Universidad de Buenos Aires
Fac. de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

Integrantes		**Director**
Gabriela Fernanda Rizzo	LU 1320/86	Claudio Righetti
María Eugenia Fernández Fariña	LU 262/86	

Un ambiente de desarrollo y ejecución en redes LAN TCP/IP con equipos Unix para programación paralela-distribuida implementando Linda-C

Agosto de 1996

Un ambiente de desarrollo y ejecución en redes LAN TCP/IP con equipos Unix para programación paralela-distribuida implementando Linda-C

María Eugenia Fernández Fariña y Gabriela Fernanda Rizzo
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Agosto de 1996

Resumen

Describimos Linda - el lenguaje de coordinación propuesto por D. H. Gelernter. Linda provee un conjunto de primitivas que son incorporadas a un lenguaje de programación para convertirlo en un lenguaje paralelo-distribuido. Describimos una implementación de Linda incorporado al lenguaje C. Esta implementación incluye la primitiva de Linda para creación de procesos. Definimos un ambiente de desarrollo y ejecución para computadoras con sistema Unix de arquitecturas heterogéneas conectadas en red local mediante protocolo TCP/IP. El ambiente de desarrollo que definimos brinda la estructura suficiente para que el programador Linda se independice de las plataformas de hardware en las que trabaja. El ambiente de ejecución garantiza al programador Linda la no interferencia de los datos manipulados por los procesos que cooperan entre sí, con los datos de procesos de otros programadores Linda. Proveemos herramientas para el desarrollo y la ejecución de programas Linda.

Abstract

We describe Linda - the coordination language proposed by D. H. Gelernter. Linda provides a set of primitives that is added to a programming language in order to provide paralel and distributed capabilities. We describe an implementation of Linda embeded in the C programming language, including the Linda primitive for process creation. We define a development and execution environment for heterogeneous hardware computers running Unix on a TCP/IP conected local area network. The development environment provides the infrastructure that allows the programmer to work in a set of machines with different hardware in a transparent way. The execution environment guarantees the no-interference among the data belonging to processes of several programmers. We provide tools for the development and execution environments.

Agradecimientos

A Leonardo Balbiani, coautor de las primeras ideas de esta implementación, por introducirnos en este tema, por el material prestado y por sus valiosos comentarios.

A Claudio Righetti, nuestro director, por todo lo que hizo por este trabajo.

A Michael y a Ricardo, por el impulso para terminar este trabajo, por su comprensión y apoyo.

A Hewllet Packard Argentina, Banco de Galicia y Andersen Consulting por facilitarnos los equipos para el desarrollo de esta tesis.

Tabla de Contenidos

Tabla de Figuras

Apéndices

1. Introducción

1.1 Conceptos de Linda

Linda es un lenguaje de coordinación propuesto por D. H. Gelernter [GEL/85], que provee un conjunto de primitivas para ser agregadas a un lenguaje tradicional de programación - llamado **lenguaje host** - transformando al mismo en un lenguaje de programación paralela-distribuida. *Linda* está basado en la idea de **Comunicación Generativa** la cual unifica las nociones de creación de procesos, comunicación y sincronización.

Linda no constituye un lenguaje de programación en sí mismo. De hecho, las primitivas de *Linda* no son un medio para resolver tareas básicas de un lenguaje como loops, operaciones aritméticas, Entrada/Salida, etc. Estas tareas son delegadas al lenguaje *host* al que se incorporan las primitivas, quedando a cargo de las primitivas de *Linda* las operaciones para implementar la *Comunicación Generativa*.

La intercomunicación entre procesos y la distribución de los datos son provistas a través de una memoria asociativa lógica, compartida por estos procesos, llamada **Espacio de Tuplas**. Los procesos no interactuan entre sí directamente, sino a través del *Espacio de Tuplas*, valiéndose del conjunto de primitivas de *Linda*. Esta característica difiere de la comunicación entre procesos que se da en los métodos tradicionales, como pasaje de mensajes, o uso de variables compartidas.

Los elementos almacenados en el *Espacio de Tuplas*, llamados **tuplas** , son un conjunto ordenado de campos, por ejemplo:

("P", 1 ,2 , 3 , "hola")
("fila" , 1 , 20 , 3.5 , 40 , 21)

La unidad de memoria en esta memoria asociativa compartida no es el byte, ni el campo, sino la *tupla.* Es decir que en toda operación realizada contra el *Espacio de Tuplas* a través de las primitivas de *Linda*, ya sea para leer, poner o sacar datos, se utilizan *tuplas*. Un proceso que

genera datos, en forma de *tuplas*, las coloca en el *pool* de *tuplas* que constituye el *Espacio de Tuplas*. Cualquier proceso que desee acceder a la *tupla* puede sacarla del *Espacio de Tuplas*, o leerla sin sacarla del mismo, de manera que otro proceso también pueda accederla. En el *Espacio de Tuplas*, los elementos son accedidos por contenido, para lo cual se necesita un mecanismo de **unificación** o **pattern matching** que se explicará más adelante.

Cada campo de una *tupla* pertenece a un tipo de datos determinado, y puede ser **formal** o **actual**. Los *campos actuales* son los que contienen datos, y los *campos formales* son los que tienen una dirección para almacenar un dato del tipo al que pertenecen. Por ejemplo,

("tupla", 3 , 658.73 , &var_ent, "linda" ,&var_real)

es una tupla cuyos *campos actuales* son la cadena de caracteres *"tupla"*, el entero *3*, el número real *658.73* y la cadena de caracteres *"linda"*, y cuyos *campos formales* son las variables *var_ent* y *var_real*. La forma de indicar que un campo es *formal* es anteponiendo al nombre de la variable un signo especial, en nuestro ejemplo el signo &.

Las operaciones atómicas provistas por *Linda* para interactuar con el *Espacio de Tuplas* son las primitivas **in**, **inp**, **out**, **read**, **readp** y **eval**.

La primitiva *out* permite agregar una *tupla* en el *Espacio de Tuplas*. Esta operación se realiza en forma asincrónica, es decir, un proceso que invoca un *out* continúa su ejecución inmediatamente después de la invocación.

Las primitivas *in* y *read* extraen información del *Espacio de Tuplas*. Un **template** es una *tupla* que cuenta generalmente con uno o más *campos formales*, que se utiliza como argumento en las primitivas *in* y *read* para buscar una *tupla* que *unifique* con el mismo según las *reglas de unificación de tuplas* de *Linda*, explicadas más adelante. Cuando un *template* que contiene *campos formales* unifica con una *tupla* del *Espacio de Tuplas*, los *campos formales* reciben el valor del *campo actual* correspondiente, de la *tupla* con la que unificó.

Las operaciones *in* y *read* son sincrónicas. Los procesos se bloquean y el control es devuelto al proceso en ejecución recién cuando se encuentra una *tupla* que unifica con el *template*. La diferencia entre el *read* y el *in* es que después de un *read* la *tupla* que unificó con el *template* permanece en el *Espacio de Tuplas*, mientras que con el *in* es borrada del mismo.

Notar que ninguna de estas primitivas permite modificar una *tupla* del *Espacio de Tuplas*. Esta operación debe simularla el programador extrayendo la *tupla* con un *in*, modificándola en su programa y poniéndola nuevamente en el *Espacio de Tuplas* utilizando la primitiva *out*.

En la *Figura 1* se ilustra el efecto de las primitivas *in, out y read* sobre el Espacio de Tuplas.

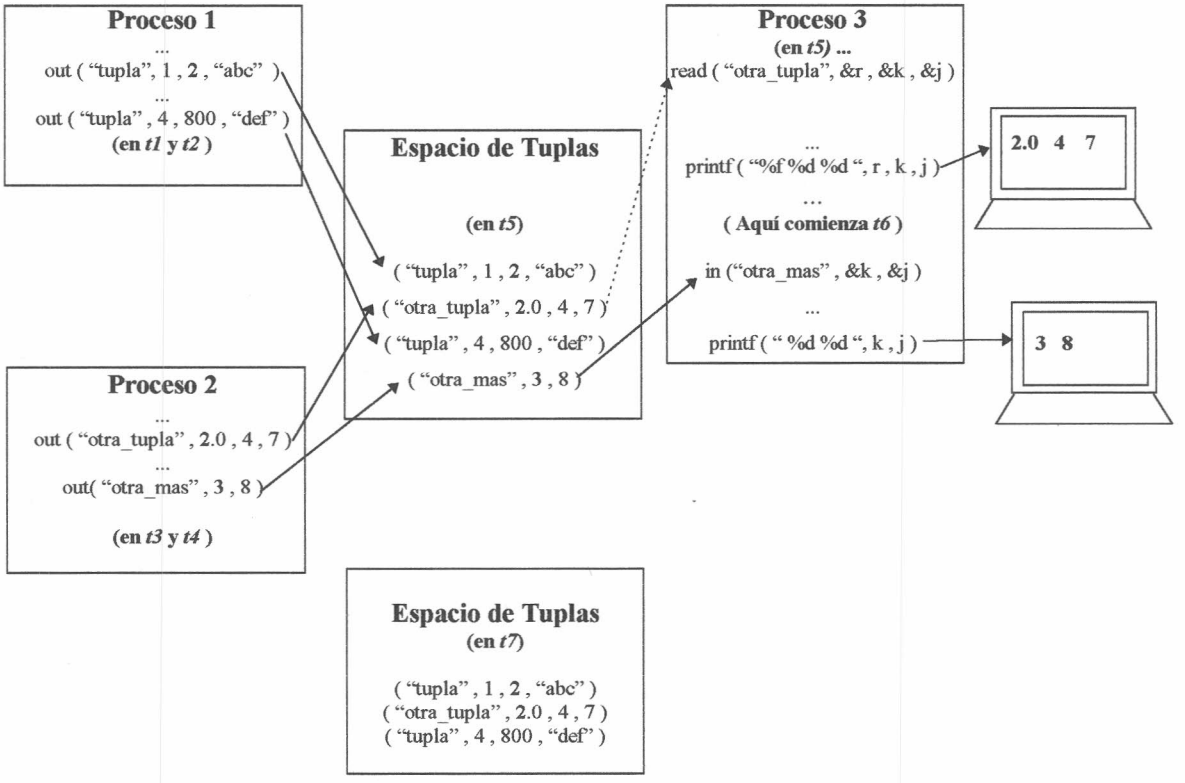

Figura 1: Efecto de las primitivas in, out y read sobre el Espacio de Tuplas

De funcionamiento similar a las primitivas *in* y *read*, son *inp* y *readp*. La variante que introducen es el no bloqueo del proceso ante la ausencia de una *tupla* que unifique con el *template* argumento de estas operaciones. En otras palabras, estas operaciones son asincrónicas.

Las **reglas de unificación** de *Linda* [CAR/87] estipulan que una *tupla* y un *template* unifican si:

a) tienen la misma cantidad de campos;
b) los campos correspondientes entre sí, es decir, los que ocupan la misma posición, tienen el mismo tipo de datos;
c) teniendo la *tupla* y el *template campos actuales* en la misma posición, los valores de estos campos son iguales; y
d) los campos correspondientes entre sí no son ambos *campos formales.*

Ejemplos:

Introducimos - para los ejemplos de *unificación de tuplas* - la nomenclatura que usaría un programador trabajando con nuestra implementación de *Linda*-C (*Linda* sobre C como *lenguaje host*).

