker que hizo la solicitud.

A continuación se muestra parte de la función busca().

```
void funcion_busca(struct tupla tt)
{ struct tupla *q,*l, t2;
 int encontrado,enc,x,y;
 q=TS;  l=NULL;  encontrado=0;  enc=1;
while(q!=NULL && encontrado==0)
{      //compara las tuplas y si en algún momento no coinciden cambia enc=0
  } //while
 if(enc)
    encontrado=1;
if(encontrado)     //forma la tupla a enviar
{  for(x=0;x<tt.tot_dat;x++)
   {
      switch(q->argumentos[x].tvar)
     {
        case CHAR_TYPE:
             t2.argumentos[x].valor.c=q->argumentos[x].valor.c;
             break;
        case STRING_TYPE:
             strcpy(t2.argumentos[x].valor.s,q->argumentos[x].valor.s);
             break;
         case INT_TYPE:
              t2.argumentos[x].valor.i=q->argumentos[x].valor.i;
             break;
         case FLOAT_TYPE:
              t2.argumentos[x].valor.f=q->argumentos[x].valor.f;
              break;
         case TOT_ARR:
              t2.argumentos[x].valor.td=q->argumentos[x].valor.td;
              break;
          case INT_ARR:
              for(y=0; y<t2.argumentos[x-1].valor.td; y++)
               t2.argumentos[x].valor.ve[y]=q->argumentos[x].valor.ve[y];
               break;
          case FLOAT_ARR:
              for(y=0; y<t2.argumentos[x-1].valor.td; y++)
              t2.argumentos[x].valor.vf[y]=q->argumentos[x].valor.vf[y];
              break;
       }//switch
       t2.argumentos[x].tacceso=tt.argumentos[x].tacceso;
    }//for
   switch(tt.tipo)
   {
     case IN: t2.tipo=IN; break;
     case RD: t2.tipo=RD; break;
     case INP: t2.tipo=INP; break;
     case RDP: t2.tipo=RDP; break;
    }

    t2.tot_dat=tt.tot_dat;
    send(i,&t2,sizeof(struct tupla),0);
```

Esta función lo primero que realiza es verificar que el espacio de tuplas no este vacío. Si esta condición se cumple, entonces se realizan una serie de comparaciones semejantes a la funcion_out() para comprobar si las tuplas igualan, si en algún momento encuentra una tupla que iguale a la solicitada, entonces para cada uno de los argumentos por referencia se realizan sustituciones por los valores correspondientes en el espacio de tuplas hacia la tupla solicitada, de tal forma que al terminar, la tupla solicitada este disponible para ser enviada al worker que la solicitó.

Una vez que la tupla es enviada, se analiza el tipo de tupla atendida, y si la tupla atendida fue tipo IN o INP, entonces la tupla se borra del espacio de tuplas, como se muestra en el código siguiente:

```
if(t2.tipo==IN || t2.tipo==INP)
{  if(l!=NULL)
      l->sig=q->sig;
    else
      TS=q->sig;
    free(q);
 } // del IF
```

Si la tupla no fue encontrada y el tipo de tupla solicitada fue un IN o RD, entonces, el worker que hizo la solicitud se agrega en la tabla de bloqueados, es decir en la tabla declarada como *tpb*.

```
if( tt.tipo==IN || tt.tipo==RD)
{
   tpb[npb].tt=tt;
   tpb[npb].ds=i;
   npb++;
 } // in,rd
```

Si la tupla no fue encontrada y el tipo de tupla solicitada fue un INP o RDP, entonces, se manda una tupla de control 0, con el nombre “NO EXISTE” al worker que hizo la solicitud.

```
 if(tt.tipo==INP || tt.tipo==RDP)
  { t2.tipo=0;
    strcpy(t2.argumentos[0].valor.s,"NO_EXISTE");
    send(i,&t2,sizeof(struct tupla),0);
  }//if INP RDP
```

