# JSimured
## Simulador de Redes de Multicomputadores Paralelo


VNIVERSITAT DE VALÈNCIA

Escuela Técnica Superior de Ingenierías
Departamento de Informática

Proyecto final de Carrera de Ingeniería Informática:

Oscar Bayo Vega
*Dirigido por* Dr. Fernando Pardo Carpio

Mayo de 2005

*A mis padres*

# Agradecimientos

A mi familia, por darme la posibilidad de estudiar una carrera, y ante todo soportarme

Al Doctor Nebot, sin él ahora no estaría aqui

A Little, Sergio, y todos los compañeros del club de debate

A Manel, por su gran ayuda y consejos

A los compañeros de carrera,

A J. Peris, por esas risas en prácticas

A Pedro, Jaime, Miguelio, Dani, por esas partidas de ajedrez memorables

A Chovares, que lugares!

A Pableiras, por esos grandes ratos de biblioteca

A Emilio y David, por su dedicación en el videoclub

A Vicente, sigo esperando, recuerdalo

A mi tutor, Fernando Pardo, por estar siempre disponible a ayudar

Al personal del departamento de infórmatica, por darme los conocimientos necesarios para realizar cualquier proyecto

A Mentis, por darme esa ayuda necesaria en los últimos años de carrera

A Somset, por ser algo más que una empresa

y a mis niñas, no os olvido

# Resumen

La arquitectura multicomputador es una técnica de computación que permite realizar grandes cálculos en poco tiempo, pero presenta un gran handicap : su coste. Este es un problema que se ha intentado solucionar con los simuladores. Si realmente se desea estudiar el comportamiento de estas arquitecturas se recurrirá a un simulador. El problema que presentan los simuladores es su tiempo de respuesta y su dependencia de la plataforma de desarrollo.
Nuestro simulador JSimured intenta solventar este problema recurriendo a la programación paralela. Este proyecto realiza la simulación de una red de Multicomputador de forma paralela mediante el uso de hilos, intenta aprovechar el hardware disponible para acelerar la simulación y obtener tiempos de simulación inferiores a un simulador secuencial. Incluso evita en su medida ser dependiente del sistema operativo utilizado.
En esta memoria se introducen los conceptos necesarios, se presenta el método utilizado para desarrollar dicho paralelismo y se comentan los resultados obtenidos.

# Índice general

# Índice de figuras

# Capítulo 1

# Motivación y Objetivos

## 1.1. Motivación

A lo largo de estos años cursando Ingeniería Informática me surgió la curiosidad de trabajar con computadoras de más de un procesador. Fue entonces cuando Fernando Pardo me ofreció la posibilidad de paralelizar su simulador de multicomputadores (SimuRed) y poder realizar pruebas en un multiprocesador.

Este proyecto nace de la necesidad de disminuir los tiempos de simulación, y evitar la perdida de tiempo esperando resultados, asi como proporcionar una herramienta educacional multiplataforma para el estudio de las redes de interconexión de los multicomputadores.

Dada mi orientación curricular centrada en asignaturas del ámbito Hardware me pareció una buena manera de retomar un campo bastante descuidado por mi parte como la programación de software.
Por este motivo, el presente proyecto de fin de carrera se convirtió más bien en un reto personal, consitente en poder evaluar mis conocimientos tanto de hardware como de software.

## 1.2. Objetivos

Los objetivos de este proyecto son:

- **Paralelizar el simulador** Al intentar ejecutar de forma simultánea algunas partes del simulador se espera reducir el tiempo de respuesta. Se tendrá que prestar mucha atención a las partes que son paralelizables y a la dependencia de datos
- **Aumentar el rendimiento de la herramienta** Se tendrá que buscar el método idóneo que reduzca al mínimo la espera

- **Convertir el simulador de multicomputadores en una herramienta multiplataforma** Se buscará que el simulador sea ejecutable desde cualquier plataforma, sin realizar ningún cambio en el código, y a ser posible no requiera de ningún software adicional que suponga algún desembolso económico
- **Integrar el simulador en web** Integrando el simulador en forma de aplicación web nos aseguramos que sea fácilmente accesible y ejecutable

## 1.3. Estructura de la memoria

En una primera parte se verán unos conceptos previos necesarios para comprender el difícil campo de la computación paralela. Se estudiarán también algunos de los simuladores que existen en la actualidad y nos adentraremos en el complejo mundo de los multicomputadores.
Una vez explicado el contexto del proyecto se procederá al análisis del simulador desarrollado por Fernando Pardo para realizar en base a este un simulador con prestaciones paralelas. Por último, se comentarán las pruebas realizadas y se sacarán las conclusiones del trabajo realizado y sus posibles modificaciones y mejoras.

# Parte I

# Fundamentos de la investigación

# Capítulo 2

# Conceptos fundamentales

En los siguientes apartados se intentará dar unas definiciones básicas de los conceptos que se van a utilizar en este trabajo. Estas definiciones se irán ampliando según la importancia que vayan ganando en el proyecto.

## 2.1. Introducción

La aparición de los computadores a mediados del siglo XX ha supuesto una gran ayuda para la investigación. Con la llegada de los ordenadores personales la presión social ha obligado a aumentar las prestaciones de los computadores. Hoy en día es difícil no tener un computador para realizar cualquier tipo de procesamiento, desde el más simple procesado de texto hasta los cálculos matemáticos más complejos.

Dicha mejora en las prestaciones se abordó en principio en diversas innovaciones tecnológicas tales como la integración de circuitos VLSI, el aumento de la frecuencia de reloj en los procesadores, el aumento del tamaño en el bus de control o de datos, etc... Pero estos avances tienen un límite marcado por la naturaleza física de los componentes lo que hizo que se pensara en otra manera de abordar el problema de rendimiento. Es entonces cuando se dirigió la vista a la arquitectura que presentaban los ordenadores.

## 2.2. Arquitectura de los computadores

La clasificación más popular se le debe a Flynn(Figura 2.1), que clasificó las arquitecturas de acuerdo a los flujos de datos (data streams) y a los flujos de instrucciones (instruction streams). El concepto de flujos de datos se refiere al número de operandos que se pueden procesar al tiempo y el de flujos de instrucciones se refiere a cuantos programas se pueden ejecutar al tiempo. De acuerdo a su clasificación existen cuatro tipos de computadoras:

- ***SISD:***(Single instruction stream, single data stream): un solo flujo de instrucciones y un solo flujo de datos.
- ***SIMD:***(Single instruction stream, multiple data stream): un solo flujo de instrucciones y varios flujos de datos.
- ***MISD:***(Multiple instruction stream, single data stream): Varios flujos de instrucciones y un solo flujo de datos.
- ***MIMD:***(Multiple instruction stream, multiple data stream): Varios flujos de instrucciones y varios flujos de datos.

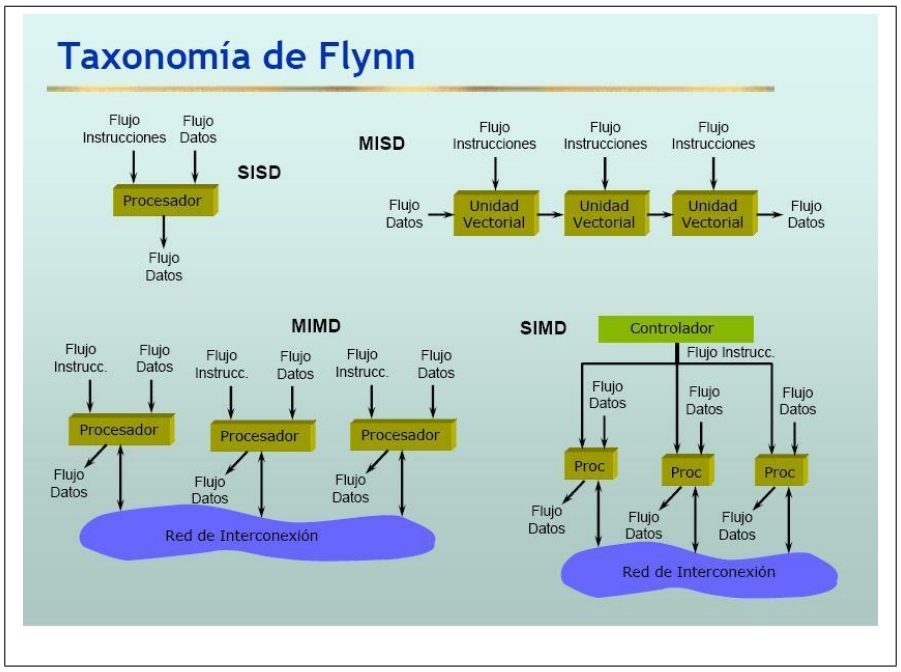


Figura 2.1: Taxonomía de Flynn

El primer tipo (SISD) se refiere a las maquinas de arquitectura Von Newmann como los PC's que manejamos comúnmente. Como sabemos este tipo de maquinas tienen limites físicos que limitan su capacidad de procesamiento, esto motiva la investigación en maquinas de arquitectura en paralelo para obtener un mayor rendimiento. Es precisamente este esquema MIMD, la arquitectura que está basada en tener múltiples procesadores cada uno de ellos trabajando sobre diversos datos, la que puede ofrecer este aumento de prestaciones.