La *tupla* ("fila" , "iiiii" , 1 , 3 , 67 , 78 , 2)

no unifica con el *template* ("fila" , "iiii" , 1 , 3 , 67 , 2) por a)

no unifica con el *template* ("fila" , "iiifi" , 1 , 3 , 67 , 78.0 , 2) por b)

no unifica con el *template* ("fila" , "iiiii" , 1 , 3 , 67 , 50 , 2) por c)

si unifica con el *template* ("fila" , "i&i&iii" , 1 , &a , &b , 78 , 2)

La *tupla* ("fila" , "iiiii" , 1 , 3 , 67 , 78 , &a)

no unifica con el *template* ("fila" , "iiii&i" , 1 , 3 , 67 , 78 , &b) por d)

Si en el momento en que un proceso ejecuta un *in* o un *read* no existe una *tupla* en el *Espacio de Tuplas* que unifique con el *template*, el proceso queda bloqueado. Cuando algún otro proceso invoque a la primitiva *out,* poniendo en el *Espacio de Tuplas* una *tupla* tal que unifique con el *template* del proceso bloqueado, la *tupla* será entregada al proceso bloqueado. Desde el punto de vista del proceso, el efecto será que dejará de estar bloqueado y los *campos formales* del *template* estarán instanciados con los valores correspondientes en la *tupla*. Según la primitiva ejecutada haya sido un *read* o un *in,* la *tupla* quedará en el *Espacio de Tuplas* o dejará de pertenecer al mismo respectivamente.

Si en cambio, en el momento en que un proceso ejecuta un *in* o un *read* hay más de una *tupla* que pueda unificar con el *template* argumento del *in* o *read*, alguna de éstas será "devuelta" al proceso (los *campos formales* del *template* serán instanciados con los *campos actuales* de la *tupla)*. El programador no puede realizar ninguna suposición sobre cuál será la *tupla* devuelta.

Si hay varios procesos bloqueados por las primitivas *read* o *in*, y se ejecuta un *out* de una *tupla* que unifica con los *templates* por los que estos procesos esperan, se satisfacerá a un subconjunto - que podría ser vacío - de los procesos bloqueados por *reads* y a solo uno de los procesos bloqueados por *in.*

La primitiva *eval* también genera una *tupla* en el *Espacio de Tuplas*, pero los argumentos del *eval* son evaluados por un nuevo proceso creado para tal fin. El *eval* es entonces la primitiva utilizada para la generación de nuevos procesos. El *eval* difiere con el *out* en la manera en que sus argumentos son evaluados. Los argumentos de un *out* se evaluan de la misma manera que los argumentos de cualquier subrutina. Por ejemplo, para ejecutar

out (f(x))

se evalúa en primer lugar *f(x)* y luego, llamando *a* al resultado de *f(x),* se ejecuta

out (a)

En cambio, los argumentos del *eval* son evaluados por un nuevo proceso. En el ejemplo,

eval (f(x))

informa al sistema que una *tupla* debe ser creada para contener el valor de *f(x)*, y que un proceso debe ser creado para evaluarlo. Hay muchas consideraciones y preguntas sin respuesta con

respecto al funcionamiento del *eval* y su implementación. Éstas se relacionan en su mayoría, con el momento y la forma en que los argumentos de la función *f* del ejemplo son evaluados, y con la creación dinámica de procesos. Volveremos sobre el tema cuando expliquemos la semántica propuesta y la implementación del *eval* realizada en este trabajo.

1.2 Características de Linda

Las principales características de *Linda* surgen como consecuencia de los atributos del *Espacio de Tuplas*.

Comunicación anónima

Los procesos no intercambian datos entre sí sino a través del *Espacio de Tuplas*, por lo tanto, el método de comunicación es anónimo, interactúan sin necesidad de conocer sus nombres, o saber de la existencia del otro. Emisor y receptor simplemente deben ponerse de acuerdo en el formato de los datos, ya que la comunicación se lleva a cabo escribiendo o leyendo del *Espacio de Tuplas*.

Buffering

El *Espacio de Tuplas* provee también *buffering*, de manera que el proceso que genera una *tupla* no necesita sincronizarse con el proceso que la lee. Este *desacople en el tiempo*[1] (*"temporal decoupling"*) permite que los procesos puedan ejecutarse a su propia velocidad, sin tener que preocuparse por la carga o velocidad de otros procesos. Sin embargo, hay diferencias entre el modelo de *Comunicación Generativa* de *Linda* y el modelo de los sistemas de *Buffering*. Esta diferencia radica básicamente en el tiempo de vida de una tupla, que es mayor que el de un mensaje en un *buffer*. Los mensajes en *buffers* "viven" hasta que puedan ser entregados, mientras que una *tupla* puede ser consultada con un *read* y seguir estando en el *Espacio de Tuplas*, o "madurar" si es insertada con un *eval*. Otra diferencia con los sistemas de *Buffering* radica en la forma de acceder a los datos: en *Linda* se realiza utilizando una semántica de memoria asociativa.

[1] Se llama "desacople en el tiempo" cuando dos o más procesos se comunican a través de un recurso en común, y no es necesario que estos procesos estén activos simultáneamente para llevar a cabo dicha comunicación.

Modelo no distribuido

Aunque se implemente en forma distribuida, *Linda* responde a un modelo lógicamente no distribuido, debido al modelo de datos compartidos en el *Espacio de Tuplas*: éste es tratado como una memoria compartida donde todos los procesos dejan y toman datos (en forma de *tuplas*), independientemente de la implementación física del mismo. Sin embargo, la diferencia entre un sistema de memoria compartida y el *Espacio de Tuplas* radica en el acceso por contenido de este último.

Sincronización de procesos

Como las primitivas de *Linda in* y *read* son sincrónicas, el *Espacio de Tuplas* puede usarse como coordinador de procesos. Los procesos se sincronizan bloqueándose al esperar que las *tuplas* estén disponibles en el *Espacio de Tuplas*.

Independencia de la topología de la red

Linda libra al programador de consideraciones acerca de la topología de la red en la que está trabajando. El programador puede pensar sus programas de la misma forma en que lo haría si las operaciones con el *Espacio de Tuplas* fueran operaciones realizadas directamente con la memoria. Es la implementación de *Linda*, y no el programador, quien se encarga del manejo de la verdadera distribución física de los datos y de los nuevos procesos generados a partir de un *eval*. Esto permitiría que un programa escrito en *Linda* pueda ser portado a cualquier otro sistema que soporte *Linda*, independientemente de la topología existente, pero a su vez, impide al programador tener control directo de todos los recursos del sistema. Por esta causa, este modelo no es apropiado para aplicaciones en las que el programador necesite sacar provecho de su conocimiento de la topología y de la capacidad de los equipos, eligiendo la distribución del procesamiento.

No determinismo

Cuando un proceso está bloqueado en un *read* o *in*, no se puede determinar qué proceso le proveerá la información que está esperando, ya que la comunicación es anónima. Tampoco se

puede saber qué proceso enviará al *Espacio de Tuplas* la información que otro proceso está esperando. Este comportamiento es no determinístico.

Aplicaciones típicas

Las aplicaciones típicas en *Linda* siguen el modelo **master-worker**, de maestro y trabajadores replicados. En este modelo existe un proceso maestro encargado de generar los datos necesarios para ser tomados por los procesos trabajadores. Estos datos consisten de información e instrucciones para que el proceso trabajador que los toma realice una determinada tarea. El maestro deja estos datos en el *Espacio de Tuplas*, donde a su vez los trabajadores dejarán las tuplas con sus resultados.

La ventaja de la utilización de *Linda* para este modelo, es que el trabajo se distribuye automáticamente entre los trabajadores, y que si alguno de éstos terminara antes que los otros, puede solicitar más trabajo.

Este modelo se puede implementar independientemente de la cantidad de nodos existentes en la red, inclusive con un único nodo.

2. Antecedentes

En [CAR/87] y [ARA/89] se describen y analizan las primeras implementaciones de *Linda*. En ambos, la primitiva *eval* es mencionada, pero no se detalla ninguna posible implementación en profundidad. Los trabajos se limitan a plantear dudas sobre su funcionamiento teórico [CAR/87], o directamente la ignoran [ARA/89]. En ningún caso hay definiciones semánticas de la misma.

En [BOB/90] se implementan las primitivas *in*, *read* y *out* utilizando como lenguaje host al lenguaje C. La plataforma desarrollada es estática en el sentido que el código está preparado para funcionar con un número fijo de nodos conectados en red local, y utiliza las APIs de Ethernet para poner los mensajes en la red. El *Espacio de Tuplas* desarrollado distribuye las *tuplas* según una

función de hash entre ambos nodos. El *eval* es analizado teóricamente en una amplia gama de posibilidades y se plantean diversas formas para su posible funcionamiento.

En [BAL/94] se propone una implementación de un kernel *Linda*. El trabajo incluye una disertación teórica sobre las distintas formas de programar las primitivas, plantea diversas formas de enviar los mensajes conteniendo las *tuplas* o los *templates* y analiza distinas posibilidades de la semántica del *eval*.

En ninguno de estos trabajos se presenta cuál es la idea de los autores sobre cómo debería ser un ambiente distribuido de trabajo donde se utilice *Linda*. No se maneja el concepto de "usuario de *Linda*", como aquel que tiene un ambiente definido tal que le permita desarrollar y ejecutar programas *Linda*. Tampoco queda claro cuales serían las consecuencias si varios programadores trabajaran con un único *Espacio de Tuplas* , ejecutando programas que no hayan sido diseñados para cooperar entre sí. Ninguno de los trabajos contempla cómo resolver el problema de pertenencia a uno u otro conjunto de procesos, para brindar de esta forma un soporte multiusuario.

3. Motivación

Linda no es un lenguaje de programación en sí mismo, sino que provee primitivas para ser incorporadas a otro lenguaje. Estas primitivas, una vez disponibles convierten al lenguaje elegido en un lenguaje con el que es posible escribir programas paralelos-distribuidos, sin la necesidad de programar el control del paralelismo o de la distribución de procesos y datos, como se haría, por ejemplo, si se utilizara en forma cruda memoria compartida, semáforos, u otros métodos de intercomunicación y generación de procesos.

Esta característica nos impulsó a realizar una implementación de estas primitivas, dada la ventaja que significa para un programador afianzado en un lenguaje, tener la posibilidad de escribir programas paralelos-distribuidos con el sólo requerimiento de aprender una mínima cantidad de funciones adicionales. Esta ventaja es aún mayor dado que estas funciones cuentan, además, con

una sintaxis simple y concreta, a la vez que pueden manejar distintos tipos de datos, combinando éstos en sus argumentos a gusto del programador.

A diferencia de las implementaciones de trabajos anteriores, citados en los "Antecedentes", proponemos una implementación de *Linda* sobre redes locales, - que son un soporte natural para sistemas distribuidos - que cuenta con una **infraestructura** que la soporta y brinda al programador un ambiente de desarrollo y ejecución, necesarios para que la implementación no se limite a un prototipo. También definimos una posible implementación de la primitiva para generación distribuida de procesos de *Linda*, cuyos aspectos semánticos y de implementación no estaban totalmente definidos por los autores de *Linda*.