Función asigna_tarea

Esta función es la encargada de asignar las tuplas activas que llegan al master a los workers disponibles (LISTOS).

```
void asigna_tarea(struct tupla tt)
{ int j=0;int asignada=0, noencontrado;struct workerp *q;

 if(tw>0 && wo<tw)
 { j=0; noencontrado=1;
    if(lwp!=NULL)
     {
        while(j<tw && noencontrado)
        { if(strcmp(twork[j].ipw,lwp->ip)==0)
                { noencontrado=0; asignada=1;}
                else j++;
        }//while
        if(noencontrado==0)
         {  q=lwp;
            lwp=lwp->sig;
            free(q);
         }//if
      }//if lwp
   else
   while (!asignada && j<=(tw-1))
   {
       if(twork[j].edo)
          asignada=1;
       else j++;
     } //WHILE
  }// if tw>0
if(asignada)
 {  tt.tipo=ACK;
    send(twork[j].ds,&tt,sizeof(struct tupla),0);
    twork[j].edo=OCUPADO;
    //refresca monitor
     if(twork[j].env==0)
    {  twork[j].env=1;
        conecta();
        if(fork()==0)
        {   envia_archivo(tt.argumentos[0].valor.s,j);
            exit(1);
          }
        close(s_aux1);
       }
        //refresca monitor
       wo++;
  }//if asignada

 else
 {
     inserta_tupla(tt);
     cta++;
    //refesca monitor
  }
```

Esta función lo primero que realiza es verificar que existan workers disponibles, es decir que ya exista alguno que se haya conectado y además que su estado sea LISTO, si estas condiciones se cumplen indican que algún worker ya esta esperando por la asignación de una tupla activa. En este caso ***tw*** es un contador para el número de conexiones activas en el master y ***wo*** es un contador para el número de worker que están en estado OCUPADO.

Si hay workers disponibles primero va a buscar en la lista de pendientes, por si ya anteriormente se había solicitado una tupla activa que no fue asignada por no haber disponibles y de esta manera lograr una asignación equilibrada de trabajo, sobre todo al momento de reasignar el trabajo. La tabla de pendientes esta controlada por la variable ***lwp***.

Si la lista de pendientes no esta vacía, se toma al primero de la lista, se le ubica en la tabla de worker para actualizar su estado, y tomar la información para enviar la tupla activa. Posteriormente se elimina de la lista de pendientes.

Si por otro lado, la lista de pendientes estuviera vacía, entonces simplemente se busca en la tabla de worker al primero que se encuentre disponible, es decir LISTO.

Una vez que ya se tiene los datos del worker al que se le va a asignar la tupla activa, se le envía una tupla tipo ACK, para indicarle al worker que ya se le va a atender y que se conecte por el socket indicado para recibir el archivo asociado a la tupla activa. Enseguida se acepta la conexión y se envía el archivo.

Si no se encontró ningún worker disponible, entonces la tupla activa se agrega al espacio de tuplas, y se incrementa el contador de tuplas activas ***cta***.

Función trabaja_worker( )

Esta función se utiliza cada vez que un worker solicita trabajo, ya sea la primera vez o si se le va a reasignar trabajo.

Antes de asignarle trabajo o una tupla activa, lo primero que realiza es verificar si hay tuplas activas en el espacio de tuplas (cta >0).Si hay tuplas activas, entonces toma la primera que encuentre en el espacio de tuplas, al worker que esta haciendo la solicitud se le cambia el estado a OCUPADO.

Una vez que actualizó lo anterior, le manda una tupla tipo ACK, para que el worker este listo para recibir el archivo y le envía el archivo.

Si por el contrario el valor de la variable cta es 0, entonces indica que no hay tuplas activas, así que la solicitud no se atiende, y el worker se agrega en la lista de pendientes para su posterior asignación.