Una dificultad para definir esta arquitectura paralela y además clasificarla es la gran cantidad de características que deben considerarse.

**Multiprocesador:** máquina de procesamiento paralelo con un conjunto de procesadores homogéneos. Puede haber memoria local, compartida o ambas. El número de Elementos de Proceso(EPs) varía de pocos a miles.

**Multicomputador:** sistema distribuido con un conjunto de procesadores, típicamente heterogéneos, cada uno con memoria local y (quizás) con una memoria global compartida.

**Diferentes clasificaciones:** por el mecanismo de control, por organización del espacio de direcciones, por la granularidad de los procesadores y por la red de interconexión

En la actualidad, los multicomputadores son considerados como la tecnología mas prometedora para alcanzar una capacidad de procesamiento más allá de los teraflops. Tales computadores de gran escala se organizan como una replicación o aglomeración de unidades funcionales de tratamiento muy parecidas a las que encontramos en una máquina clásica. Cada una de estas unidades reciben el nombre de nodos. Cada nodo tiene su propio procesador, su memoria local y otros periféricos.

### 2.2.1. La arquitectura Multiprocesador

Los sistemas multiprocesador(Figura 2.2)son un tipo de arquitectura con una importancia creciente y ampliamente difundido. La mayoría de los constructores de computadores ofrece máquinas en las que están presentes más de una CPU, configuración que es hoy en día es de uso habitual en casi todos los sistemas de tamaño medio y grande. Asimismo, los fabricantes de procesadores incorporan a sus arquitecturas, desde hace pocos años, los mecanismos necesarios para que éstos se puedan emplear fácilmente, y con un coste reducido (publicidad de Sun Microsystems en 1999: "si compra un procesador, le regalamos otro"), en la construcción de este tipo de sistemas.

En los Multiprocesadores la implementación y ejecución de procesos paralelos contempla una única forma de comunicación entre ellos restringida pero muy rápida, materializada a través de lecturas/escrituras de posiciones compartidas de memoria en las que se alojen variables reconocidas por todos los procesos o sea de carácter global.

### 2.2.2. La arquitectura Multicomputador

Los Multicomputadores o sistemas de memoria distribuida(Figura 2.3) son arquitecturas en los que la comunicación explícita entre unidades se

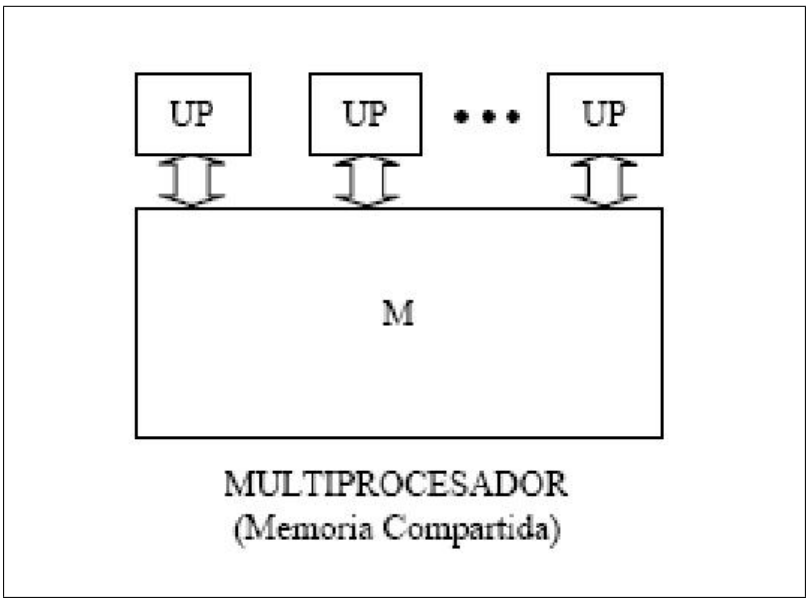


Figura 2.2: Esquema de un multiprocesador

realiza a través de una **Red** de paso de mensajes y donde, a diferencia de los sistemas multiprocesador, cada unidad de procesamiento opera sobre su propia memoria.

El mecanismo de paso de mensajes es el que permite a los procesos de un Multicomputador correr en forma integrada, coherente y sincronizada a fin de que la ejecución de cada uno resulte una cooperación con la ejecución global de una aplicación que, en pos de obtener un mejor rendimiento, haya sido distribuida entre las distintas unidades.

## 2.3. Concurrencia y Paralelismo

La Real Academia de la Lengua Española define Concurrencia como 'Acaecimiento o concurso de varios sucesos en un mismo tiempo.' Si en esta definición sustituimos suceso por 'proceso' ya tenemos una primera definición de lo que va a ser la concurrencia en programación[TPGCSFQA03]. Cuando se dispone del hardware necesario para ejecutar simultáneamente procesos se habla de 'Paralelismo'.

Un programa concurrente define un conjunto de acciones que pueden ser ejecutadas simultáneamente, mientras que un programa paralelo es un tipo de programa diseñado para ejecutarse en un sistema multiprocesador.

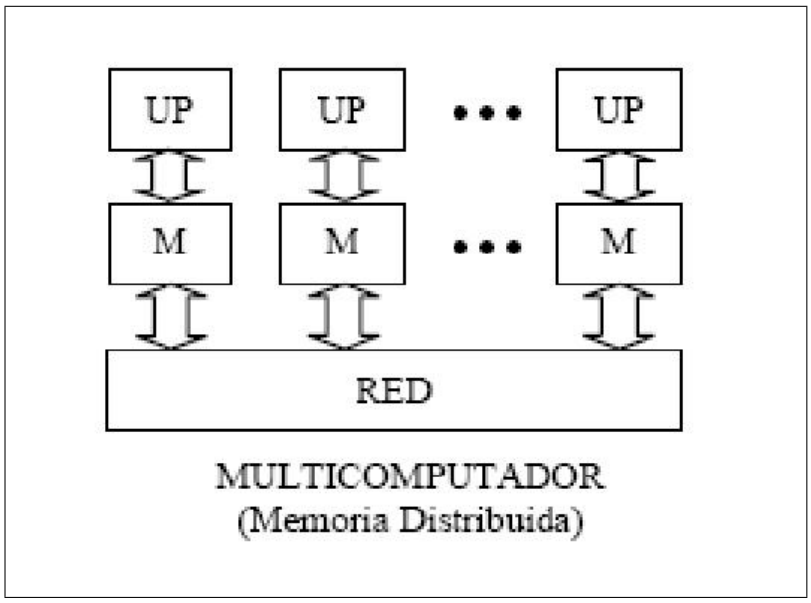


Figura 2.3: Esquema de un multicomputador

### 2.3.1. Concepto de programa

Un programa es una estructura pasiva. Se trata de un conjunto de instrucciones, una secuencia de líneas de código que dicen que hacer con unos datos de entrada para producir unos datos de salida. Por tanto, el programa no es más que un fichero almacenado en un disco, para que el programa haga algo en realidad hay que ponerlo en ejecución.

### 2.3.2. Concepto de proceso

Se puede definir un proceso como un programa en ejecución. Es decir, un proceso es algo dinámico, que incluye el código del programa, sus datos, y el estado de los recursos que necesita el programa para ejecutarse. Esta definición es incompleta pues un programa puede estar compuesto por varios procesos que se ejecutan al mismo tiempo. Cuando varios procesos se ejecutan concurrentemente necesitan mecanismos para **comunicarse** y **sincronizarse** con los demás procesos.

El sistema operativo, que es el que proporciona la abstracción proceso, mantiene por cada uno una estructura de datos privada(a la que solo tiene acceso el propio sistema operativo) denominada bloque de control del proceso(BCP), donde almacena la información necesaria para describir el estado del proceso. Este modelo tradicional de proceso tiene dos características básicas:

- Ejecución secuencial. Las instrucciones se ejecutan de forma estrictamente secuencial, salvo cuando recibe notificación estricta del sistema operativo. Aún así, la ejecución se interrumpe únicamente durante la ejecución del manejador y después continua donde fue interrumpida.
- Ejecución independiente. La ejecución de un proceso es independiente del resto de procesos del sistema, ya que cada uno tiene su espacio de direcciones privado, inaccesible por el resto de procesos del sistema a menos que se decida compartirlo de forma explícita.