4. Implementación de Linda-C

4.1 Decisiones generales de implementación

- El lenguaje elegido para incorporar las primitivas de Linda en nuestra implementación es el lenguaje C. Esta es la elección natural dada la plataforma de sistema operativo en la que trabajaríamos, ya que en general, todas las instalaciones de Unix incluyen el compilador de lenguaje C. También contribuye a esta elección el hecho que las librerías para programar las comunicaciones utilizan sintaxis y semántica de lenguaje C, y que las llamadas al sistema operativo (system calls) se realizan como simples llamadas a funciones del lenguaje.
- Para la comunicación entre nodos de la red local se eligió la familia de protocolos TCP/IP debido a que es un estandard de facto en las comunicaciones entre equipos Unix, y a que los fuentes de los protocolos están disponibles, en caso de necesitarlos.
- Para la comunicación entre procesos se eligieron las librerías de Berkeley Sockets (BSD Sockets), que utilizan la familia de protocolos TCP/IP y que proveen los mecanismos para el envío de mensajes entre procesos.
- Los tipos de datos soportados para los campos de las *tuplas* son: entero (int), real (float) y cadena de caracteres (char *).
- El primer campo de una *tupla* debe ser del tipo cadena de caracteres, y es interpretado como el nombre de la *tupla*.

- El segundo campo de una *tupla* deber ser del tipo cadena de caracteres, representa la signatura de la *tupla*, y estará formado por una letra (**i**, **f**, o **s**) por cada campo de la *tupla* según su tipo sea entero, real o cadena de caracteres respectivamente, anteponiendo el signo **&** cuando se trate de un campo formal.

4.2 Diseño del Espacio de Tuplas

Nuestro diseño del *Espacio de Tuplas* fue realizado para aumentar la eficiencia de búsquedas de las tuplas. Por tratarse de una estructura compleja, no beneficia la inserción de tuplas en el *Espacio de Tuplas*. Dependiendo del tipo de aplicación, esta decisión puede ser acertada o no. Sin embargo, dado que con la *tupla* generada por una operación *out* se puede satisfacer a varias operaciones *read* y a una operación *in*, en las aplicaciones típicas la cantidad de operaciones *read* o *in* es igual o mayor a la cantidad de operaciones *out*. Por esto consideramos que la elección más inteligente es beneficiar a las búsquedas más que a las inserciones. Otra razón para beneficiar las búsquedas en el *Espacio de Tuplas* es que indefectiblemente una operación *read* o *in* generará comparaciones del *template* con las *tuplas del Espacio de Tuplas,* por lo que creemos conveniente tratar de mejorar la eficiencia de estas dos operaciones, tratando de reducir la cantidad de comparaciones necesarias para encontrar una *tupla* que unifique con el *template*.

La *Figura 2* muestra la estructura del *Espacio de Tuplas*.

Las *tuplas* en el *Espacio de Tuplas* son almacenadas en una tabla, llamada **Tabla de Tuplas**. Cada fila de la misma contiene los siguientes campos:

nombre: identifica al nombre de una *tupla* y es de tipo cadena de caracteres. Las *tuplas* que llevan el mismo nombre se llaman **tuplas tocayas** y son almacenadas en la misma fila de la *Tabla de Tuplas*.

signatura: identifica la signatura de tipos de las *tuplas tocayas*. Es del tipo cadena de caracteres y está formado por una letra (i, f, o s) por cada campo de la *tupla* según su tipo sea entero, real o

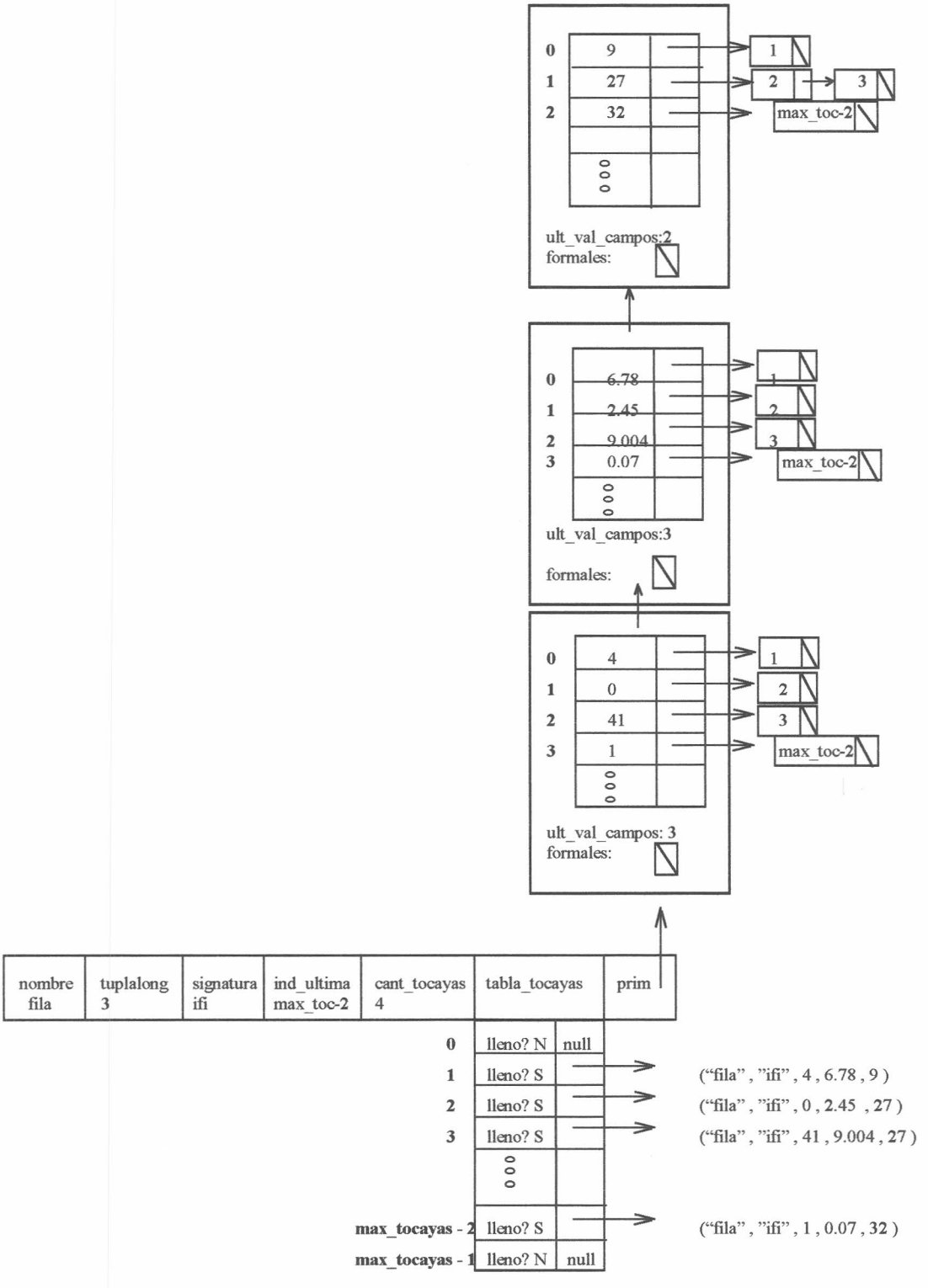

Figura 2: Diseño del Espacio de Tuplas

cadena de caracteres respectivamente, con el signo & antepuesto a la letra cuando se trata de un *campo formal*. Las *tuplas tocayas* tienen la misma signatura.

tuplalong: su valor es la cantidad de campos de la *tupla.* Las *tuplas tocayas* tienen la misma cantidad de campos.

ind_ultima: contiene el índice a la última posición utilizada en la *Tabla de Tuplas Tocayas* (ver abajo *tabla_tocayas*).

cant_tocayas: su valor es la cantidad de *tuplas* almacenadas en la *Tabla de Tuplas Tocayas*.

tabla_tocayas: es una tabla donde se almacenan las *tuplas tocayas*. Cada elemento de esta tabla se compone de una bandera y de un puntero a una estructura del tipo *tupla*. La bandera indica si la entrada contiene una *tupla* o está vacía. En caso de que la entrada esté vacía, el puntero a la *tupla* es nulo.

prim: es un puntero a una lista cuyos nodos constan de una estructura compuesta que describimos más adelante. La cantidad de nodos de esta lista está predeterminada por la cantidad de campos de las *tuplas tocayas*. La estructura de un nodo de la lista consta de la *Tabla de Indices por Valores*, un campo que indica cuántos elementos contiene esta tabla y una lista de *Indices de Campos Formales*. Estas tres estructuras se describen a continuación.

Tabla de Indices por Valores: esta tabla contiene un dato y un puntero a una *lista de índices*. El dato representa el valor que tiene una *tupla* en un campo determinado - dado por la posición del nodo dentro de la lista apuntada por *prim*. El tipo de datos de este valor será igual al tipo de datos de dicho campo. Los índices de la *lista de índices* identifican a la entrada en *la Tabla de Tuplas Tocayas* indicando cuáles son las *tuplas* almacenadas que contienen este valor en el campo.

ult_val_campos: es un índice a la última posición ocupada de la *Tabla de Indices por Valores*.

Indices de Campos Formales: es un puntero a una *lista de índices* que apuntan a la *Tabla de Tuplas Tocayas* e indican cuáles son las *tuplas* almacenadas que contienen un valor formal en el campo determinado por la posición del nodo dentro de la lista apuntada por *prim*.

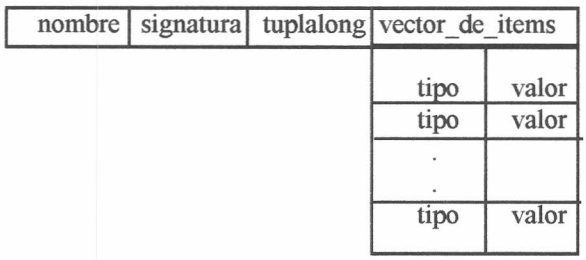

Figura 3: Diseño de la estructura de una tupla

La *Figura 3* muestra la estructura de tipo tupla, que se compone de los siguientes campos:

nombre: almacena el nombre de la *tupla* y es de tipo cadena de caracteres.

signatura: identifica la *signatura de tipos* de la *tupla*. Es del tipo cadena de caracteres y está formado por una letra (i, f, o s) por cada campo de la *tupla* según su tipo sea entero, real o cadena de caracteres respectivamente, con el signo & antepuesto a la letra cuando se trata de un *campo formal*.

tuplalong: su valor es la cantidad de campos de la *tupla* (sin contar el nombre ni la signatura).

vector_de_items: es un vector cuyos elementos son estructuras con dos campos: el primero indica si el item correspondiente es un *campo actual o formal*, y el segundo es el valor del campo, en el caso de que sea un *campo actual*. Los campos pueden ser de tipo entero, real o cadena de caracteres

Las estructuras en lenguaje C que definen una *tupla* y el *Espacio de Tuplas* se encuentran en el archivo *linda.h* listado en el Apéndice V.

La ventaja de utilizar este diseño para el *Espacio de Tuplas* radica en la facilidad para realizar búsquedas de *tuplas* que unifiquen con un *template* recibido a través de un pedido de *in* o de *read*. La búsqueda comienza por el nombre de la *tupla* y continúa en la *Tabla de Indices por Valores*. Para cada campo del *template*, se busca el valor correspondiente en la *Tabla de Indices por Valores*. Si no se encuentra quiere decir que en el *Espacio de Tuplas* no hay *tuplas* que unifiquen con ese *template*. Si se encuentra dicho valor, se toma la lista de índices a *tuplas* que contienen ese valor. Al repetir esto para el siguiente campo, se obtienen otras listas de índices a *tuplas*. Si la intersección entre estas listas de índices es vacía, entonces no existe una tupla en el *Espacio de Tuplas* que unifique con el template. Si la intersección entre estas listas no es vacía se continúa el proceso de búsqueda para el siguiente campo. Cuando este procedimiento termina habiendo unificado todos los campos, se cuenta con una lista de índices a las *tuplas* que unifican con el *template*. De esta lista se elige un índice que apunta a una *tupla* que unifica con el *template*.