Enseguida se muestra el código de la función.

```
void trabaja_worker(struct tupla tt, int p)
{  struct tupla *q,*l;
   struct workerp *r;
   int encontrado=0;
   if(cta>0)
   { // buscar la primera tupla activa y recuperar el nombre del archivo
     // a enviar y activar envia_archivo(nom_arch); y elimina tupla
     // activa del ET.
     q=TS; l=NULL;
     while(q!=NULL && encontrado==0)
     { //busca la primera tupla activa
       if(q->tipo==EVAL)
        { encontrado=1;
        strcpy(tt.argumentos[0].valor.s,q->argumentos[0].valor.s);
        }
       else
         {l=q;q=q->sig;}
      }//while
      if(encontrado)
      {   borra_tupla(l,q);
          twork[p].edo=OCUPADO;
             tt.tipo=ACK;
          send(twork[p].ds,&tt,sizeof(struct tupla),0);
          wo++;
            mvwprintw(chw2,twork[p].pp,13,"O"); wrefresh(chw2);
            if(twork[p].env==0)
          {
            twork[p].env=1;
              conecta();
              if(fork()==0)
                 {          envia_archivo(tt.argumentos[0].valor.s,p);
                            exit(1); }
                }
           close(s_aux1);
          //refresca monitor
            }//if encontrado
    }//if cta>0
   else
   { r=malloc(sizeof(struct workerp));
     strcpy(r->ip,twork[p].ipw);
     r->s=i;
     r->sig=NULL;
     if(lwp==NULL)
       { lwp=r;  fin=r;}
     else
       {fin->sig=r; fin=r;}
   }//else
}
```

Función finaliza ( )

Esta función es importante ya que en ella se tiene una de las aportaciones al software master, con respecto al modelo Linda, ya que en este sistema una vez que se termina una aplicación, el master, como servidor sigue atendiendo solicitudes, sin necesidad de tener que iniciarlo cada vez que se desee correr una nueva aplicación. Además de que cada vez que termina una aplicación el master revisará en el espacio de tuplas, y si existen tuplas, las elimina para iniciar con el espacio vacío para atender una nueva aplicación, con menor riesgo de errores.

Esta función se va a invocar cada vez que el master reciba una tupla de control tipo ENDA del desarrollador. Enseguida se muestra parte del código de esta función.

```
void finaliza()
{  struct tupla *aux , *q,tt;
   int conta=0,l;
   x=5; xx=5; xxx=5;
   q=TS;
   if(q!=NULL)
   {   while(q!=NULL)
       { conta++;
         TS=q->sig;
         free(q);  q=TS;
       } //WHILE
    }// if(q!=NULL)
   for(l=0;l<=fdmax+1;l++)
   { if(FD_ISSET(l,&master))
     { if (l!=s )
       {  tt.tipo=END;        //Envia Fin para que Terminen los workers
            strcpy(tt.argumentos[0].valor.s," WORKER... ");
          send(l,&tt,sizeof(struct tupla),0);
          close(l);
        }
      }
    }//for
    for(l=0; l<tw; l++)
      twork[l].edo=LISTO;
    tw=0; npb=0; cta=0; wo=0; ctp=0; ta=0; xxx=5; con=0;
    FD_ZERO(&master);       //Limpia conjuntos de descriptores
    FD_ZERO(&read_fds);
    FD_SET(s,&master);     //coloca en el conjunto unicamente al socket principal
    fdmax=s;
    TS=NULL; lwp=NULL; ipd="nadie";
    strcpy(borra,"rm -f ");
    strcat(borra,nomarchv);      sleep(5);
    system(borra);
    close(s_aux1);      close(s_aux2);
}
```

Entre las acciones de importancia que se realizan dentro de esta función se pueden listar las siguientes:

- En esta función se inicializan todas las variables que son importantes para el buen funcionamiento del sistema.
- Se inicializan las estructuras utilizadas
- Se borra el archivo asociado a la tupla activa.
- Se cierra cada uno de los descriptores que fueron asociados a cada worker para la comunicación.
- Se cierran los sockets auxiliares para el manejo de archivos.
- Se envía una tupla de control llamada END a cada uno de los workers activos para que puedan terminar de forma normal la aplicación, sin necesidad de iniciarlo cada vez.

4.4 PRUEBAS

Durante la fase de pruebas del sistema MEMVIR, se realizaron diversas aplicaciones, en las cuales se mandaban tuplas al master (servidor) para verificar su correcta atención a cada solicitud, gracias a esta serie de pruebas se pudo corregir algunas situaciones no contempladas durante la fase de análisis y diseño, hasta llegar a lo que sería la fase final en la implementación de la versión 1.0 de la memoria virtual compartida con monitoreo MEMVIR.

Otro punto que también fue importante para llegar a esta fase fue el monitoreo, el cual al inicio del desarrollo no tenía un formato especifico o un diseño, pero que sirvió de base para implementar lo que sería el sistema de monitoreo con el que cuenta, en el cual se muestra la mayor información que pueda serle útil al usuario de esta memoria.