Los procesos son entidades pesadas, su estructura está en la parte del núcleo y cada vez que el proceso quiere acceder a ella tiene que hacer una llamada al sistema, consumiendo tiempo extra de procesador. Por otra parte, los cambios de contexto entre procesos son costosos en cuanto a tiempo de computación.

### 2.3.3. Concepto de hilo

De la misma manera que se pueden ejecutar varios procesos al mismo tiempo, dentro de un proceso pueden existir varios hilos de ejecución. Un **hilo**(thread en inglés) puede definirse como cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente.Un proceso tradicional sería un proceso que contiene un único hilo.
Un hilo es una entidad ligera, su estructura reside en el espacio de usuario.

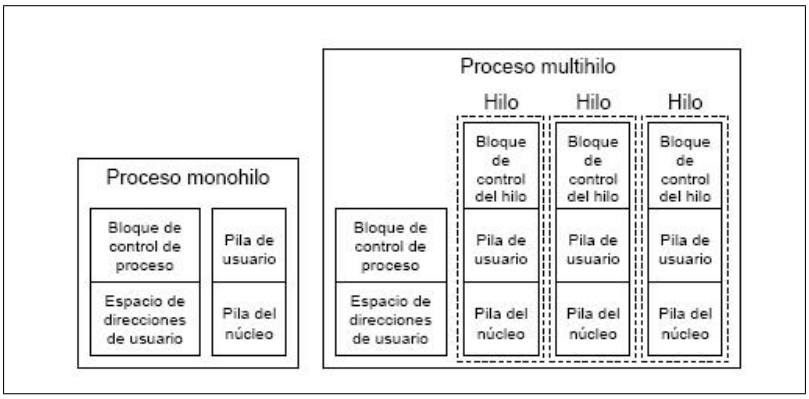


Figura 2.4: Representación de Procesos e Hilos

Los hilo comparten la información del proceso, por ello, cualquier recurso que tenga asignado el proceso puede ser utilizado por cualquiera de sus hilos, independientemente del hilo que lo haya solicitado.(Figura 2.4) Si un hilo modifica una variable del proceso, el resto de hilos verán esa modificación cuando accedan a esa variable.
Los cambios de contexto entre hilos consumen poco tiempo de procesador.

## 2.4. ¿Qué es un simulador?

La simulación es una de las herramientas de análisis más poderosas de las que se dispone para el diseño y operación de procesos o sistemas complejos. El concepto de simulación es simple e intuitivo, consiste en diseñar ecuaciones matemáticas que modelicen el sistema que se quiere simular. En otras palabras, expresar mediante cálculos matemáticos y estadísticos una situación del mundo real. El objetivo de este estudio es tratar de permitir al usuario de experimentar con sistemas(reales o propuestos) en casos donde sería imposible o impracticable.

Un simulador nos proporciona un marco para el análisis de modelos y permite experimentar con ellos para tomar decisiones acerca de los sistemas que estos representan. En el estudio que nos concierne, con una sola computadora estaremos simulando una red de multicomputadores, con el consiguiente ahorro económico, pero además nos permite estudiar diferentes topologías y situaciones con un simple cambio de parámetros en el simulador.

# Capítulo 3

# Estado del Arte

## 3.1. Simuladores de Multicomputadores

Desde 1980 se viene trabajando con los simuladores de multicomputadores. Muchos de estos simuladores se centran en emular un tipo de máquina existente y estudiar el comportamiento de esta sin incurrir en sus coste hardware, otros por contra estudian el comportamiento general de un multicomputador.

### 3.1.1. Multicomputer Simulator (SIMON)

En 1983 Richard Fujimoto[Fuj83] crea el primer simulador de multicomputadores. Se trata de un simulador que modela la ejecución de programas paralelos en un multiprocesador. El simulador ejecuta un conjunto de programas como si se estuvieran ejecutando en procesadores separados, y realiza estadísticas de la ejecución completa. Una parte del simulador, el "modelo Switch"simula el paso de mensajes entre procesadores a través de la red de interconexión.
El simulador se desarrolló para albergar diferentes tipos de interconexión hardware(ej: packet switches, crossbars , ...) con solo cargar el modulo apropiado. Casi todos los simuladores de redes para multicomputadores están de alguna manera inspirados en este primer trabajo. Dada su importancia, la información que se facilita es más bien escasa, y no se ha podido conseguir el código para su testeo.

### 3.1.2. PP- MEss-SIM

Desarrollado por miembros de la IEEE, pp-mess-sim es un entorno de simulación, orientado a objetos, basado en eventos, para la evaluación de redes de inteconexión en sistemas de paso de mensajes[RFDS97]. El simulador proporciona una toolbox con distintas topologías de red, distintas cargas de trabajo, algoritmos de routing, y modelos del encaminador. Permite hacer

un estudio del rendimiento de la red y del trabajo del simulador (su carga). Esta escrito en C++ lo que permite una gran flexibilidad para el desarrollo, al estar la mayor parte de componentes divida en clases. Su estructura se muestra en la figura 3.1

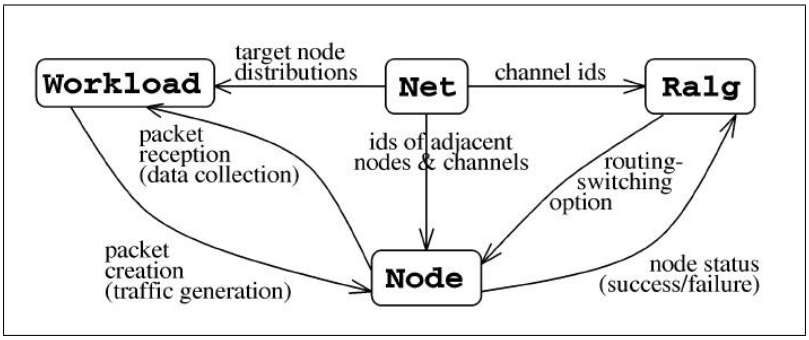


Figura 3.1: Estructura del simulador pp-mess-sim

### 3.1.3. SimuRed: Simulador de Redes de Multicomputadores

No se podía realizar un repaso del estado del arte sin mencionar el SimuRed de F.Pardo [Par03]. Es un sencillo simulador de redes de multicomputadores escrito en C++. Simula el envío de paquetes a través de una red presentando estadísticas de tiempos consumidos, bloqueos producidos, etc. Permite reconfigurar la red y el proceso de envío de los paquetes, además de permitir especificar los paquetes a enviar mediante un fichero de trazas. El programa es completamente visual y permite incluso ver la evolución de los paquetes a través de la red.

Es una herramienta muy buena para la eseñanza del funcionamiento de las redes de multicomputadores. Existe también una versión en línea de comandos, que al no ser visual, solo muestra los resultados estadísticos.

## 3.2. Simuladores de Redes

Como se ha visto anteriormente, la implementación de simuladores esta en pleno y constante desarrollo. En esta incursión en el mundo de los simuladores de redes veremos los principales y más utilizados.

### 3.2.1. Network Simulation Testbed (NEST)

Desarrollado en la universidad norteamericana de Columbia (USA) es un simulador de propósito general que permite la modelación de sistemas distribuidos de redes así como de protocolos básicos.[Dup88]

La versión analizada es la 2.6 + patch 3.

Este sencillo simulador escrito en lenguaje C proporciona un entorno gráfico simple y sustenta su arquitectura en un modelo cliente/servidor. Sus principales inconvenientes son que actualmente está sin mantenimiento, lo que limita muchísimo el soporte a los posibles usuarios, y que proporciona unas características de modelado excesivamente simples para nuestro estudio. Por otro lado, la lentitud en la ejecución de las simulaciones desaconseja su uso para modelos medios o grandes.

### 3.2.2. Maryland Routing Simulator (MARS)

Como su nombre indica, el MARS es un simulador de redes desarrollado en el departamento de ciencias de la computación de la universidad de Maryland (USA)[ADZM91]. Este programa nace como la evolución de otro simulador más antiguo denominado NetSim. La versión analizada es la 2.1.

Este programa desarrollado en C dispone de dos interfaces, uno en modo simple y otro en modo gráfico desarrollado con las librerías gráficas X Motiv. Su principal característica es la de proporcionar un entorno de simulación de algoritmos de rutado simple, lo que no permite el estudio de todo el abanico de posibilidades que necesitamos.

Por otro lado también carece de un soporte estable en la actualidad y no proporciona las capacidades de modelar comportamientos de sistemas autónomos o simulaciones en tiempo real.

### 3.2.3. Realistic And Large network simulator (REAL)

Este programa desarrollado en la de la universidad de Cornell (USA) [Kes97] se construyó a partir del simulador NEST dotándolo de varias mejoras para el estudio de la congestión de redes conmutadas. Entre sus características principales cabe destacar una mayor eficiencia en la simulación que su predecesor, lo que minimiza el tiempo de ejecución y un soporte nativo para la familia de protocolos TCP/IP.
La versión analizada es la 5.0.