4.3 Las comunicaciones subyacentes en el modelo

Una de las primeras decisiones del diseño para implementar *Linda-C* en una arquitectura de red es cómo soportar la memoria lógicamente compartida de *Linda* en un ambiente distribuido.

Un **out-set**, es un subconjunto de los nodos de la red, donde se almacenan las *tuplas* generadas por la primitiva *out*. Un **in-set** es un subconjunto de los nodos de la red donde se envían pedidos de *in* o *read*. Se llama **esquema de distribución uniforme** cuando las *tuplas* se envían a un *out-set* determinado y los pedidos de las mismas se envían a todos los nodos en un *in-set*. Existen, pues, varias posibilidades de *in-sets* y *out-sets* para implementar las primitivas de *Linda*.

En un extremo, podría implementarse el *out* de una *tupla* con un *broadcast* a todos los nodos de la red. En este caso, todos los nodos tendrían una copia completa del *Espacio de Tuplas*. Por lo tanto, el *in* o *read* podrían simplemente requerir una búsqueda local. Este modelo es posible solamente si se cuenta con una red donde la facilidad de *broadcasting* sea confiable, ya que de ésto depende la consistencia del *Espacio de Tuplas*, y por lo tanto de la implementación de *Linda*. La desventaja de este modelo es que la implementación de *Linda* debe controlar cuidadosamente que cuando un

programa realiza un *in* que es satisfecho por el nodo local, la *tupla* devuelta a este programa debe ser eliminida también del resto de los nodos, y esto significa muchas comunicaciones para coordinar las operaciones de todos los nodos (se debería controlar también que no se entregue la misma *tupla* a dos programas que simultánamente realizan un pedido de *in*).

Un modelo utilizado anteriormente [BOB/90] es un **hashing distribuido**, donde a las *tuplas* se les aplica una función de hashing que ocasiona que sean almacenadas en distintos nodos. En este caso, el *out-set* y el *in-set* están determinados por la función de *hashing*. En un esquema de este tipo es más complicado variar la cantidad de nodos que participan en la computación distribuida.

En el otro extremo, la primitiva *out* puede ser implementada almacenando localmente las *tuplas* generadas. En este caso, un *in* o un *read* requieren el envío de un mensaje *broadcast* a toda la red para localizar la *tupla* pedida. En este esquema, la cantidad de nodos que puede participar en una computación distribuida no tiene que ser necesariamente fija, y es fácilmente escalable.

En nuestro trabajo implementamos este último esquema, de manera que el *Espacio de Tuplas* se encuentra distribuido en los nodos de la red, la primitiva *out* envía las *tuplas* al nodo local y las primitivas *in* y *read* envían los pedidos de *tuplas* en forma *broadcast*. En el punto 4.5 se explica detalladamente la implementación de las primitivas *out, in* y *read.*

Las comunicaciones entre los procesos - tanto las *punto a punto* como las *broadcast* - fueron programadas usando las librerías de Berkeley Sockets, sobre la familia de protocolos TCP/IP para equipos Unix. En el Apéndice III se exponen las principales características de la programación con estas librerías. Los mensajes transmitidos entre los nodos de la red local son transformados a un formato común antes de ser transmitidos, y son vueltos a su formato original al ser recibidos.

4.4 El gestor

El **gestor** es el proceso que maneja el *Espacio de Tuplas*. Dado que el *Espacio de Tuplas* se encuentra distribuido en toda la red local, este proceso deberá ejecutarse en cada uno de los nodos de la red local donde se quiera ejecutar programas *Linda*.

El *gestor* que se ejecuta en cada uno de los nodos es el encargado de administrar la porción del *Espacio de Tuplas* que se almacena en dicho nodo. Este proceso debe ejecutarse en forma continua en cada nodo. En el sistema operativo Unix, este tipo de proceso se denomina **proceso demonio** (daemon).

El *gestor* abre un **port de comunicación**, que es una dirección de red por donde escucha **pedidos** originados por los programas *Linda* al ejecutar las primitivas *in, read, out* o *eval.*

Figura 4: Diseño de un requerimiento enviado al gestor

En la *Figura 4* muestra el formato lógico de un *pedido*.

port_fuente: es el *port* de comunicaciones donde el proceso que envía el *pedido* espera un mensaje de respuesta. Este campo tiene un valor sólo cuando difiere del *port* por el que se envió el *pedido*.

operación: indica la *operación* pedida al *gestor*. Puede ser *IN, READ, OUT, EVAL, CLEAN* o *SHOW*. El comportamiento del *gestor* al recibir cada uno de estos tipos de *pedidos* será explicado más adelante.

tupla: es la *tupla* o *template* que viaja en el *pedido*.

4.4.1 Resolución de Pedidos recibidos por el gestor

La ejecución de toda primitiva de *Linda* invocada desde un programa *Linda,* consta de dos etapas: una primera etapa que es ejecutada por el programa *Linda* que la invoca y una segunda etapa que es ejecutada por el *gestor*, que consiste en las operaciones que éste realiza sobre el *Espacio de Tuplas* para llevar a cabo la inserción, lectura o remoción de la *tupla*. La primera etapa se detalla en el punto 4.5 donde se explica la implementación de las primitivas. La segunda etapa se explica a continuación.

Al recibir un **pedido de IN**, el *gestor* busca en el *Espacio de Tuplas* local una *tupla* que unifique con el *template* recibido con el *pedido de IN*. Si no encuentra una *tupla* que unifique con el mismo, agrega este *pedido de IN* junto con la dirección IP de la máquina donde se originó el pedido, en una **lista de pedidos pendientes**. Si encuentra una *tupla* que unifique con el *template* recibido en el *pedido*, se intenta establecer una conexión con el proceso que realizó el pedido mediante la primitiva *in.* Como se detalló en la *Figura 4*, el *gestor* conoce el *port* de comunicaciones por donde el programa *Linda* está escuchando ya que éste viene incluído en el *pedido de IN*.

Como el *pedido de IN* es enviado en forma *broadcast* a todos los *gestores* de la red local, puede ocurrir que sean varios los *gestores* en condiciones de satisfacer este pedido. Todos estos intentarán establecer la comunicación con el port recibido en el pedido, pero sólo uno de ellos lo logrará, ya que como se detalla en la implementación del *in,* una vez que un *gestor* establece la comunicación con el proceso y envía la *tupla* que satisface el pedido, se cierra la conexión y el proceso deja de escuchar por ese port.

Cuando el pedido de conexión es aceptado, se envía la *tupla.* El protocolo de transporte utilizado es **TCP** -Transmission Control Protocol- [RFC/793], por su confiabilidad y porque es un protocolo que ofrece un **servicio orientado a conexión** [TAN/88]. Una vez transmitida la *tupla*, el *gestor* borra la *tupla* del *Espacio de Tuplas* y cierra la conexión con el programa *Linda*.

```
Pedido de IN (template)
        Buscar tupla en el Espacio de Tuplas que unifique con template
        Si no la encuentro
                Agregar a la lista de pedidos pendientes (dirección IP+port+tipo in+template)
        Sino ( encontré una tupla que unifica )
                Trato de establecer conexión con el proceso en dirección IP+port
                Si el pedido de conexión es aceptado
                        envío la tupla
                        borro la tupla del Espacio de Tuplas
                        cierro la conexión
                Sino ( otro gestor puede haber contestado antes que yo )
                        no hago nada
                fin si
        fin si
fin Pedido de IN
```

Figura 5: Pseudocódigo de un pedido de IN recibido por el gestor

Al recibir un **pedido de READ**, el *gestor* procede de la misma manera que al recibir un *pedido de IN*. Los *pedidos de IN* y los *pedidos de READ* que no pueden ser satisfechos se almacenan en la misma *lista de pedidos pendientes*. Cada pedido conserva el tipo de operación (*IN* o *READ*). La única diferencia entre el *pedido de READ* y el *pedido de IN* es que luego de satisfacer el *pedido de READ* el *gestor* no borra la *tupla* del *Espacio de Tuplas*.

```
Pedido de READ (template)
        Buscar tupla en el Espacio de Tuplas que unifique con template
        Si no la encuentro
                Agregar a la lista de pedidos pendientes(dirección IP+port+tipo read+template)
        Sino ( encontré una tupla que unifica )
                Trato de establecer conexión con el proceso en dirección IP+port
                Si el pedido de conexión es aceptado
                        envío la tupla
                        cierro la conexión
                Sino ( otro gestor puede haber contestado antes que yo )
                        no hago nada
                fin si
        fin si
fin Pedido de READ
```

Figura 6: Pseudocódigo de un pedido de READ recibido por el gestor

Al recibir **un pedido de OUT** enviado por un programa *Linda*, el *gestor* almacena en el *Espacio de Tuplas* la *tupla* recibida con el pedido de *OUT* , aunque verifica previamente si existe un *pedido pendiente de IN o READ* para ser satisfecho. En caso de que exista más de un pedido de este tipo, el gestor satisface a todos los *pedidos de READ* y a un sólo *pedido de IN*. Cabe acotar que ésta es una decisión de implementación y no un requerimiento de *Linda*, ya que en función de lo definido en [ARA/89] y [AHU/86], el único requerimiento es satisfacer un subconjunto -que puede ser vacío - de los *pedidos de READ* y un *pedido de IN*. El programador no puede realizar ninguna asumción sobre cuál de todos sus *pedidos de IN* será satisfecho.

Para satisfacer a los *pedidos pendientes* a partir de un *pedido de OUT* recibido, el *gestor* recorre la *lista de pedidos pendientes* verificando cuáles de estos pedidos esperan una *tupla* que unifica con la recién recibida. En todos los casos de *pedidos de READ*, y en todos los *pedidos de IN* hasta que uno sea satisfecho, trata de establecer una conexión con el proceso que realizó este pedido. Si algún proceso recibió con anterioridad la *tupla* esperada, entonces el *gestor* no logrará

comunicarse. En el resto de los casos, enviará la *tupla.* A medida que recorre la lista de pedidos, va eliminando de la misma a aquellos que podría satisfacer (logre comunicarse o no).

En caso de no haber satisfecho ningún *pedido de IN,* guarda la *tupla* en el *Espacio de Tuplas*.

```
Pedido de OUT (tupla)
        Verifico si existen pedidos pendientes de in o read que puedan ser satisfechos
        Si no existen
                guardo tupla en el Espacio de Tuplas
        Sino ( sí hay pedidos que puedo satisfacer )
                Para todos los pedidos de read y un solo pedido de in que puedo satisfacer
                        Trato de establecer conexión con el proceso en dirección IP+port
                        Si el pedido de conexión es aceptado
                                envío tupla
                                cierro la conexión
                        Sino ( otro gestor puede haber contestado antes que yo )
                                no hago nada
                        fin si
                        borro el pedido de la lista de pedidos pendientes
                fin para
                Si satisfice a un pedido de in
                        no inserto tupla
                sino
                        inserto tupla
                fin si
        fin si
fin Pedido de OUT
```

Figura 7: Pseudocódigo de un pedido de OUT recibido por el gestor

Al recibir un **pedido de EVAL**, el *gestor* arma un vector con los campos de la *tupla* recibida en el *pedido de EVAL*. Luego averigua el nombre del usuario que realizó este pedido, averigua la plataforma del equipo en donde está ejecutando, y crea un nuevo proceso - hijo del *gestor*- que será ejecutado con la identidad del usuario *Linda* que realizó el *pedido de EVAL*. Esto sirve a varios propósitos, el principal es que todas las operaciones con *tuplas* que realice este proceso conservarán la identidad original del usuario (en el punto 6.1 se verá la importancia de la identificación de *tuplas* de usuarios para que no exista interferencia entre programas de distintos usuarios) . Otro propósito es que el usuario *Linda* puede tener control de los procesos que sus programas *Linda* generan en los distintos nodos.