Finalmente se diseñó una aplicación cliente-servidor que proporciona el servicio de comunicación entre diferentes computadoras, para intercambio de mensajes, utilizando este servidor de memoria como puente de comunicación, y logrando resultados satisfactorios. Así los usuarios de las computadoras que utilicen esta aplicación podrán entrar en una sesión de comunicación (Chat) sin necesidad de saber que están haciendo uso de un modelo de memoria virtual compartida como lo es el espacio de tuplas, y mucho menos tendrán que preocuparse de cómo funciona.

CONCLUSIONES

Como resultados del presente trabajo se tiene que se creó un sistema de memoria virtual compartida MEMVIR, que funciona bajo el esquema cliente-servidor que ofrece dar atención a solicitudes de memoria que le lleguen de cualquier computadora cliente, que se encuentre conectada al servidor.

El objetivo general planteado que fue: d*iseñar e implementar un modelo de memoria virtual compartida el cual fuera adaptado en un modelo de máquina paralela basada en varias computadoras personales conectadas por una red LAN*, se logró ya que MEMVIR se implementó como una unidad de memoria virtual compartida (basada en el modelo espacio de tuplas) que atiende solicitudes de conexión y datos el cual fue adaptado en modelo de máquina paralela con arquitectura MIMD, en la cual varias computadoras se conectaron en red (LAN) para poder accesar la memoria.

El sistema de monitoreo que se planteó como un objetivo específico, también se logró ya que MEMVIR cuenta con un sistema de monitoreo que le permite al usuario poder tener conocimiento de cómo se está llevando a cabo el acceso a la memoria de tal forma que pueda ayudarle a poder corregir y depurar sus aplicaciones que hagan uso de esta memoria.

Igualmente se implementaron 2 funciones con las cuales se puede dar atención a solicitudes de datos y código (tuplas pasivas y tuplas activas) de forma correcta y eficaz.

Durante el diseño del sistema de memoria se plantearon e implementaron aportaciones importantes con respecto al modelo Linda, lográndose cada una de ellas, como son: contar con un servidor dinámico que todo el tiempo este disponible para atender solicitudes de memoria, y al mismo tiempo solicitudes de conexión lo cual hace escalable el sistema, por el otro lado también se puede en cualquier momento dar de baja a cualquier computadora que se encuentre en la red sin que eso ocasione que el servidor de memoria se tenga que reiniciar, se tiene un sistema tolerante a fallas sencillo ya que en varios casos el servidor puede corregir situaciones que le pueden causar problemas, cada vez que una nueva conexión es atendida en ese momento se considera para la asignación de trabajo, El servidor de memoria maneja una serie de tuplas de control que son muy útiles para su buen funcionamiento y mejor entendimiento en el sistema de monitoreo, el servidor puede ser suspendido pero antes de salir envía una tupla de control a todas las computadoras que se encuentren conectadas para evitar que se queden “colgadas” esperando a ser atendidas.

En resumen podemos decir que todos los objetivos que fueron planteados al inicio de este trabajo fueron alcanzados de manera satisfactoria.

APORTACIONES DEL PROYECTO

Dentro de las aportaciones que tiene el sistema MEMVIR con respecto al modelo Linda, tenemos lo siguiente:

- Se tiene un servidor dinámico (master), el cual todo el tiempo está en espera de solicitudes de memoria o de conexión.
- El servidor soporta que en cualquier momento, se pueda dar de baja a alguna de las computadoras que se haya conectado, sin afectar su funcionamiento, es decir simplemente se actualizan las tablas correspondientes par que ya no sea considerada dentro del sistema.
- El servidor integra inmediatamente al sistema a cualquier computadora que solicite conexión aún cuando ya este atendiendo la ejecución de alguna aplicación, asignándole trabajo, si existiera disponible.
- El servidor todo el tiempo está disponible para atender solicitudes de memoria sin tener que reiniciarse.
- Si el desarrollador suspende su ejecución, el servidor puede ser interrumpido, pero antes de terminar se reinicia el servidor y envía a todos los workers la tupla de control END para evitar que se queden “colgados”, y puedan terminar de forma normal.
- El sistema cuenta con un sistema de monitoreo, con la información más importante que pueda servir para que los usuarios tengan conocimiento de que información es depositada, por ejemplo que tipo de tuplas llegan, que máquinas están conectadas solicitando trabajo, quienes están disponibles, quienes están ocupadas, quienes se desconectaron, cuántas tuplas activas llegaron, cuantas se depositaron en el espacio de tuplas, y cuántas tuplas pasivas se atendieron.
- Se utilizan tuplas de control que hacen más claro el funcionamiento del sistema.
- Una vez que el servidor recibe una tupla de control ENDA se reinicia para evitar que se vaya acumulando basura en el espacio de tuplas.