Este programa no permite el estudio de sistemas o parámetros que no afecten de forma directa al flujo de conexiones TCP/IP principal. De esta forma, la capacidad de modelar un sistema real queda muy limitada.
Como sucede en los anteriores programas, el soporte ofrecido para este simulador es prácticamente inexistente en la actualidad.

### 3.2.4. Network simulator 2 (Ns-2)

El simulador Ns-2 [oSC95] parte del código escrito para el simulador REAL con el objetivo de crear un simulador discreto de redes TCP. Incorpora las capacidades de rutado y multicast en redes de cable y satélite (wireless connection).
Este proyecto está patrocinado por DARPA, Xerox y el instituto de ciencias de la información (ISI) de la universidad del sur de california (USA).
La versión analizada es la 2.7.

Este simulador permite la modelación de las estructuras deseadas y actualmente ofrece un cierto soporte al usuario. Sus principales inconvenientes son la complejidad de las estructuras que utiliza (debido principalmente a la herencia de otros simuladores) y que permite únicamente la simulación discreta de eventos, lo que elimina la posibilidad de simulaciones continuas en tiempo real.

### 3.2.5. S3 project / Scalable Simulation Framework

Este proyecto de simulación creado por la colaboración de DARPA, Institute for security technology studies at Dartmouth y Renesis Corporation es uno de los mas reputados actualmente[ONC99].
Este completo programa de simulación creado en C++ proporciona dos interfaces (APIs) de programación en los lenguajes JAVA y C++ así como un lenguaje específico de modelación denominado DML (Domain Modeling languaje).
Es altamente escalable e implementa masivamente el paralelismo permitiendo el uso de prácticamente todos los protocolos existentes en Internet de forma nativa (IP, UDP, TCP, OSPF, BGP...).
La versión analizada es la 2.0.

Su paradigma de simulación se basa en las cinco clases que proporciona al usuario para modelar su sistema (Entity, inChannel, outChannel, Process y Event) y permite únicamente una simulación discreta. Por otro lado la interacción con la simulación que se realiza es únicamente a través del DML, lo que elimina una rápida interactividad entre el usuario y la simulación.

Como puntos negativos cabe destacar su falta de rendimiento en las versiones gratuitas, ya que la obtención del paquete optimizado es de pago (lo distribuye la empresa Renesys). Las versiones gratuitas de este simulador si ofrecen un cierto soporte al usuario (todo y que no se acercan a la estabilidad y velocidad que proporciona el simulador de pago).

### 3.2.6. JAVA Simulator (J-Sim)

El J-Sim [yT99] es uno de los últimos programas de simulación creados en la comunidad universitaria. Este proyecto nace con la colaboración de NSF, DARPA, CISCO, la universidad de Illinois (USA) y Ohio (USA).
Este simulador está creado en JAVA y TCL utilizando una arquitectura de componentes autónomos. De esta forma se proporciona el conjunto de clases básicas para su funcionamiento y se permite la extensión o creación de nuevas clases mediante JAVA por parte de los usuarios.
La versión analizada es la 1.3.

El control de la simulación se realiza mediante scripts/TCL de forma que existe una consola de trabajo dónde interactivamente podemos ir dando órdenes al simulador. Las simulaciones que soporta este programa son discreta y continua o tiempo real. Al igual que SSFnet proporciona soporte para la gran mayoría de protocolos utilizados en Internet así como multicast y QOS (Quality Of Service).

## 3.3. Simulación de arquitecturas paralelas

### 3.3.1. Linux Memory Simulator (LIMES)

Linux memory simulator: Limes [oB96], es un simulador (conducido por ejecución) que a diferencia de casi la mayoría de sus equivalentes corre en PC i486 (o superior) bajo sistema operativo Linux, por supuesto, originalmente diseñado para correr programas de la colección SPLASH-2 (Stanford Parallel Applications for Shared Memory) que tienen carácter de benchmark. Se trata de programas escritos en 'C' con el apoyo del juego de macros ANL (Argonne National Laboratory) para resolver las cuestiones de carácter global/paralelo.

Para correr Limes cualquier versión de kernel de Linux es adecuada pero es requisito que el compilador sea GNU CC 2.6.3, si se cuenta con uno superior (actualmente la versión es 4.0) basta con copiar sólo los archivos que conforman el entorno de desarrollo (development environment) a un directorio determinado y retocar ligeramente algunos archivos previo a la compilación.

El paquete incluye simuladores basados en cuatro protocolos (snoopy) de coherencia cache, Berkeley, Dragon, WIN (Word Invalidate, un protocolo basado en el método de invalidación parcial de palabra) y WTI (Write-Through Invalidate un protocolo elemental que exhibe la peor performance de todos).
También permite implementar lo que llama Simulador Ideal y que no es

otra cosa que la simulación de un sistema de memoria perfecta en el que todas las referencias se completan en un ciclo y además se admiten accesos simultáneos, es decir que se completan en un solo ciclo todos los accesos que pudieran coincidir para un instante determinado.

### 3.3.2. Multiprocessor Simulator(MulSim)

Es un simulador de sistemas multiprocesadores de memoria compartida [Mat00] cuya ventaja es su simplicidad e independencia de plataforma para correr. A diferencia de Limes resulta portable a casi cualquier versión de Unix incluso Linux. La arquitectura que simula (definida por el lenguaje ensamblador Mas) es similar a la de los procesadores SPARC de Sun Microsystems: las instrucciones se ejecutan en un solo ciclo, se comporta como máquina load/store, las operaciones aritméticas se realizan únicamente en el modo registro-a-registro y tiene un sistema de ventanas de registro usado para almacenar la 'pila global' (runtime stack).

Permite elegir entre cuatro variantes de interconexión de memoria,

- PRAM: un mecanismo idealizado también conocido como 'memoria perfecta' en el cual no sólo los accesos a memoria principal se completan en un solo ciclo sino que además se admiten accesos simultáneos, es decir que todos los accesos programados para un instante dado se completan en un único ciclo de ejecución (útil para ensayar programas pero no arquitecturas). Desde luego este mecanismo sólo es implementable en simuladores, no tanto por la rapidez sino por la simultaneidad y por supuesto que no utiliza memoria caché.
- Bus: un protocolo (también sin memoria caché) en el que sólo se idealiza la velocidad de respuesta de la memoria principal, considerándola un ciclo, pero se permite un único acceso por vez.
- Snoopy-update: un protocolo elemental que simplemente actualiza copias de líneas presentes en más de una caché cuando una de ellas es actualizada; no hay flujo de información entre cachés ni estados que requieran tratamiento especial. Se consideran cachés de tamaño infinito.
- Snoopy-Invalidate: un protocolo elemental que simplemente invalida copias de líneas presentes en más de una caché cuando una de ellas es actualizada; no hay flujo de información entre cachés ni estados que requieran tratamiento especial. Se consideran cachés de tamaño infinito.

MulSim permite además a los usuarios programar su propio sistema de interconexión de memoria.

A diferencia de Limes que es un simulador pensado para correr programas escritos en C con la inclusión de macros de la colección ANL para las operaciones de carácter global, MulSim tiene un compilador para programas escritos en el lenguaje ensamblador Mas, mencionado más arriba, siendo esa la forma más directa, aunque menos práctica, de crear aplicaciones. También se provee un compilador para programas escritos en C con la inclusión de algunas funciones y macros (también provistos) consistiendo el proceso en este caso en la compilación del programa en C a su equivalente en Mas y luego la obtención del ejecutable (para el equipo host o del usuario) mediante el compilador anterior.

### 3.3.3. SMPCache

SMPCache [VR02] es un simulador mediante trazas para el análisis y la docencia de sistemas de memoria caché en multiprocesadores simétricos. La simulación se apoya en un modelo construido de acuerdo con los principios básicos arquitectónicos de estos sistemas. El simulador posee una interfaz gráfica completa y amigable, y opera en sistemas PC con Windows.

Es un simulador educacional muy completo. Algunos de los parámetros que se pueden estudiar con el simulador son: Localidad de los programas; influencia del número de procesadores, de los protocolos de coherencia caché, de los esquemas para arbitración del bus, de la función de correspondencia, de las políticas de reemplazo, del tamaño de caché (bloques en caché), del número de conjuntos en caché (para cachés asociativas por conjuntos), del número de palabras por bloque (tamaño del bloque de memoria), del ancho de palabra,...

Inicialmente, el simulador fue concebido como una herramienta orientada a la docencia de memorias cachés. Sin embargo, el potencial del sistema desarrollado ha probado su utilidad en el análisis de programas y estrategias de diseño de sistemas de memoria en multiprocesadores. Las características anteriores permiten que el simulador sea usado para comprender el diseño de organizaciones que ejecutan de forma óptima un determinado tipo de programas paralelos o para mejorar el modo de operación de una arquitectura paralela concreta.