El *gestor* toma el programa a ejecutar por este nuevo proceso del directorio *linda/exe* que se encuentra debajo del directorio HOME del *usuario Linda* que realizó el *pedido de EVAL*. En este directorio hay un subdirectorio para cada tipo de arquitectura soportada que contiene los programas compilados en este equipo, por este usuario *Linda*. El *gestor* tomará la versión adecuada de acuerdo a la arquitectura del equipo. La estructura de directorios y los ambientes de desarrollo y ejecución son explicados en los puntos 5 y 6.

El *gestor* pasa el vector con los campos de la *tupla* generado en el formato *argc/argv*[2] [KER/88] como parámetros de este nuevo proceso.

El código del *gestor* se encuentra en el Apéndice V.

[2] En los ambientes que soportan lenguaje C, existe una forma de pasar los argumentos de la línea de comandos a un programa cuando éste comienza su ejecución. Cuando la función principal del programa (*main*) es invocada, es llamada con dos argumentos. El primero (convencionalmente llamado *argc*) es la cantidad de argumentos con los que el programa fue invocado, y el segundo (*argv*, por vector de argumentos) es un puntero a un vector de cadenas de caracteres que contienen los argumentos, uno por cadena

```
Pedido de EVAL (tupla)
        Armar un vector con los campos de tupla
        Obtener el nombre del usuario que realizó el pedido
        Obtener la plataforma en la que estoy ejecutando
        Crear un nuevo proceso idéntico a mí,  hijo mío (fork)
        Si soy el padre
                continúo mi ejecución
        Sino ( soy el hijo )
                cambio mi identidad de ejecución a la del usuario que hizo el pedido
                identifico el directorio de donde tomar el programa a ejecutar
                ejecuto el programa pasándole como parámetros los campos de la tupla a través
                        del vector armado, vía argc/argv
        fin si
fin Pedido de EVAL
```

Figura 8: Pseudocódigo de un pedido de EVAL recibido por el gestor

4.4.2 Otros pedidos

Existen dos *pedidos*, *CLEAN* y *SHOW*, que se generan a partir del uso de las *herramientas* *LimpiaTuplasL* y *MuestraTuplasL* (explicadas en el punto 6.3).

Al recibir un **pedido de CLEAN**, el *gestor* obtiene de la *tupla* en el *pedido,* el nombre del *usuario Linda* que envía el *pedido de CLEAN*, y procede a borrar del *Espacio de Tuplas* todas las *tuplas* que fueron generadas por dicho usuario. También borra de la *lista de pedidos pendientes* todos los *pedidos pendientes* generados por dicho usuario. El concepto de *usuario Linda* se explica en el punto 5.2.

Al recibir un **pedido de SHOW**, el *gestor* obtiene de la *tupla* en el *pedido,* el nombre del *usuario Linda* que envía el *pedido de SHOW*, y busca las *tuplas* en el *Espacio de Tuplas* que fueron generadas por ese usuario. También busca todos los *pedidos pendientes* generados por dicho

usuario. Estas *tuplas y pedidos* son grabados en un archivo temporario. Si el usuario que realiza el *pedido de SHOW* es el usuario *linda*, el *gestor* escribe en su log la porción del *Espacio de Tuplas* que él administra.

4.5 Las primitivas in, out y read

Los programas *Linda* utilizan las primitivas *in, out, read, eval* que se encuentran en una librería llamada **liblinda.a** y que es utilizada al compilar programas *Linda.*

Todas estas primitivas reciben como parámetro una *tupla*, cuyo nombre modifican anteponiéndole el nombre del *usuario Linda* en tiempo de ejecución.

La primitiva *out* recibe como parámetro una *tupla* y genera con ésta un *pedido de OUT* que es enviado al *gestor* local. La *tupla* es enviada utilizando el protocolo de comunicaciones **UDP** -User Datagram Protocol- [RFC/768], debido a razones de eficiencia [TAN/88] y a que la comunicación es local, o sea, entre procesos del mismo nodo. Luego de enviar el mensaje con el pedido, el programa sigue su ejecución.

```
out (tupla)
        Generar a partir de tupla y el nombre del usuario un requerimiento de out
        Enviar el requerimiento al gestor local
fin out
```

Figura 9: Pseudocódigo de la función de librería *out*

La primitiva *in* recibe como parámetro un *template* (normalmente con algún *campo formal*) y envía un *pedido de IN* en forma *broadcast* a todos los *gestores* que ejecutan en los nodos de la red local. Previamente, abre un *port* de comunicaciones utilizando el protocolo *TCP* y establece el valor de la cola por la que aceptará pedidos de conexión en uno (1), por lo que aceptará un solo pedido de conexión de alguno de los *gestores.* El protocolo *TCP* es utilizado como transporte por

razones de confiabilidad [TAN/88]. Junto con el *pedido de IN broadcast* es enviada la dirección del *port* de comunicaciones abierto por el que este proceso se queda escuchando a la espera de un pedido de conexión por parte de algún *gestor*. Cuando ésto ocurre, queda establecida una conexión *tcp* a través de la cual el *gestor* envía la *tupla* al proceso, y una vez finalizada la transmisión se cierra la conexión. Si ésto no ocurre, el proceso queda bloqueado escuchando por el *port* mencionado anteriormente.

La otra tarea que realiza la implementación de la primitiva *in* es guardar las direcciones de memoria de retorno de los *campos formales* del *template* utilizado en el *in*, y devolver en estas direcciones cada uno de los campos cuando la *tupla* es recibida.

```
in (template)
        Guardo las direcciones de las variables de template
        Abrir un port de comunicaciones TCP para recibir solo una conexión
        Generar a partir de template, el nombre del usuario, dirección IP y port el requerimiento
        Enviar el requerimiento de in en forma broadcast a todos los gestores
        Ponerme a escuchar en el port en espera de un pedido de conexión de algún gestor
        Cuando el pedido llega, acepto, y el gestor me transfiere la tupla
        Copio los campos de la tupla recibida en las direcciones que guardé al comenzar
fin in
```

Figura 10: Pseudocódigo de la función de librería *in*

La primitiva *read* se comporta de la misma forma que la primitiva *in*, excepto que el pedido enviado al gestor es un *pedido de READ* y no de *IN*.

```
read (template)
        Guardo las direcciones de las variables de template
        Abrir un port de comunicaciones TCP para recibir solo una conexión
        Generar a partir de template, el nombre del usuario, dirección IP y port el requerimiento
        Enviar el requerimiento de read en forma broadcast a todos los gestores
        Ponerme a escuchar en el port en espera de un pedido de conexión de algún gestor
        Cuando el pedido llega, acepto, y el gestor me transfiere la tupla
        Copio los campos de la tupla recibida en las direcciones que guardé al comenzar
fin read
```

Figura 11: Pseudocódigo de la función de librería *read*

La *Figura 12* muestra un ejemplo del uso de las primitivas *in*, *out*, y *read*.

```
master()
{       int p, res;
while ....
        out("primos", "i", p);      ...
while ...
        worker();           ...
while ...
        in("resultado", "&i", &res);
        printf( ... );
}
```

Figura 12: Ejemplo del uso de las primitivas *Linda-C*

En el Apéndice IV se encuentran programas de ejemplo escritos en *Linda-C*. El código de las primitivas se encuentra en el Apéndice V.

4.6 La primitiva eval

El propósito de la primitiva *eval* es el de crear una *tupla* cuya evaluación genera un nuevo proceso. En [BOB/90] y [BAL/94] se puede encontrar una disertación interesante sobre la semántica del *eval*. Como mencionamos al introducir los conceptos de *Linda*, hay muchas consideraciones y

preguntas sin respuesta con respecto al funcionamiento del *eval* y su implementación. Éstas provienen de la falta de concreción en la descripción de la primitiva. De esta falta de concreción surgen inconsistencias en la definición de lo que el *eval* debería hacer. Algunas de éstas son presentadas a continuación:

- Cómo conocen los nodos de la red las funciones que eventualmente pueden tener que ejecutarse en ellos. Por ejemplo, si *f(x)* es una función de librería existente en el nodo A, pero no en el resto de los nodos, cómo se resolvería *eval(f(x))*.
- Cómo se actualizan los valores de las variables locales de un programa si la función invocada en el *eval* las recibe por referencia, siendo que la función y el programa se ejecutan en espacios separados de memoria (incluso en distintos nodos), aunque originalmente la función esté declarada dentro del mismo programa.
- En qué momento se realiza la evaluación de los parámetros de las funciones que son argumentos de un *eval*.
- Qué construcciones del lenguaje *host* pueden ser aceptadas como argumentos de un *eval* y cuáles no. Por ejemplo, es necesario definir si se pueden utilizar como argumentos de un *eval* descriptores de archivos, punteros, recursión, funciones de llamadas al sistema, etc, y en caso de permitirlas, cuál sería su semántica.

En este trabajo proponemos una semántica para el *eval* respetando lo que a nuestro entender constituye su sentido principal, que es la generación de procesos. La implementación de la misma se detalla en el punto 4.6.2.

4.6.1 Una primera aproximación

La *Figura 13* muestra una primera aproximación a la implementación de la primitiva *eval*, en la que no se realiza comunicación alguna con el *gestor*. Esta primera aproximación crea un nuevo proceso en el nodo local, que ejecutará la función pasada como nombre de la *tupla* que es argumento del *eval*. Una implementación de este tipo, requiere que la función a ser evaluada esté incluída en el código del mismo programa *Linda* que realiza la invocación al *eval*.

```
#include <stdio.h>

void eval(f)
void (*f)();
{
        int pid;
        pid=fork();
        switch (pid) {
        case -1:
                exit(1);
                break;
        case 0:
                (*f)();
                exit(0);
                break;
        default:
                break;
        }
}
```

Figura 13: Una primera aproximación a la implementación del *eval*

La desventaja obvia de esta implementación es que no se realiza ningún balanceo de carga, ya que todo nuevo proceso será ejecutado en forma local. Esta característica impide a esta implementación ser considerada distribuida, aunque no pierde las propiedades de concurrente. Otra desventaja de esta implementación es que no preserva el formato de *tupla* que debería ser utilizado con la primitiva *eval* y que se reduce a la creación de un nuevo proceso. Tampoco permite que la función invocada en el *eval* reciba argumentos, y si bien por la forma en que funciona el *fork* la función "evaluada" tiene acceso a los valores de las variables globales del programa que la invoca, no ocurre la recíproca si el nuevo proceso modifica alguna de estas variables.

La ventaja obvia de esta implementación es su simplicidad.

Una mejora inmediata a esta implementación consiste en agregar la posibilidad de pasar parámetros al ejecutar el nuevo proceso. Cabe aclarar que en esta mejora, la función de la que se

realiza el *eval* debe estar preparada para recibir sus parámetros en el formato *argc/argv*[3] [KER/88].