El servidor presenta cierta tolerancia a fallas ya que si el desarrollador se desconecta esto no afectará el funcionamiento del servidor ni el de los workers de manera negativa.

PERSPECTIVAS

Dentro de las perspectivas que se consideran para el proyecto presentado, se clasifican en posibles modificaciones y trabajos a futuro, las cuales se listan a continuación.

1. Una modificación que se le puede realizar al proyecto es cambiar el comportamiento del servidor que se tiene. Como se mencionó se trata de un servidor interactivo, en el cual las solicitudes son atendidas por el mismo servidor, a un servidor basado en hilos, esto debido a que la creación de hilos no es tan costosa y varios hilos de un solo proceso si pueden compartir datos entre ellos ya que comparten el mismo espacio de direccionamiento, esta modificación se propone para evitar el inconveniente que se tiene ya que si llega una solicitud que tarde un tiempo excesivo, se dejará sin servicio a las demás conexiones durante ese tiempo por que el servidor no es concurrente.

2. Otra modificación que se le puede realizar es cambiar la función *select( )* que se utiliza para consultar los descriptores de sockets asociados, por la función *epoll( ), la cual e*s más reciente, y su principal bondad con respecto a select radica en su complejidad O(1) en comparación con select que tiene una complejidad O(n).

3. Se le puede agregar un módulo sencillo de tolerancia a fallas en donde el master sea un servidor más estable ante la presencia de eventos no contemplados, como son la llegada de tuplas “basura” es decir con un tipo desconocido, el que un worker no responda para dar resultados a la tarea asignada, etc.

4. Un trabajo que se puede continuar a partir del presente, es cambiar el esquema de memoria centralizada, que se utiliza a una memoria distribuida, es decir en lugar de que se tenga un solo espacio de tuplas, se disponga de múltiples espacios de tuplas.

5. Que se le pueda implementar un módulo de tolerancia a fallas bastante completo que proporcione un servidor altamente eficiente.

6. Se puede implementar un servidor que pueda dar atención a mas de un usuario que desee utilizar el espacio de tuplas para ejecutar sus aplicaciones.

7. Se puede cambiar el lenguaje que se utilizó para la implementación del servidor.
8. Se puede cambiar de plataforma (sistema operativo) sobre el cual se implemente el servidor.

9. Se puede ampliar el conjunto de instrucciones con las que cuenta el servidor (instrucciones Linda).

10. Se puede mejorar el sistema de monitoreo con respecto a la presentación de las ventanas.

11. Se puede tener un monitor más completo.

12. Se puede agregar un módulo de seguridad, que se utilice para evitar que algún usuario no autorizado se conecte y pueda extraer información.

APENDICE A

SERVIDOR MASTER

Se desarrolló un programa denominado ***master*** el cual una vez instalado en una computadora personal que forme parte de una arquitectura paralela de tipo MIMD de memoria compartida, sea vista por los usuarios como una unidad de memoria en la cual pueden depositar sus datos e incluso programas para ser ejecutados de forma paralela.

INSTALACION DEL MASTER

Para poder hacer uso de este software se debe compilar el programa denominado master.c utilizando el gcc de LINUX Mandrake 7.2 o 9.2. de la siguiente manera:

gcc master.c –o master –lnsl –lpthread –lnsl –lncurses

En la línea anterior se indica en primer lugar el nombre del programa que se va a compilar que en este caso es ***master.c,*** enseguida se especifica ***–o master*** para indicar que se genere un ejecutable de nombre master, se le agrega ***–lnsl*** y ***-lpthread*** debido a que se utilizaron sockets e hilos en la implementación y por último se le agrega ***–lncurses*** debido a que para la implementación del monitor se utilizaron funciones de pantalla que se encuentran en la librería ncurses.