Ha sido elegido por William Stallings como herramienta de simulación para la realización de proyectos de estudiante [Sta03]

## 3.4. El lenguaje de Programación C++

El lenguaje C/C++ es un lenguaje compilado, lo que implica su dependencia de la plataforma en la cual se quiere ejecutar. Existen varios

compiladores de C++, uno de ellos es el GNU GCC. Es una 'colección' de compiladores que permite crear programas con C, C++, Objetive C, Fortran, Java y Ada. También puede usarse en múltiples plataformas hardware y hay versiones para varios sistemas operativos (Linux y Windows incluidos). Lo mejor de este compilador es que su licencia es GPL.

Por otro lado esta el C++ Builder de la casa Borland. El entorno de desarrollo de C++ Builder es simple, flexible y potente al mismo tiempo, cuenta con un gran número de componentes prefabricados que facilitan de forma notable la creación de cualquier aplicación. El compilador, por su parte, está altamente optimizado, procesa el código a una velocidad vertiginosa y genera ejecutables de un tamaño razonable. Tiene la desventaja de tener que pagar una licencia para su uso. C++ Builder lanzó en 2003 una versión libre para Linux, utilizando el entorno Kylix que permitía el uso de librerías cruzadas para Windows y Linux, pero el proyecto fracasó y actualmente ya no tiene soporte.

### 3.4.1. Los hilos en C/C++

#### Hilos POSIX

Existen multitud de librerías en C sobre hilos. Una de las comunes y más utilizada es libpthread, la librería de hilos POSIX. La especificación POSIX (IEEE 1003.1c) está pensada para todas las plataformas y está disponible para la mayoría de las implementaciones UNIX y Linux, así como para VMS y AS/400. Consta de más de un centenar de funciones para el manejo de hilos. Implementa para la sincronización de hilos los dispositivos comunes tales como mutex, variables condición, etc. Su uso es bastante simple. Tiene la desventaja de que no es compatible con el sistema operativo Windows. Una buena guía para programar utilizando POSIX se puede consultar en [But97]

#### Hilos C++Builder

C++ Builder proporciona la clase abstracta TThread para la creación de hebras. Además incluye un asistente para su creación y manejo. Como era de esperar Windows y Linux no manejan los mismos códigos, por lo que para utilizar esta clase sobre Linux se convierte en una tarea tediosa, teniendo que definirse el usuario sus propia clase hilo derivada de esta.

## 3.5. El lenguaje de Programación Java

Java es un lenguaje de propósito general, de alto nivel y orientado a objetos. Es un lenguaje de programación orientado a objetos puro, en el sentido

de que no hay ninguna variable, función o constante que no esté dentro de una clase. Se accede a los miembros a través de los objetos y de las clases. El lenguaje Java es a la vez compilado e interpretado. Con el compilador se convierte el código fuente que reside en archivos con extensión **.java**, a un conjunto de instrucciones que recibe el nombre de *bytecodes* que se guardan en archivos de extensión **.class**. Estas instrucciones son indepedientes de la plataforma o sistema opertivo que se este utilizando, por lo que se asegura que el lenguaje Java es **multiplataforma**. Solamente es necesario, por tanto, compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un ordenador.

La **Máquina Virtual de Java** (JVM) es el entorno en el que se ejecutan los programas Java, su misión es la de garantizar la portabilidad de las aplicaciones Java. Define esencialmente un ordenador abstracto y especifica las instrucciones(*bytecodes*) que este ordenador puede ejecutar.

La versión actual es la *1.5*, que incluye la definición de tipos genéricos(al estilo de C++) y una revisión de las primitivas para concurrencia, a diferencia de la versión anterior *1.4*.

### 3.5.1. Los hilos en Java

El lenguaje Java proporciona un sencillo API [OW04] para manejar los hilos. Este API consta de una veintena de métodos(a diferencia de los hilos POSIX, como hemos visto anteriormente)

Estos hilos se representan mediante una clase, la clase *Thread*, esta clase se encuentra en el paquete *java.lang.Thread*. Para manejar los hilos en java basta con utilizar los métodos de esta clase y algún que otro método de la clase *Object*.

Nada más empezar un programa en Java, existe un hilo de ejecución que es el indicado por el método *main*, el hilo principal. Si no se crean más hilos el hilo va recorriendo los diferentes métodos y objetos de que consta el programa. Sin embargo, desde el hilo principal se pueden ir creando otros hilos de ejecución. Los hilos pueden estar ejecutando código en diferentes objetos, diferente código en el mismo objeto o incluso mismo código en el mismo objeto y al mismo tiempo. Se plantea pues la diferencia entre Objeto e Hilo.

Un *objeto* es algo estático, con sus atributos y métodos, quién realmente ejecuta esos métodos es el hilo de ejecución. En ausencia de hilos sólo hay un hilo que va recorriendo los objetos

**Creación de hilos**

Existen 2 formas de crear hilos

- Heredando de la clase Thread y redefiniendo el método run
- Implementando la interfaz Runnable

**Heredando de la clase Thread y redefiniendo el método run**

La clase Thread tiene un método especial, si queremos crear una clase cuyas instancias sean hilos debemos de heredar de la clase Thread y redefinir el método *run*. Dentro de este método describiremos lo que queremos que ejecute el hilo.[Ejemplito].Para poner este hilo en ejecución se llama al método **`start()`**.Este método pertenece a la clase *Thread* y se encarga de algunas inicializaciones propias de los hilos y llama al método *run()*.

**Implementando la interfaz *Runnable***

La interfaz *Runnable* sólo tiene un método con la signatura *public void run()*, este método es el que debemos implementar. Al contrario que antes, no hemos heredado de la clase *Thread* por lo que nuestros objetos no serán hilos. Si queremos que se ejecuten como hilos deberemos crearnos un objeto de la clase Thread y pasarle como argumentos el objeto donde queremos que empiece su ejecución ese hilo. Posteriormente se invoca al mátodo *start()* y ya se encarga este de invocar al método *run()*.

Java además incluye una librería para programas concurrentes, donde se implementan los tipos más comunes de sincronización(cerrojos, barreras,...)

## 3.6. Conclusiones

Como se ha comentado, la simulación es una herramienta muy potente para el estudio de las arquitecturas de computadores. Nos permite evitar los elevados costes que supondrian los cambios de topología de cualquier sistema, y se convierte en la principal herramienta para la investigación. No es casualidad que los principales simuladores de arquitecturas se desarrollen en universidades, por lo que suelen dejar de mantenerse.

Aqui se ha intentado dar una visión general del estado de la investigación sobre simuladores del mismo tipo que este proyecto, incluyendo en todo lo posible los simuladores gratuitos y de fácil manejo, pero por mucho indagar en el tema no se ha encontrado un simulador que utilice el paralelismo para acelerar la simulación.

# Capítulo 4

# Los Multicomputadores

Antes de situarnos en el análisis de nuestro problema, se ha considerado conveniente hacer un estudio de la arquitectura multicomputador. Como se ha descrito en los conceptos previos, los multicomputadores son arquitecturas de memoria distribuida, es decir, tienen una memoria local por cada procesador y cuando los procesadores necesitan acceder a la información de otro procesador, o enviar datos, se comunican enviando **mensajes**. Se puede definir entonces a los multicomputadores como multiprocesadores de paso de mensajes.
Este capítulo ha sido extraído de [Par01]

## 4.1. Introducción

Los multicomputadores son un tipo especial de sistemas con múltiples procesadores. Las características que les distinguen son:

1. La memoria es privada. Cada procesador tiene un mapa de direcciones propio que no es accesible directamente a los demás.
2. La comunicación entre procesadores es por paso de mensajes a través de una red de interconexión.

- cada nodo es una computadora clásica.
- Los nodos colaboran para resolver juntos un mismo problema (ejecutar la misma aplicación).
- La compartición de datos es explícita, ya que el acceso a datos comunes es por paso de mensajes.