Esta mejora no resuelve ninguno de los problemas que hemos estado mencionando, y conserva, salvo por el pasaje de argumentos a la función "evaluada", las desventajas de la primera aproximación presentada.

```
#include <stdio.h>

void eval(f, argc, argv)
void (*f)(int,char*[]);
int argc;
char *argv[];
{
        int pid;
        pid=fork();
        switch (pid) {
        case -1:
            exit(1);
            break;
        case 0:
            (*f)(argc,argv);
            exit(0);
            break;
        default:
            break;
        }
}
```

Figura 14: Modificación de la primera aproximación al *eval*

La *Figura 14* muestra el código con la implementación de la mejora que planteamos. En la próxima sección se explica el modelo que proponemos e implementamos en este trabajo para la primitiva *eval.*

[3] En los ambientes que soportan lenguaje C, existe una forma de pasar los argumentos de la línea de comandos a un programa cuando éste comienza su ejecución. Cuando la función principal del programa (*main*) es invocada, es llamada con dos argumentos. El primero (convencionalmente llamado *argc*) es la cantidad de argumentos con los que el programa fue invocado, y el segundo (*argv*, por vector de argumentos) es un puntero a un vector de cadenas de caracteres que contienen los argumentos, uno por cadena

4.6.2 Creación distribuida de procesos

En esta sección detallamos la implementación final de la primitiva *eval* propuesta en nuestro trabajo. Comenzamos explicando la sintaxis de la primitiva y continuamos con la definición de la semántica propuesta. Finalmente describimos la implementación realizada. En las siguientes secciones, al introducir el ambiente de desarrollo y ejecución propuesto, se completa el cuadro del funcionamiento del *eval.*

La sintaxis de la primitiva *eval* es idéntica a la sintaxis de las otras primitivas en el sentido que el parámetro que recibe la primitiva *eval* es una *tupla*, y como tal, cuenta con un nombre, una signatura de tipos, y cero o más campos. Las *tuplas* generadas por la primitiva *eval* reciben el nombre de **tuplas activas**, ya que son *tuplas* que, por causar que se genere un nuevo proceso para su evaluación, pueden ser a su vez, generadoras de nuevas *tuplas*.

```
master()
{       int t;
        float res;

        out("tareas", "iffs", t, 98.006, 7.354, "dividir");      ...
...
        eval("worker", "i", t);      ...
while ...
        in("resultados", "i&f", t, &res);
        printf( ... );
}
```

Figura 15: Ejemplo de uso de la primitiva *eval*

La semántica de una *tupla activa* es distinta a la semántica de las otras *tuplas*, en el sentido que el nombre de una *tupla activa* indica el nombre de un programa a ejecutar en algún nodo de la red, los campos de la *tupla activa* son los parámetros que acepta ese programa y la *signatura* de la *tupla activa* son los tipos de datos de los parámetros.

El programa ejecutable que denota el nombre de la *tupla activa* debe estar previamente compilado en el equipo en que se ejecute la primitiva *eval* y residir en un directorio especial del usuario cuyo programa invoca al *eval*. En los puntos 5 y 6 se describe el ambiente de desarrollo y ejecución

distribuida que implementamos para facilitar esta operatoria. En la *Figura 15* se muestra un ejemplo del uso de la primitiva *eval*.

Dado que en nuestra implementación el *eval* recibe una *tupla* cuyo nombre referencia a un programa y no a una función definida en el programa invocante, la semántica del *eval* cambia en este aspecto, y por lo tanto muchas de las preguntas sin respuesta no son planteables. Esto requiere evidentemente un cambio en la forma de escribir los programas, ya que requiere que en lugar de incluir una función en el archivo que contiene el cuerpo principal de programa, se escriba su código en un archivo separado. Esto implica también que el nuevo programa debe prepararse para recibir sus parámetros desde la línea de comandos. En la sección de Trabajos Futuros planteamos el desarrollo de un precompilador que realice ésta y otras tareas, que resolvería alguna de las preguntas acerca del *eval*.

Cuando un proceso de un usuario *Linda* ejecuta la primitiva *eval* se genera un *pedido de EVAL* al *gestor*. Al igual que las otras primitivas, *eval* agrega el nombre del usuario *Linda* que la invocó, al nombre de la *tupla activa*. Esto permitirá al *gestor* al que se envíe el pedido, generar un proceso cuya identidad de ejecución (execution user id en la terminología utilizada en Unix) coincida con la del usuario que generó el pedido.

La implementación de la primitiva *eval* elige un *gestor* en la red para enviarle el *pedido de EVAL* con la *tupla* activa. De esta forma se determina en qué nodo se ejecutará el nuevo proceso. La elección de este nodo es hecha mediante un balanceo de carga simple con la idea de enviar la *tupla activa* al nodo de la red con menor carga. Esta elección se implementó mediante la invocación de una función llamada *EligeHost*, que a su vez invoca para cada nodo que figura en un archivo del usuario *Linda*, a otra función que le devuelve la carga de este nodo. Con la información obtenida, *EligeHost* decide cuál es el nodo al que se debe enviar la *tupla activa*. Una vez resuelto ésto, se envía un *pedido de EVAL* al *gestor* que ejecuta en el nodo elegido por el balanceo de carga, y se escribe en un log el nombre de dicho nodo.

De lo explicado en el párrafo anterior se desprende que la cantidad de nodos utilizada en diferentes ejecuciones de programas *Linda* puede variar.

5. Diseño del ambiente de desarrollo distribuido

5.1 Motivación

Consideramos que una implementación de *Linda* que pretenda ser más que un prototipo debe contar con una **infraestructura** que la soporte y posibilite a un programador, o a cualquier usuario interesado en este modelo, a hacer uso de la misma.

Una de las propiedades del modelo distribuido es que a partir de un programa se pueden originar varios procesos que cooperan entre sí. Idealmente cada uno de estos procesos podría ejecutarse en un nodo de la red. Para que esto sea posible, el código de estos procesos debe previamente ser compilado en cada uno de estos nodos. Pero en una red es posible que existan grupos de usuarios que no tengan acceso a todos sus nodos (no tengan un usuario definido en los nodos), y por lo tanto sus programas no deben (ni pueden) ser compilados en estos nodos. Esto llevaría a los usuarios a enviar los fuentes de sus programas a los nodos donde tengan acceso, y a compilarlos en ese nodo, en forma manual, cada vez que desarrollan un programa. En el caso de nodos con igual plataforma de hardware, la tarea podría limitarse a mandar una copia del ejecutable (originándola siempre desde el nodo cuya versión del sistema operativo sea la más vieja), pero esto es imposible de aplicar entre nodos de distinta plataforma.

Por otro lado, dado que a partir de un *eval* ejecutado en un programa se debe generar un nuevo proceso (sin intervención del usuario), y que sería ideal que su ejecución se realice de acuerdo a la carga de los nodos a los que el programador tiene acceso, éste debería indicar en el momento de la compilación el lugar dentro de la estructura de directorios de donde los archivos ejecutables serán tomados por el *eval*.

Por estas y otras razones, creemos que un ambiente de desarrollo distribuido debería proveer al programador las siguientes facilidades:

- Dejar establecido una sola vez en qué nodos sus programas pueden ser ejecutados, y deben por lo tanto ser compilados. Tener opción de modificar esta información a lo largo del tiempo.

- Contar con una estructura de directorios similar en todos los nodos de la red a los que tiene acceso, que le facilite su trabajo mediante el conocimiento de los lugares comunes donde dejar fuentes y ejecutables.
- Contar con una estructura de directorios tal que si su directorio HOME está montado sobre NFS[4], sus archivos ejecutables no se "pisen" con las versiones ejecutables de los mismos programas compilados en un nodo con otra plataforma de hardware, desde la que monta el mismo HOME.
- Compilar sus programas *Linda* en todos los nodos en los que definió que puede trabajar, sin preocuparse por la transmisión de sus fuentes, ni por la plataforma de hardware de los nodos.
- Compilar sus programas *Linda* solo en el nodo local para no gastar recursos en otros nodos, hasta no contar con un fuente libre de errores.

Nuestra implentación del ambiente de desarrollo para *Linda* cubre todos estos requerimientos.

5.2 Concepto de usuario Linda

El concepto de *Usuario Linda* es introducido en este trabajo, y se define como aquel usuario de los equipos Unix que cuenta con todas las facilidades del ambiente de desarrollo que anteriormente mencionamos. En nuestra implementación, cualquier usuario de un nodo donde se instale *Linda* (Ver Apéndice II - Manual del Administrador) puede convertirse en *usuario Linda* en ese nodo, mediante la ejecución de un solo comando, **CrUsrLinda**, que forma parte de las herramientas desarrolladas. Este procedimiento modifica el archivo de inicialización de sesión (.profile o .login según el shell inicial del usuario) para facilitar el acceso a las herramientas de *Linda.* Durante la ejecución de este procedimiento el usuario debe ingresar los nombres de los nodos *Linda* a los que puede (y desea) acceder. Esta información será usada tanto durante las compilaciones como durante las ejecuciones de sus programas *Linda*, y puede ser modificada en el futuro con la herramienta **AgregaHosts**. Otra de tareas llevadas a cabo por *CrUsrLinda* es la de construir debajo del directorio HOME del usuario la estructura de directorios descripta en la *Figura 16.*

[4] NFS (Network File System) es marca registrada de Sun Microsystems,Inc. Permite montar sistemas de archivos ubicados físicamente en un nodo de la red, desde otro nodo, y accederlos como si fueran sistemas de archivos locales.

El directorio *src* es el lugar donde el usuario dejará sus fuentes *Linda*.

En los subdirectorios del directorio *exe* quedarán los archivos ejecutables al compilar los programas *Linda* con las herramientas provistas para tal fin.

En el directorio *env* se almacena el archivo *lindaHosts* , que contiene los nombres de los nodos ingresados por el usuario durante la ejecución del procedimiento *CrUsrLinda.* Vale aclarar que si el usuario ingresa un nombre erróneo (de un nodo inexistente), los controles que se realizan impiden que este nombre sea agregado a este archivo.

En el directorio *smp* se copian programas de ejemplo escritos usando las primitivas de *Linda*, y un archivo con información para la compilación de los mismos (makefile).

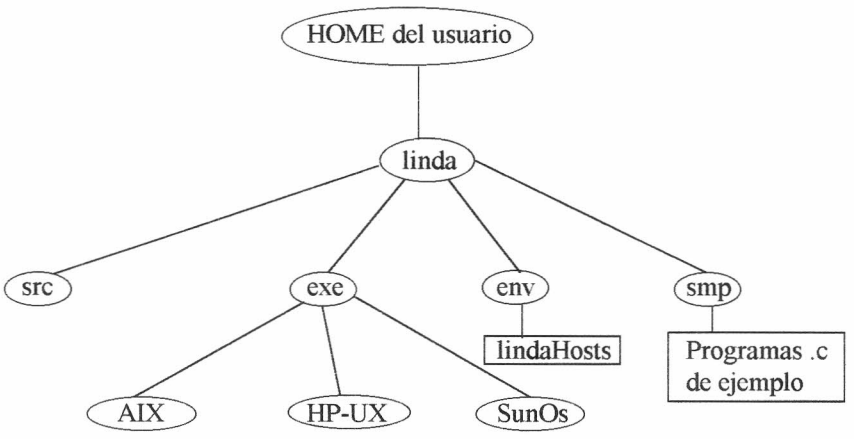

Figura 16: Estructura de directorios de un usuario *Linda*

El código de *CrUsrLinda* está en el Apéndice VI.