Una vez que se tiene el ejecutable master solo hay que ejecutarlo de la siguiente manera:

./master

Al ejecutar la orden anterior se comienza por crear los primeros sockets necesarios para la comunicación, en particular el socket por el cual se va a aceptar solicitudes de conexión hacia la memoria, de manera predeterminada se estableció el puerto 8080 para llevar a cabo esta comunicación.

Una vez que el socket principal ha sido creado se inicia el monitor y la ejecución del servidor de memoria. En la pantalla del monitor se visualizará la siguiente información:

1. IP de la computadora del usuario que desea hacer uso de la memoria para almacenar código de programas que desea sean procesados de manera paralela. (DESARROLLADOR).
2. IP de cada una de las computadoras que se conectan para solicitar tuplas ya sea pasivas o activas, además del estado en que se encuentran, es decir ocupadas, bloqueadas o listas.

3. socket por el cual se están atendiendo sus solicitudes.
4. Tuplas pasivas que llegan a la memoria.
5. Tuplas activas que llegan y cuantas se van asignando.

Cuando se termine de atender al usuario de la memoria se recibirá una tupla de control END que se visualiza en el monitor y en seguida se informa que el servidor se va a reiniciar para atender a más usuarios.

Para dar por terminado el servidor se debe presionar CTRL-C la cual es una señal que se ha capturado para que antes de que finalice la aplicación se cierren todos los sockets utilizados, se limpie el espacio de tuplas y se envíen tuplas de control a los usuarios de la memoria para que terminen de manera normal.

APENDICE B

APLICACION PARA EL INTERCAMBIO DE MENSAJES ENTRE USUARIOS (CHAT)

Después de múltiples pruebas que se realizaron con el servidor de memoria, se desarrolló una aplicación que hiciera uso de la memoria, en este caso para llevar a cabo la comunicación entre usuarios que se conectan al servidor para el intercambio de mensajes.

INSTALACION DE LA APLICACIÓN CHAT

Para poder hacer uso de esta aplicación se debe compilar el programa denominado chat.c utilizando el gcc de LINUX Mandrake 7.2 o 9.2, de la siguiente manera:

gcc chat.c –o chat –lnsl –lncurses

En la línea anterior se indica en primer lugar el nombre del programa que se va a compilar que en este caso es ***chat.c,*** enseguida se especifica ***–o chat*** para indicar que se genere un ejecutable de nombre chat, se le agrega ***–lnsl*** debido a que se utilizaron sockets en la implementación para realizar la comunicación y por último se le agrega ***–lncurses*** debido a que para la implementación del chat se utilizaron funciones de pantalla que se encuentran en la librería ncurses.

Una vez que se tiene el ejecutable chat solo hay que ejecutarlo de la siguiente manera:

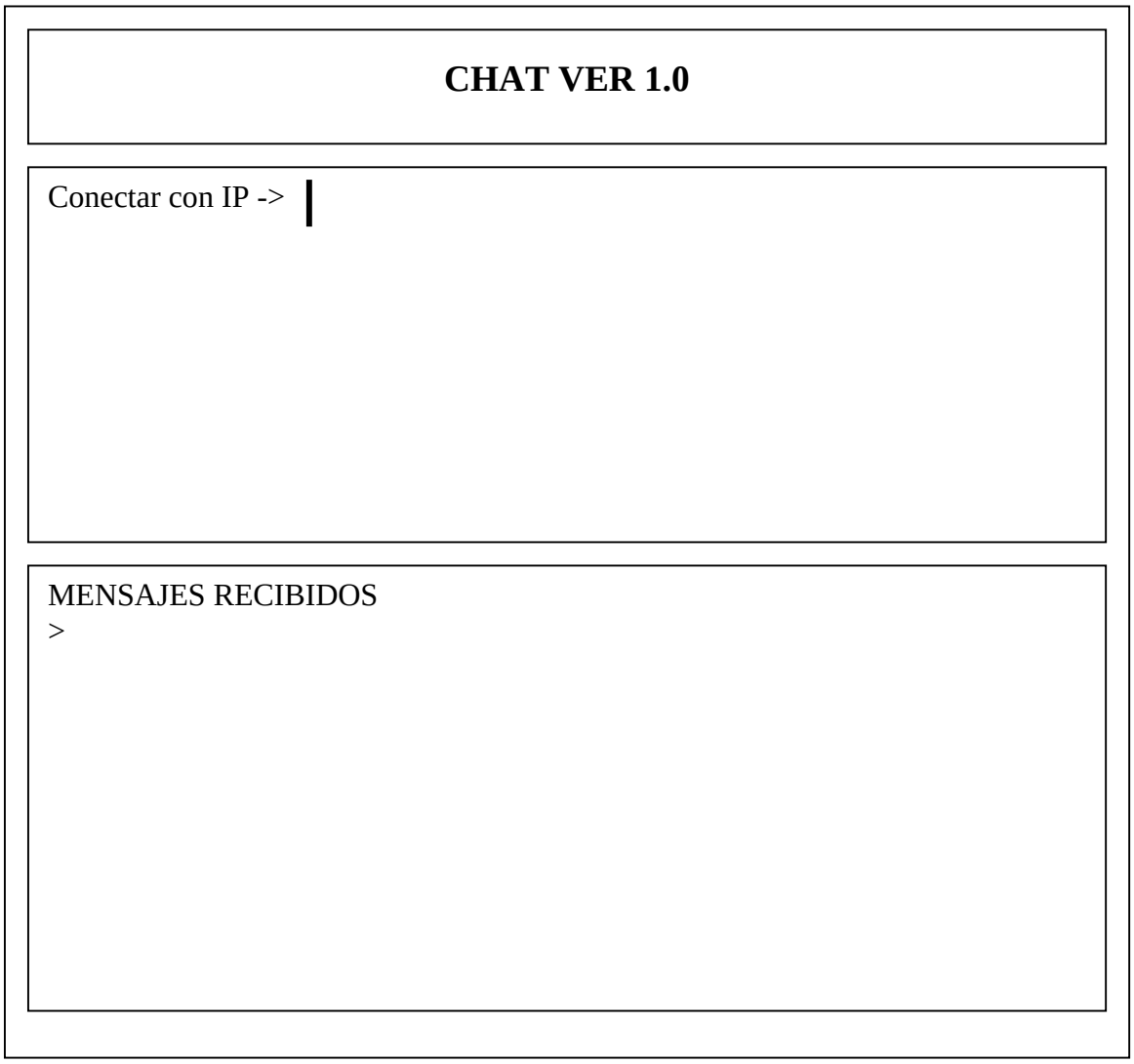
./chat ipmaster num_puerto

En donde en ***ipmaster*** se debe especificar la IP de la computadora en donde se instaló el programa master (memoria). En ***numpuerto*** se debe especificar el puerto por el cual se va a realizar la comunicación con el master en este caso debe ser el puerto 8080.

Al ejecutar la orden anterior se despliegan tres ventanas por medio de las cuales se lleva acabo la transmisión de mensajes de la siguiente manera:

- La primera ventana muestra el nombre de la aplicación y la versión.
- La Segunda ventana es destinada para que el usuario pueda escribir en primer lugar la IP de la computadora con la que se desea establecer comunicación, y una vez establecida la conexión se utiliza para escribir los mensajes que se desean enviar.
- La tercera ventana se utiliza para visualizar los mensajes que están llegando desde otra máquina.

Gráficamente se tiene la siguiente distribución de ventanas en la aplicación CHAT.



Esta es la primera ventana de la aplicación que se visualizará en la cual se debe introducir la IP de la computadora con la cual se desea establecer comunicación. Una vez que se introduce la IP, cambiará el mensaje por el siguiente: “MENSAJES A ENVIAR ESCRIBA , O FIN PARA TERMINAR”, lo que indica que ya se pueden teclear los mensajes que se desean enviar, y pulsar <<enter>> para que se envíen a la memoria (master), aquí no es necesario que la computadora a la cual se enviaran los mensajes este conectada antes a la memoria, esto debido a que los mensajes se envían en forma de tuplas (OUT) a la memoria (espacio de tuplas) y permanecen ahí , de tal forma que en el momento que se establezca la conexión con la máquina destino, en esta se visualizaran todos sus mensajes que se encuentren en el espacio de tuplas.