En la figura 4.1 podemos observar los diferentes tipos de multicomputadores y sus características.

| Generación | 1ª | 2ª | 3ª | Blue Gene |
|---|---|---|---|---|
| Año | 1983-87 | 1988-92 | 1993-97 | 2005 |
| **NODO** | | | | |
| MIPS | 1 | 10 | 100 | 1334 |
| MFLOPS escalares | 0.1 | 2 | 40 | 700 |
| MFLOPS vectoriales | 10 | 40 | 200 | NA |
| Memoria (Mbytes) | 0.5 | 4 | 32 | 256 |
| **SISTEMA TÍPICO** | | | | |
| Nodos | 64 | 256 | 1024 | 65536 |
| MIPS | 64 | 2560 | 100K | NA |
| MFLOPS escalares | 6.4 | 512 | 40K | 360000K |
| MFLOPS vectoriales | 640 | 10K | 200K | NA |
| Memoria (Mbytes) | 32 | 1K | 32K | 16000K |
| **COMUNICACIONES** | | | | |
| Latencia (mensaje de 100 bytes) | | | | |
| Vecinos (microseg) | 2000 | 5 | 0.5 | NA |
| No locales (microseg) | 6000 | 5 | 0.5 | NA |

Figura 4.1: Características y tipos de Multicomputadores

La arquitectura de paso de mensajes presenta ciertas ventajas respecto a los multiprocesadores, y es mucho mejor cuando el número de procesadores es grande, y como veremos el número de canales entre nodos suele oscilar entre 4 y 8.
Pero su principal virtud es que esta arquitectura es directamente escalable y presenta un bajo coste para sistemas grandes.
El tamaño de un proceso puede describirse por su **granularidad**:

$$Granularidad = TiempodeCalculo/TiempodeComunicacion$$

1. Ganularidad gruesa: gran número de instrucciones secuenciales que tardan un tiempo sustancial en ejecutarse.

2. Granularidad media: intermedio entre fina y gruesa.

3. Granularidad fina: pocas instrucciones, incluso una instrucción.

Al reducirse la granularidad, la sobrecarga de los procesos suele aumentar, es por ello que en este tipo de máquinas la granularidad empleada suele ser media o gruesa.
Las operaciones de encaminamiento de los mensajes suelen estar soportadas por hardware lo que reduce la latencia de las comunicaciones.

Pero esta arquitectura también presenta algunas desventajas:

- Sobrecarga: El código y los datos deben transferirse a la memoria local de cada nodo antes de su ejecución lo que puede suponer una sobrecarga importante. Los resultados deben trabsferirse al computador anfitrión.

Los cálculos a realizar deben ser lo suficientemente largos para compensar este inconveniente.

**El programa a ejecutar deber ser INTENSIVO EN CÁLCULO, no en operaciones de entrada/salida o de paso de mensajes**

La comunicación por paso de mensajes se realiza a través de una red de interconexión entre los diferentes nodos, esta red juega un papel relevante en el rendimiento de los multicomputadores.

## 4.2. Redes de interconexión

Las redes más extendidas para la comunicación entre multicomputadores son las redes *directas o estáticas*. En este tipo de redes los enlaces son *directos* y son fijos una vez construida la red (Conexiones estáticas).

Para entender la importancia de la red en la comunicación se definen los siguientes conceptos:

- Grado del nodo : Es el número de canales que conectan un nodo con el vecino.
- Diámetro de la red: Es la máxima distancia mínima entre dos nodos de una red.
- Regularidad: Una red es regular si todos los nodos tienen el mismo grado.
- Simétrica: Una red es simétrica si se ve igual desde cualquier nodo.

Como se observa en la figura 4.2 las redes directas basadas en encaminador(Router) son las mallas, los toros, los hipercubos, etc ... Con el encaminamiento hardware(ej. Whormhole Routing que veremos más tarde), el diámetro de la red ya no es un problema ya que el retraso en la comunicación entre dos nodos cualesquiera se ha convertido casi constante con un alto nivel de segmentación.

En definitiva, con la utilización de técnicas segmentadas en el encaminamiento, la reducción del diámetro de la red ya no es un objetivo primordial. La facilidad de encaminamiento, la escalabilidad y la facilidad para ampliar el sistema son actualmente temas más importantes, por lo que se imponen actualmente las redes estrictamente ortogonales en los multicomputadores actuales.

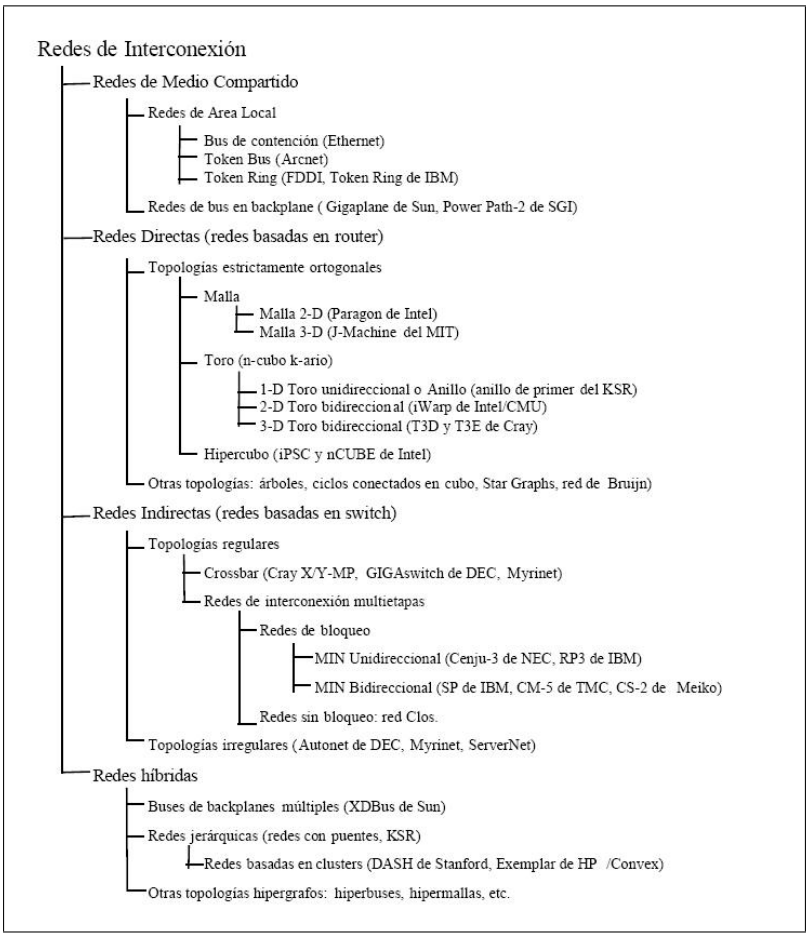


Figura 4.2: Clasificación de las redes de interconexión

## 4.3. La capa de conmutación(switching)

Los multicomputadores necesitan pasarse continuamente información. Esa información debe atravesar la red siguiendo un camino fijado por los encaminadores. Las técnicas de conmutación son los procedimientos que se implementan dentro de los encaminadores(routers) para realizar el mecanismo por el cual los mensajes pasan a través de la red. En resumen, consiste en como y cuando se conectan los conmutadores internos del router para conectar las entradas con las salidas, asi como cuando los componentes del mensaje pueden transferirse a través de esos caminos. Estas técnicas estan ligadas a los mecanismos de **control de flujo**.

El control de fujo sincroniza la transferencia de información entre encaminadores(y a través de estos) durante el proceso de envío de los mensajes a través de la red.
Este control de flujo está a su vez fuertemente acoplado al manejo de **buffers** que determinan como se asignan y liberan los buffers, determinando como resultado como se manejan los mensajes cuando se bloquean en la red.

### 4.3.1. El control de flujo

El control de fujo es un protocolo asíncrono para transferir y escribir una unidad de información. La unidad del control de flujo es el **flit**. El flit es la menos unidad de información cuya transferencia es solicitada por el emisor y notificada por el receptor. Es un mecanismo *request/acknowledgement* para asegurar una transferencia exitosa y la disponibilidad de espacio en el buffer. Se trata de transferencias *atómicas*, debe asegurarse un espacio suficiente para asegurar que el paquete se tranfiere en su totalidad.

El control de flujo ocurre a dos niveles, el control de flujo del mensaje ocurre a nivel de paquete, pero la transferencia del paquete a nivel físico se realiza en varios pasos. Por ejemplo si se desea enviar un paquete de 128 bytes por un canal de 16 bits, este envío necesitará de varios ciclos. La transferencia resultante multiciclo usa un control de flujo del canal para enviar un flit a través de la conexión física.

En general, cada mensaje puede dividirse en paquetes de tamaño fijo. Estos paquetes son a su vez divididos en unidades de control de flujo o flits. Debido a la anchura del canal puede ser necesario varios ciclos de canal para transferir un único flit.
Un **phit** es la unidad de información que puede transmitir a través de un canal físico en un único paso o ciclo, es decir, es el número de bits que se pueden trasnferir en paralelo en un único ciclo.

Las transferencias inter-encaminador deben realizarse necesariamente en términos de phits, mientras que las técnicas de conmutación manejan flits, y manejan el conmutador interno para conectar el buffer de entrada con los buffers de salida, y enviar los flits a través de este camino.