5.3 La estructura multiplataforma

Como mencionamos anteriormente, un usuario *Linda* debe contar con herramientas que le permitan compilar sus programas en nodos de diferentes plataformas de hardware en forma transparente. Para que esto sea posible, diseñamos la estructura multiplataforma que puede observarse debajo del directorio *exe* en la *Figura 16.* Cuando se invoca al compilador *Linda*, éste se encarga de dejar el programa compilado en uno de los subdirectorios, según la plataforma donde se ejecute (IBM, Hewlett-Packard o Sun). A primera vista, parecería suficiente contar con el directorio *exe* sin los subdirectorios, ya que lo importante es tener un ejecutable compilado en cada nodo, de acuerdo a su plataforma. Pero qué ocurriría si el HOME del usuario estuviera en un disco montado a través de NFS[5] por más de un nodo de la red? Quedaría en el directorio *exe* solo la versión ejecutable compilada en último lugar. Podemos imaginar lo que ocurriría en tiempo de ejecución cuando cualquiera de los otros nodos intentara ejecutar este programa! Contando con la estructura definida en nuestro ambiente, si un usuario es usuario *Linda* en tres nodos de la red, llamemos A, B y C, con AIX[6], HP-UX[7] y SunOs[8] respectivamente, y estos nodos son clientes NFS de tal manera que el directorio HOME del usuario se encuentra físicamente en un solo lugar (a pesar de ser visto desde los tres nodos como propio), entonces, si el usuario logoneado en A (o B o C indistintamente) mandara a compilar su programa en forma distribuida, obtendría una versión ejecutable debajo de cada uno de los directorios AIX, HP-UX y SunOs.

5.4 Compilación distribuida de programas Linda

Ya hemos mencionado al hablar de la estructura multiplataforma, que un programador puede compilar un programa *Linda* utilizando el "front end" de compilación provisto por nuestra implementación, y que la versión ejecutable del programa para las distintas plataformas será dejada en el directorio correspondiente debajo del directorio *exe*. Profundizaremos ahora sobre la herramienta de compilación de *Linda*: el par **lcomp** y **lcompl**.

---

[5] NFS (Network File System) es marca registrada de Sun Microsystems,Inc. Permite montar sistemas de archivos ubicados físicamente en un nodo de la red, desde otro nodo, y accederlos como si fueran sistemas de archivos locales.
[6] AIX es marca registrada de International Business Machines
[7] HP-UX es marca registrada de Hewlett-Packard Co.
[8] SunOs es marca registrada de Sun Microsystems,Inc.

El *lcompl* es el compilador de *Linda* local. Dado que los programas *Linda* son programas C que hacen uso de las primitivas de *Linda*, *lcompl* acepta todos los argumentos que acepta el compilador C, y se los pasa al invocarlo, junto con los parámetros necesarios para incluir las primitivas de *Linda*. *lcompl* deja el archivo ejecutable en el directorio que corresponda según la plataforma de hardware donde el usuario esté compilando. Si el usuario desea sacar provecho de las distintas plataformas, puede hacerlo escribiendo en su fuente porciones de código dependientes del hardware donde se compile (utilizando las directivas al preprocesador de C #ifdef, #ifndef y #endif), dado que la compilación se realiza además estableciendo una variable del compilador según la plataforma.

El *lcomp* es el compilador de *Linda* distribuido. Al igual que *lcompl* puede recibir los mismos argumentos que el compilador C. *lcomp* no realiza una compilación distribuida a menos que la compilación local (*lcompl* invoca a *lcomp*) sea exitosa. Si detecta que la compilación local es satisfactoria, envía el fuente a ser compilado al directorio *linda/src* del usuario en todos los nodos especificados en el archivo *lindaHosts* (Ver *Figura 16*), y lo compila, dejando los ejecutables en los subdirectorios del directorio *exe* de acuerdo a la plataforma.

El código de *lcomp* y el de *lcompl* están en el Apéndice VI.

6. Diseño del ambiente de ejecución distribuido

6.1 Motivación

Mientras desarrollábamos el código del *gestor* (el adminstrador del *Espacio de Tuplas*) y de las primitivas *in*, *out* y *read*, nos planteamos el caso de dos o más usuarios programando en *Linda*, que, tal vez influidos por la lectura de publicaciones o de ejemplos, o probando los mismos, utilizaran los mismos nombres de *tuplas*. Recordemos antes de seguir, nuestras decisiones de implementación de *Linda*: el *Espacio de Tuplas* se constituye a partir de todas las porciones de éste administradas por los distintos procesos *gestores* que se ejecutan (uno por nodo *Linda*), y es compartido por todos los programadores; la primitiva *out* es local, es decir, solamente el *gestor*

ejecutando en el mismo nodo que el programa que la invoca recibe la *tupla*; las primitivas *in* y *read* expanden su pedido de *tupla* en forma broadcast, es decir, el pedido es recibido por todos los *gestores*, y solo uno de los que estén en condiciones de satisfacerlo lo hará; y por último, la *tupla* activa generada por un *eval* para crear un proceso es enviada al *gestor* que esté ejecutando en el nodo que en ese momento tenga más porcentaje de CPU ociosa, lo que causará que procesos de este usuario se ejecuten en varios nodos.

Evidentemente, como ninguno de estos programadores tiene la obligación de conocer los nombres que otros programadores dan a sus *tuplas*, la ejecución simultánea de sus programas sería probable, y la interferencia entre estos un hecho seguro: por un lado, porque dos *tuplas* del mismo nombre no pueden tener distinta *signatura*, por lo que alguno de los usuarios obtendría un error al tratar de hacer un *out* de una *tupla* tal que otra con el mismo nombre y distinta *signatura* ya estuviera en el *Espacio de Tuplas*. Por otro lado porque, si por casualidad coincidieran las *signaturas* de las *tuplas* de igual nombre pero de distintos usuarios, los *read*s e *in*s de los procesos de un usuario podrían ser satisfechos con las *tuplas* de otro usuario, y seguramente ninguno de estos obtendrían los resultados esperados. Se necesitaba, por lo tanto, una forma de identificar las *tuplas* de los distintos usuarios en el *Espacio de Tuplas* para evitar los problemas mencionados anteriormente, sin necesidad de obligar a los programadores a tomar ninguna convención para armar los nombres de sus *tuplas*, hecho que los perjudicaría.

Otra duda que se nos planteó durante el desarrollo fue, de qué forma un usuario que probando sus programas poblara el *Espacio de Tuplas* se las arreglaría para eliminarlas, y poder así reanudar sus pruebas. La respuesta inmediata es: haciendo un programa que haga *in*s de las *tuplas* que su programa en estado de prueba hizo *out*s (para eliminarlas del *Espacio de Tuplas*), y *out*s de las *tuplas* que su programa hizo *in*s (para sacarlas del *Espacio de Tuplas* como pedidos pendientes). ¿Pero cuántos tendría que hacer? ¿Podría saberlo exactamente? Consideramos que no. Se necesitaba, por lo tanto, una forma de mostrarle al usuario cuáles son sus *tuplas* en el *Espacio de Tuplas*, y cuáles *tuplas* están en el *Espacio de Tuplas* como pedidos pendientes (en espera de un *out*, para satisfacer un *read* o un *in*). Y por qué no, una forma rápida de eliminar todas sus *tuplas* del *Espacio de Tuplas*.

Para terminar, mientras desarrollábamos el *eval*, nos pareció que un usuario querría ver en qué nodos sus procesos se estarían ejecutando (los lanzados a partir del *eval*, de acuerdo a la carga de CPU de los nodos a los que el usuario tiene acceso). Se necistaba, por lo tanto, una herramienta que le permitiera ver sus procesos en todos los nodos especificados en su archivo *lindaHosts*.

6.2 Multiples conjuntos de procesos sin interferencia entre sí

Cuando hablamos de conjuntos de procesos que no interfieran entre sí, nos referimos a procesos de distintos usuarios. En una red bien administrada un nombre de usuario repetido en dos o más nodos corresponden a la misma persona. Este usuario podría además, ser usuario *Linda* en estos nodos. Por lo tanto, el usuario podría mandar a ejecutar sus programas *Linda* desde diferentes nodos, o dejar esta tarea a la primitiva *eval*, que se encargará de decidir a qué nodo mandar a ejecutar un proceso.

De esto se desprende que si el nombre del usuario forma parte del nombre de la *tupla*, entonces, aunque dos usuarios nombren a sus *tuplas* con el mismo nombre, sus nombres "finales" serán diferentes. De esta forma se logra que no interfieran. Esta solución es rápida y simple de implementar, y es la que elegimos.

Las modificaciones al código solo se requirieron en las primitivas que se ejecutan del lado cliente (las que forman parte de la librería). Estas modificaciones consisten en averiguar la identidad del proceso (que es la misma que la del usuario que lo mandó a ejecutar) que viene dada por un número de usuario. Luego se averigua el nombre del usuario a partir del número. Finalmente, se transforma el nombre de la *tupla* con la que se está trabajando - ya sea para hacer un *in*, un *read*, un *out*, o un *eval*- anteponiendo al nombre original el nombre del usuario seguido de algún caracter especial hasta completar una determinada cantidad de caracteres. No hizo falta modificar el código del *gestor* en lo que a satisfacer pedidos de *in*s, *read*s o *out*s se refiere.

El código de las primitivas se encuetra en el Apéndice V.

6.3 Balanceo de carga utilizado por la primitiva eval

Como se explicó en el punto 4.6.2, nuestra implementación del *eval* utiliza una función *EligeHost*, que averigua cuál de los hosts al que el usuario *Linda* tiene acceso (especificados en su archivo *lindaHosts*) es el más indicado para mandar a ejecutar el proceso. Si bien hay muchos parámetros a analizar para calcular la carga de un equipo, en nuestra implementación optamos por considerar únicamente el porcentaje de CPU libre como dato para decidir en qué nodo ejecutar, debido a que el balanceo de carga constituye un tema en sí mismo.

Una de las posibilidades para implementar la función *EligeHost*, era que ésta se comunicara con cada *gestor* y que éste devolviera la carga de CPU del nodo donde ejecuta. Pero consideramos que esto traería un *overhead* de comunicaciones, y que se perdería mucho tiempo solamente para decidir a qué *gestor* enviar la *tupla activa.* La forma de hacer más eficiente esta búsqueda era que la información estuviera lista en el momento de necesitarla, y que el método para accederla fuera rápido. Para esto, desarrollamos un programa llamado *CPUidleL* que devuelve el porcentaje de CPU libre en el momento de su invocación, y otro llamado *CPUidle*, que ejecuta remotamente al *CPUidleL* y con la información obtenida sobre los nodos mantiene una pequeña base de datos de carga. El proceso de instalación deja al *CPUidle* ejecutándose en forma periódica a través del proceso *crontab* de Unix, como se especifica en el Apéndice I (Manual del Administrador de Linda-C).

La implementación definitiva de la función *EligeHost* utilizada por el *eval* consulta la base mantenida por *CPUidle*, y compara para decidir el mejor nodo la información de los nodos en los que el usuario es un usuario *Linda.*

El código del *eval* y de *EligeHost* está en el Apéndice V. El código de *CPUidleL* y *CPUidle* está en el Apéndice VI.

6.4 Herramientas para el usuario

Nuestra implementación provee, tanto del *gestor* como de la librería de primitivas, tres versiones con distintos niveles de mensajes (enviados a través del *error standard*), que pueden ayudar al programador a ver lo que sus programas van ejecutando. Estas llevan el nombre **gestor.dbg0**, **gestor.dbg1** y **gestor.dbg2** en el caso del gestor, y **liblinda.a.dbg0, liblinda.a.dbg1** y **liblinda.a.dbg2** en el caso de la librería. Las versiones listas para usar (*gestor* y *liblinda.a*) son las mismas que las ".dbg0" y solo mandan mensajes de error. Las versiones ".dbg1" mandan mensajes sobre los pasos que se van ejecutando, y las versiones ".dbg2" incluyen todos los mensajes de las versiones anteriores y muestran además datos internos de la implementación.