Para terminar la sesión se debe escribir como mensaje END y al enviarse se cerrará la comunicación y terminará la aplicación.

Cuando se termine de atender al usuario de la memoria y se reciba la tupla de control END el master (memoria) se reiniciará para atender a más usuarios.

Para dar por terminado el servidor se debe presionar CTRL-C.

En esta aplicación se hace uso de las tuplas siguientes:

- OUT para enviar mensajes
- INP, RDP, IN Y RD para recibir mensajes.

BIBLIOGRAFIA

[1] COMPUTO PARALELO
Dr. René Cumplido
Dra. Claudia Feregrino
Ian Foster
Septiembre 2004

[2] Designing and Building Parallel Programs 1995
http://www.mcs.anl.gov/dbpp/text/book.html
Septiembre 2004

[3] PROCESAMIENTO PARALELO
Vìctor Perales Fabian vperalesf@hotmail.com
http://www.monografias.com/trabajos16/arquitectura-paralela/arquitectura-paralela.shtml
Septiembre 2004

[4] ARQUITECTURAS PARALELAS
http://strix.ciens.ucv.ve/~matcomp/algpar/CLASE1.htm
Septiembre 2004

[5] ARQUITECTURAS DE COMPUTADORAS
CICESE Febrero 2002
http://telematica.cicese.mx/computo/super/cicese2000/paralelo/Part3.html
Septiembre 2004

[6] SISTEMA DIDATICO PARA EL PROCESAMIENTO PARALELO
R. V. Gil
FCC BUAP.2002
Marzo 2004

[7] MODELOS Y LENGUAJES DE CORDINACIÓN
http://www.lcc.uma.es/~mdr/ProyInvestigador.html#MODELOS%20Y%20LENGUAJES%20DE%20COORDINACI
Marzo 2004

[8] TUPLETS: WORDS FOR A TUPLE SPACE MACHINE
http://www.cs.york.ac.uk/linda/pubs.html
Marzo 2004
[9] Linda Introduction
Blaise Barney
Maui High Performance Computing Center
Marzo 20, 1995
http://aixport.sut.ac.jp/advsys/mhppc/linda/linda.html

[10]Ray Tracing with Network Linda
Robert Bjornson, Craig Kolb & Andrew Sherman
Siam News
January, 1991

[11] La memoria
http://www.mailxmail.com/curso/informatica/arquitecturaordenadores/capitulo3.htm
Octubre 2004

[12] Tipos de Memoria- Jerarquía de memoria
http://www.mailxmail.com/curso/informatica/arquitecturaordenadores/capitulo5.htm
Octubre 2004

[13] Administración De La Memoria
http://www.tau.org.ar/base/lara.pue.udlap.mx/sistoper/capitulo4.html
Octubre 2004

[14] MEMORIA VIRTUAL
http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/MonogSO/MEMVIR02.htm
Octubre 2004

[15] Gestión de memoria
David A Rusling
http://gemini.udistrital.edu.co/comunidad/estudiantes/dguerrero/mm/memory-es.html
Octubre 2004

[16] Linda
http://www.orcero.org/irbis/disertacion/node257.html
Agosto 2004

[17] DISEÑO DE LA JERARQUÍA DE MEMORIA.
http://dac.escet.urjc.es/lrincon/uned/etc3/Etc3-08.PDF
Octubre 2004

Proyecto Cherokee: Diseño, implementación y aspectos de rendimiento de servidores web
Alvaro López Ortega <alvaro@gnu.org>
http://www.consol.org.mx/2004/material/123/Proyecto_Cherokee_-_CONSOL_2004.pdf
Julio 2004

ADVANCED COMPUTER ARCHITECTURE:
Parallelims, Scalability, Programability
Hwang Kai
International editions 1993
Mc Graw Hill

Conceptos de cómputo paralelo
Torres Jiménez José, Rodríguez Tello Eduardo Arturo
Preedición Mayo 2000
Trillas

El lenguaje de programación C
Kernighan Brian W. & Ritchie Dennis M.
Primera edición Abril 1985
Prentice Hall

Interprocess Communications in UNIX:
The NOCKS and Crannies
Shapley Gray John
1998 2nd ed
Prentice Hall