### 4.3.2. El encaminador

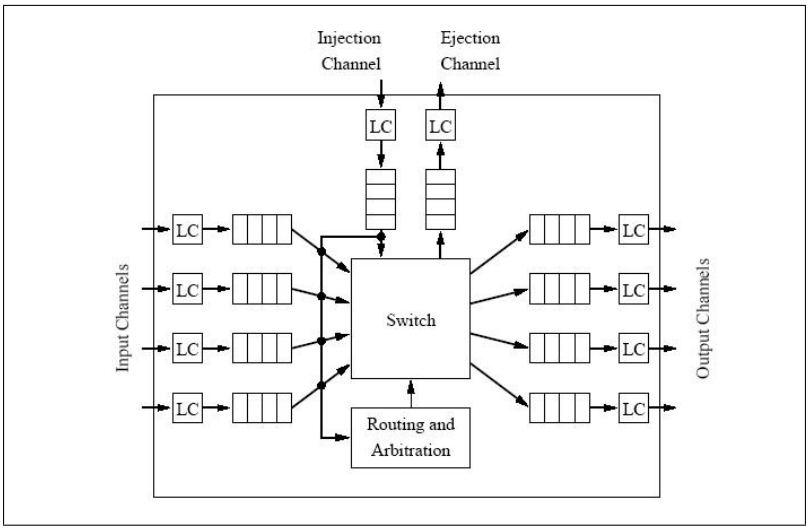


Figura 4.3: Modelo del encaminador (LC = Link controller)

La arquitectura básica de un encaminador se muestra en la figura 4.3. Esta arquitectura esta compuesta por los siguientes elementos:

- **Buffers** Son Buffers FIFO, un buffer esta asociado con cada canal físico de E/S.
- **Conmutador** Es el responsable de conectar los buffers de entrada con los buffers de salida.
- **Unidad de encaminamiento y arbitraje** Implementa los algoritmos de encaminamiento, selecciona el enlace de salida para un mensaje entrante, programando el conmutador en función de la elección. El arbitraje se encarga de solucionar los problemas de contención.
- **Controladoras de enlace (LLC)** El flujo del mensaje a través de los canales físicos entre encaminadores adyacentes se implementa mediante el LC. Sincroniza la info para transferir flits.
- **Interfaz del procesador** Consiste en uno o más canales de inyección desde el procesador y

Cuando un mensaje llega a un encaminador, este debe ser examinado para determinar el canal de salida por el cual debe ser enviado este mensaje para llegar a su destino. A este retraso se le denomina *retraso de encaminamiento* ($t_r$) y suele incluir el tiempo para configurar el conmutador.

Una vez se ha establecido un camino a través del encaminador, nos interesa saber la velocidad a la cual podemos enviar mensajes a través del conmutador. Esta velocidad viene determinada por el retraso de propagación a través del conmutador, y el retraso que permite la sincronización de la transferencia de datos entre los buffers de entrada y de salida. Es lo que se denomina la *latencia del control de flujo interno* ($t_s$). De manera similar, al retraso a través de los enlaces físicos(inter-encaminadores) se le denomina l*atencia de control del flujo externo* ($t_w$)

El retraso debido al encaminamiento y los retrasos de control del flujo determinan la latencia disponible a través del conmutador y junto con la contención de los mensajes en los enlaces determinan el rendimiento o **productividad de la red (throughput)**

## 4.4. Técnicas de conmutación

En este apartado veremos las principales técnicas de conmutación. Para ello haremos las siguientes suposiciones:

$1phit = 1flit$ determinan el $anchodebandadelcanal = Wbits$
$mensaje = Lbits$
$cabecera = 1flit$
$tam.mensaje = W + L$
$t_r$: tiempo de decisión de encaminamiento de un router.
El canal físico entre 2 encaminadores opera a $BHz$ por lo que el
$anchodebandadelcanal = BWbits/s$
$t_w$: retraso de propagación a través del canal $t_w = 1/B$
$t_s$: retraso de conmutación

### 4.4.1. Conmutación de circuitos

Esta técnica reserva un camino físico desde el origen hasta el destino antes de producirse la transmisión de los datos. Para ello inyecta un flit de cabecera en la red y este avanza hacia el destino reservando los enlaces físicos conforme se va transmitiendo de los encaminadores intermedios. Cuando alcanza el destino se habrá establecido un camino, enviándose una señal de asentimiento de vuelta al origen. La ecuación que describe el tiempo que se consume en transmitir el mensaje es la siguiente:

$$D = numerodeenlaces$$

$$L = longituddelmensaje$$
$$1paquete = L + Wbits$$

Tenemos que multiplicar por 2 porque primero se realiza el encaminamiento y después el envío.

$$t_{prep} = D(2t_w + 2t_s + t_r)$$
$$t_{datos} = \lceil L/W \rceil \, 1/B$$
EL tiempo total de esta técnica es:
$$t_{circuit} = t_{prep} + t_{datos}$$

Este mecanismo va a ser interesante para mensajes largos e infrecuentes, es decir, cuando menor sea el tiempo de setup en comparación con el tiempo de datos, mejor será esta técnica. Será una buena opción para aplicaciones de grano grueso.

Tiene el defecto que es bastante sensible a los bloqueos. En cambio, la cabecera puede detectar averías en la red, lo que supone una buena Tolerancia a fallos.

### 4.4.2. Conmutación de paquetes

En esta técnica el mensaje se divide y se transmite en paquetes de longitud fija. Los primeros bytes del paquete contendrán la información de encaminamiento. Un paquete se almacena completamente en cada nodo intermedio antes de ser enviado al nodo siguiente(*Store and Forward(SAF)*) El tiempo de esta técnica es:

$$t = D(t_r + (t_s + t_w) \lceil (L + W)/W \rceil)$$

Esta ténica es ventajosa para mensajes cortos y frecuentes.

### 4.4.3. Conmutación de paso a través virtual, *Virtual Cut-Through(VCT)*

Aquí, en lugar de esperar a recibir el paquete en su totalidad, la cabecera del paquete puede ser examinada tan pronto como llega. El encaminador puede comenzar a enviar la cabecera y los datos que le siguen tan pronto como se realice una decisión de encaminamiento y el buffer de salida esté libre.

El mensaje no tiene ni porque ser almacenado en la salida y puede ir directamente a la entrada del siguiente encaminador antes de que el paquete completo se haya recibido en el encaminador actual. Si la cabecera se bloquea en el canal de salida ocupado, la totalidad del mensaje se almacena en el

nodo. Como se observa, para altas cargas, esta técnica se equipara a la conmutación por paquete.

$$t_{VCT} = D(t_r + t_s + t_w) + max(t_s, t_w)\lceil L/W \rceil$$

Únicamente la cabecera experimenta el retraso del encaminador, al igual que el retraso en la conmutación y en los enlaces de cada encaminador, esto es debido a que la transmisión esta segmentada.

### 4.4.4. Conmutación de lombriz (*Wormhole*)

Un paquete se divide en flits. El flit es la unidad de control de flujo de los mensajes. Los buffers de E/S pueden almacenar algunos flits. El mensaje se segmenta dentro de la red a nivel de flits y normalmente es demasiado grande para que pueda ser totalmente almacenado dentro de un buffer. En un instante dado, un mensaje bloqueado ocupa buffers en varios encaminadores.

En ausencia de bloqueos, los paquetes de mensaje se segmentan a lo largo de la red, si el canal de salida esta ocupado, el mensaje se bloquea *in situ*.

$$t_{wormhole} = D(t_r + t_s + t_w) + max(t_s, t_w)\lceil L/W \rceil$$

### 4.4.5. Conmutación cartero loco (*Mad Postman*)

Es un caso particular del wormhole suponiendo que los flits y los phits no coinciden (ej: phit→ 1 bit, flit → 1 byte)
Cuando el flit cabecera empieza a llegar a un encaminador, se supone que el mensaje continuará a lo largo de la misma dimensión, por tanto, los bits cabecera se envían hacia el enlace de salida de la misma dimensión tan pronto como se reciben(suponiendo que esta libre el canal). Cada bit de la cabecera también se almacena localmente. Una vez se ha recibido el último bit del primer flit de la cabecera, el encaminador puede examinar este flit y determinar si el mensaje debe continuar a lo largo de esta dimensión.

En esencia, el mensaje primero se envía a un canal de salida y posteriormente después se comprueba la dirección. El primero se elimina del mensaje, pero continúa atravesando la primera dimensión, es lo que se denomina **flits muertos**.

Esta técnica tiene el inconveniente que los flits sueltos producen bloqueos, para solucionar esto se desarrollan mecanismos para eliminarlos.

$$t_h = (t_s + t_w)D + max(t_s, t_w)W$$
$$t_{datos} = max(t_s, t_w)L$$
$$t_{madpostman} = t_h + t_{datos}$$

### 4.4.6. Los Canales Virtuales

Un canal físico puede soportar varios **canales virtuales** o lógicos multiplexados sobre el mismo canal físico. El protocolo del canal físico debe ser capaz de distinguir entre los canales virtuales que usan el canal físico.