Las otras herramientas desarrolladas suplen las necesidades mencionadas en "Motivación". Detallamos su implementación a continuación.

Para visualizar el *Espacio de Tuplas* y *tuplas* pendientes en el *gestor* local, el usuario dispone de **MuestraTuplasL**. Esta herramienta utiliza un servicio brindado por el *gestor* para satisfacer un *pedido de SHOW*. En el código de *MuestraTuplasL* se procede de igual manera que para realizar un *out* , pero con la diferencia que la operación que viaja en el requerimiento al gestor es un *pedido de SHOW*. El resultado obtenido es un archivo cuyo nombre está formado por el nombre del usuario seguido de una secuencia de caracteres - para evitar escribir sobre un archivo problablemente creado por el usuario - , que queda en el directorio corriente donde el *gestor* está ejecutándose. Este archivo contiene las *tuplas* del *Espacio de Tuplas* local y los pedidos pendientes (que algún proceso está esperando).

Para visualizar el *Espacio de Tuplas* y pedidos pendientes en los *gestores* de los nodos especificados en el archivo *lindaHosts*, el usuario dispone de **MuestraTuplas**. Esta herramienta es un "front end" que invoca a *MuestraTuplasL* en forma remota y copia los archivos generados por éste al nodo local, agregándole como extensión el nombre del nodo.

Para eliminar las *tuplas* del *Espacio de Tuplas* y los pedidos pendientes en el *gestor* local, el usuario dispone de **LimpiaTuplasL**. Esta herramienta utiliza un servicio brindado por el *gestor* para satisfacer un *pedido de CLEAN*. En el código de *LimpiaTuplasL* se procede de igual manera que para realizar un *out* , pero con la diferencia que la operación que viaja en el requerimiento al *gestor* es un *pedido de CLEAN*. El resultado obtenido es que todas las *tuplas* del usuario son eliminadas, y no queda ningún pedido pendiente.

Para eliminar las *tuplas* del *Espacio de Tuplas* y los pedidos pendientes en los *gestores* de los nodos especificados en el archivo *lindaHosts*, el usuario dispone de **LimpiaTuplas**. Esta herramienta es un "front end" que invoca a *LimpiaTuplasL* en forma remota.

Para saber qué procesos fueron lanzados a partir de un *eval* en el nodo local, el usuario dispone de **MuestraLProcsL.** Esta herramienta muestra en la salida estándar los procesos del usuario, que son hijos del *gestor*.

Para saber qué procesos fueron lanzados a partir de un *eval* en los nodos especificados en el archivo *lindaHosts*, el usuario dispone de **MuestraLProcs.** Esta herramienta muestra en la salida estándar los procesos del usuario, que son hijos de los *gestores*.

7. Conclusiones

En nuestro trabajo introducimos un nuevo concepto de ambiente de desarrollo y ejecución distribuido-paralelo e implementamos una versión de *Linda* adaptada al modelo propuesto. También definimos una nueva semántica para la primitiva *eval*, que permite su implementación. Esta concepción del *eval* está fuertemente ligada con el ambiente de desarrollo y ejecución definido. El trabajo incluyó:

- diseño y codificación de la estructura del *Espacio de Tuplas*, beneficiando el tiempo de búsqueda de *tuplas* en el mismo.

- diseño y codificación del *gestor*, para resolver pedidos de *in, out, read* y *eval,* y dos nuevos pedidos surgidos a partir del ambiente de ejecución propuesto: *show* y *clean.*

- diseño y codificación de las comunicaciones entre los programas *Linda* y el *gestor,* realizando los *out*s en forma local, y los *in*s y *read*s en forma broadcast.

- diseño y codificación de las primitivas de *Linda* (*in*, *out* y *read*), incluyéndolas en una librería para ser usada por el programador *Linda.*

- diseño y codificación de la primitiva *eval* de la que se implementaron dos versiones: una versión simple totalmente local, sin comunicación con el *gestor*, y otra versión compleja que incluye balanceo de carga para distribuir los procesos generados a partir de la misma.

- diseño e implementación del ambiente de desarrollo distribuído *Linda*. Definimos el concepto de "usuario *Linda*" e implementamos el procedimiento de creación del mismo, y las herramientas para realizar compilaciones distribuidas, que consideramos indispensables para que una implementación de *Linda* no sea solamente un prototipo.

- diseño e implementación del ambiente de ejecución distribuido *Linda*, brindando un soporte multiusuario - sin interferencia entre *tuplas* de procesos que no cooperan - y herramientas de ayuda al programador *Linda*.

8. Alcances

Como explicamos en la introducción de este trabajo, existen dos primitivas de *Linda* mencionadas en [CAR/87], inp y readp, cuya función es básicamente la misma que la de las primitivas in y read, pero no son bloqueantes. Nuestra implementación no cuenta con estas primitivas

Un **gateway** es una computadora que conecta distintos tipos de redes. En el ambiente de las redes locales muchas veces se llama *gateway* a una computadora que cuenta con dos o más interfaces de

red, de manera que está conectada a dos o más redes locales. Los paquetes de tipo *broadcast* en las redes que funcionan con la familia de protocolos TCP/IP no son capaces de pasar a través de los *gateways*. Si la red local donde se corren los programas *Linda* cuenta con dos o más segmentos unidos por *gateways*, no será posible enviar a distintos segmentos de la red los pedidos de *tuplas* realizados con un *in* o *read en* forma *broadcast*. Nuestra implementación no cuenta con soporte para redes con segmentos unidos por *gateways*. Queda como extensión a este trabajo proveer al *gestor* que ejecuta en un *gateway* de una función tal que al recibir un *pedido de IN* o *READ* por una de las interfaces de red, re-envíe este pedido por la(s) otra(s) interface(s).

Nuestra implementación de *Linda* y de los ambientes de desarrollo y ejecución está preparada para soportar varias plataformas. Si bien los fuentes C, los procedimientos de instalación de ambiente y las herramientas están preparadas para ser compilados y ejecutar en hardware IBM, Hewlett-Packard y Sun (en sus versiones para sistema operativo Unix), proveemos solamente la versión compilada del *gestor*, de *MuestraTuplasL*, de *LimpiaTuplasL* y de la librería de primitivas *liblinda.a* para HP-UX, ya que el desarrollo de esta tesis se realizó en una red local de equipos Hewlett-Packard.

Nuestra implementación provee un balanceo de carga simple ya que éste es un tema en sí mismo y excede los alcances de esta tesis. De todas formas, el código C provisto es modular y puede ser fácilmente modificado para mejorar el algoritmo y los datos de balanceo de carga utilizados.

9. Futuros trabajos

- Desarrollar un precompilador *Linda*, que permita contener en un solo fuente el conjunto de programas del tipo *master-worker* y se encargue de su separación dejando distintos módulos ejecutables. Analizar la factibilidad y la forma de traducir una llamada a *eval* como una llamada a *eval* seguida de una llamada a *in*, con una template que contenga las variables que sean pasadas por referencia a la función "evaluada". Como complemento, analizar la factibilidad de incorporar a la definición de una función invocada en un *eval* , la invocación de un *out* con un *tupla* que contenga las variables pasadas por referencia. Con esto, se intentaría

solucionar parte del problema de pasaje de parámetros a funciones dentro del *eval* mencionado en este trabajo.

- Desarrollar un compilador *Linda* incorporando al mismo las ideas propuestas en [CAR/87] sobre soporte en tiempo de compilación para realizar búsquedas.
- Desarrollar librerías con las primitivas de *Linda* para ser utilizadas con otro lenguaje *host*
- Analizar la terminación distribuida de programas *Linda*
- Estudiar cómo afecta la caída de un nodo a una red *Linda* y buscar posibles implementaciones tolerantes a fallas.
- Incorporar a la actual implementación el soporte de más tipos de datos para los campos de las tuplas
- Incorporar a la actual implementación los puntos detallados en el "Alcance"
- Realizar un estudio de *performance* de la actual implementación de *Linda-C*

10. Bibliografía

[AHM/93] Ahmed, Carriero, Gelernter
A Program Building Tool for Parallel Applications
YALEU, diciembre 1993

[AHU/86] Ahuja, Carriero, Gelernter
Linda and Friends
Computer IEEE, agosto 1986

[ARA/89] M.Arango, D.Berndt
TSnet:A Linda Implementation for Networks of Unix-based Computers
YALEU/DCS/RR739, agosto 1989

[BAC/86] Maurice J. Bach
The Design of the Unix Operating System
Prentice-Hall, 1986

[BAL/89] H. Bal, J. Steiner, A. Tanenbaum
Programming Languages for Distributed Computing Systems
ACM Computing Surveys, Vol 21, No 3, septiembre 1989

[BAL/94] Leonardo Balbiani
Estudio, diseño e implementación de un kernel Linda
FCEyN (UBA), Informe final de beca de investigación para estudiantes, 1994

[BEN/90] M. Ben-Ari

Principles of concurrent and distributed programming
Prentice-Hall, 1990

[BOB/90] M. Bobrowski, L. Balbiani
Creación dinámica de Procesos en C-Linda
Trabajo de Pasantía (ESLAI), 1990

[CAR/94] N. J. Carriero, E. Freeman, D. Gelernter, D. Kaminsky
Adaptive Parallelism and Piranha
YALEU, febrero 1994

[CAR/87] N. J. Carriero
Implementation of Tuple Space Machines
Research Report YALEU/DCS/RR567. Yale University, diciembre 1987

[CHA/89] K.M. Chandy, J. Misra
Parallel Program Design: A Foundation
Addison-Wesley 1989

[DAV/85] Davidson S., Garcia-Molina H., Skeen D.
Consistency in Partitioned Networks
ACM Computing Surveys, 17(3):341, 1985

[FAC/91] Michael Factor, Scott Fertig and D. H. Gelernter
Using Linda to build Parallel AI Applications
YALEU/DCS/TR-861, junio 1991

[GEL/85] D. H. Gelernter
Generative communications in Linda
ACM Transactions on Programming Languages and Systems, 7(1):80-112, 1985

[GEL/87] D. H. Gelernter
Programas para computación avanzada
Investigación y Ciencia, diciembre 1987

[HEW/so] Berkeley (BSD) Sockets: Programming Guide
Hewlett Packard

[KER/84] B.Kernighan, R. Pike. The Unix Programming Environment
Prentice-Hall, 1984

[KER/88] B.Kernighan, D. Ritchie
The C Programming Language
Prentice-Hall, 1988

[RFC/768] Request For Comments 768
User Datagram Protocol

Advanced Research Project Agency

[RFC/793] Request For Comments 793
Transmission Control Protocol
Advanced Research Project Agency

[TAN/88] A. Tanenbaum
Computer Networks, second edition
Prentice-Hall, 1988

[TAN/92] A. Tanenbaum
Modern Operating Systems
Prentice-Hall, 1992

[TAN/96] A. Tanenbaum
Computer Networks, third edition
Prentice-Hall, 1996

[WEN/90] E. P. Wentworth
Parallelism via Linda: A Transputer Application
Rhodes University, PPG 90/5, mayo 1990

[WEN/92] E. P. Wentworth, P.G. Clayton
An hierarchical approach for supporting Linda's shared, associative tuple space on transputers
Rhodes University, PPG 92/5, abril 1992

[WHI/88] R Whiteside, J.Leichter
Using Linda for SuperComputing on a Local Area Network
YALEU/DCS/TR638. Yale University, junio 1988