Los canales virtuales fueron originariamente introducidos para resolver el problema de bloqueos en las redes con conmutación segmentada. El mecanismo consiste en añadir buffers a cada canal físico, es decir, si tenemos un canal físico, para tener canales virtuales pondremos un buffer para cada canal virtual.
El **bloqueo** en una red ocurre cuando los mensajes no pueden avanzar debido a que cada mensaje necesita un canal ocupado por otro mensaje.

Cada canal virtual adicional mejora el rendimiento en una cantidad menor, y el incremento de la multiplexación de canales reduce la velocidad de transferencia de los mensajes individuales, incrementando la latencia del mensaje. Además, los canales virtuales aumentan la complejidad del conmutador, que debe manejar más E/S.

### 4.4.7. Mecanismos híbridos de conmutación

#### Conmutación encauzada de circuitos

En este mecanismo, la cabecera establece el camino, luego se envía hacia atrás el reconocimiento y luego se envía el paquete. Podemos estar estableciendo caminos antes de terminar de enviar los datos. Un canal de comunicaciones $V_i$ se divide en:

$$v_i^d : datos$$
$$v_i^c : correspondencia(cab)$$
$$v_i^* : complementario(ack)$$

Permite la tolerancia a fallos, pero presenta un problema.Comparando con el de lombriz, este tarda más ya que tiene que esperar el reconocimiento(mayor latencia).

#### Conmutación de exploración

En esta ocasión se envía la cabecera por delante, pero en vez de esperar que llegue a destino y mande el reconocimiento, se deja una distancia entre la cabecera y el paquete que llega detrás, es decir, unos nodos o enlaces después se manda el paquete. Cuanto más largo es ese espacio, más capacidad de tolerancia, y cuanto menor tendremos menor latencia y mayor rendimiento.

Cada vez que la cabecera llega a un nodo envía un *ack* hacia atrás. Cuando llega a destino la cabecera envía *acks* de forma continuada, estos llevan la información del camino. Los canales van a tener un contador asociado (un contador por canal). Cuando pasa la cabecera pone el contador a 0. Cada nodo aumenta el contador cada vez que pasa un *ack*, si ese nodo recibe 2 *acks* quiere decir que la cabecera va 2 nodos por delante.

Si tenemos k enlaces, el paquete que viene lee el contador y si es $\geq$k sigue adelante. Por contra si es ¡se espera hasta recibir los *acks* necesarios para seguir avanzando.
Tenemos 2 situaciones extremas:

$$\text{k= 0} \Leftrightarrow \text{lombriz}$$
$$\text{k = diametro de la red} \Leftrightarrow \text{conmutación encauzada de circuitos}$$

El tiempo depende de k (el número de enlaces que vayan a conectar). Como el reconocimiento tiene que ir k enlaces atrás, y para que el paquete llegue, su cabecera tiene que ir k enlaces delante con lo que tenemos 2k.

$$t_{prep} = D(t_r + t_s + t_w)$$
$$t_{datos} = max(t_s, t_w(\lceil L/W \rceil - 1)$$
$$t_{ss} = t_{prep} + (t_s + t_w)(2k - 1) + t_{datos})$$

### 4.4.8. Conclusiones sobre las técnicas de conmutación

Como se acaba de ver, existen muchas técnicas de conmutación. Al añadir canales virtuales el rendimiento de la red también aumenta. Cuando la capacidad de almacenamiento por canal físico es la misma en VCT que en Lombriz, el mecanismo de la lombriz con canales virtuales consigue un mayor rendimiento de la red.

La lombriz es capaz de conseguir latencias y rendimiento comparables a los del VCT en el caso de existir suficientes canales virtuales y teniendo una capacidad de buffer similar. Se antoja pues el mecanismo Wormhole como una técnica bastante interesante.

A continuación se verá como se deciden los caminos a seguir por el los paquetes para llegar a destino, es decir como decide el encaminador por dónde enviar el paquete.

## 4.5. La capa de encaminamiento (routing)

Los encaminadores o routers se encargan de decidir que camino va a seguir cada mensaje o paquete. Para ello se basan en diferentes algorítmos de encaminamiento. Estos algorítmos tienen las siguientes propiedades:

- Conectividad. Es la habilidad de encaminar paquetes desde cualquier nodo origen a cualquier nodo destino.
- Adaptabilidad. Es la habilidad de encaminar paquetes a través de caminos alternativos en presencia de contención o componentes defectuosos.
- Libre de Bloqueos(deadlock y livelock). que no presente bloqueos, ni que se quede en espera indefinida.
- Tolerancia a fallos. que pueda encaminar en presencia de fallos.

### 4.5.1. Clasificación de los algorítmos de encaminamiento

Los diferentes procedimientos que puede seguir un encaminador se pueden clasificar en base a diferentes criterios. Claro esta que cada uno es libre de realizar su propia ordenación. Aqui se presentan algunos criterios de clasificación:

- Encaminamiento centralizado. Un controlador (el nodo origen) es el encargado de realizar los cálculos, se trata de un encaminamiento desde el origen.
- Encaminamiento distribuído. A medida que va atravesando la red se va procediendo el cálculo del camino.
- Deterministas. Siempre suministran el mismo camino entre origen y destino.
- Adaptativos. Usan la información del tráfico y/o el estado de los canales para evitar la congestión.
- Aprovechables(mínimos) Sólo utilizan caminos que acerquen al destino.
- Dendiendo del número de caminos alternativos. Totalmente adaptativos o parcialmente adaptativos.

### 4.5.2. Los Bloqueos

Al permitir que un paquete cuya cabecera no ha llegado a su destino solicite buffers adicionales al mismo tiempo que mantiene los buffers que almacenan actualmente el paquete puede surgir una situación de **bloqueo**.

Un **bloqueo mortal(deadlock)** ocurre cuando algunos paquetes no pueden avanzar hacia su destino ya que los buffers que solicitan están ocupados. Todos los paquetes involucrados en una configuración de bloqueo

mortal están bloqueados para siempre.

Un paquete puede estar viajando alrededor del nodo destino sin llegar nunca a alcanzarlo porque los canales que necesita están ocupados por otros paquetes. A esta situación se le conoce con el nombre de **bloqueo activo (livelock)** y sólo puede ocurrir en el caso de que los paquetes puedan seguir caminos no mínimos. Por tanto, una solución para evitar este tipo de incidencias es no permitir usar caminos no mínimos.

Un paquete puede permanecer completamente parado si el tráfico es intenso y los recursos solicitados son asignados siempre a otros paquetes que los solicitan. Esta situación es conocida como **muerte por inanición(starvation)**. Suele deberse a una asiganción incorrecta de los recursos en el caso de conflicto entre 2 o más paquetes. Una solución a este problema es la asignación de recursos basada en una cola circular.

La principal motivación de usar caminos no mínimos es la *Tolerancia a Fallos* y además evitar el deadlock mediante el encaminamiento usando desviación.

#### Solución al deadlock

- prevención del bloqueo. Los recursos (canales o buffers) son asignados a un paquete de tal manera que una petición nunca da lugar a una situación de bloqueo. Para ello se reservan todos los recursos necesarios antes de empezar la transmisión del paquete.
- evitar el deadlock. Los recursos son asignados a un paquete al tiempo que se avanza a través de la red. Sin embargo, un recurso se le asigna a un paquete únicamente si el estado global resultante es seguro.
- detección del bloqueo. Se liberan algunos recursos que son reasignados a otros paquetes.

### 4.5.3. Algoritmos deterministas

Estos procedimientos establecen el camino como una función de la dirección destino, proporcionando siempre el mismo camino entre cada para de nodos.

#### Encaminamiento por orden de la dimensión

Es un algoritmo que se utiliza en hipercubos, mallas y toros. Se trata de calcular la distancia entre el nodo actual y el nodo destino como una suma de las diferencias de posiciones entre todas las dimensiones. El algoritmo más

simple consiste en reducir una de estas diferencias a cero antes de considerar la siguiente dimensión → **Encaminamiento por dimensiones**.
Para mallas 2D se trata del encaminamiento XY ( figura 4.4)

**Algorithm: XY Routing for 2-D Meshes**
**Inputs:** Coordinates of current node $(Xcurrent, Ycurrent)$
and destination node $(Xdest, Ydest)$
**Output**: Selected output $Channel$
**Procedure:**
$Xoffset := Xdest - Xcurrent;$
$Yoffset := Ydest - Ycurrent;$
**if** $Xoffset < 0$ **then**
$Channel := X-;$
**endif**
**if** $Xoffset > 0$ **then**
$Channel := X+;$
**endif**
**if** $Xoffset = 0$ **and** $Yoffset < 0$ **then**
$Channel := Y-;$
**endif**
**if** $Xoffset = 0$ **and** $Yoffset > 0$ **then**
$Channel := Y+;$
**endif**
**if** $Xoffset = 0$ **and** $Yoffset = 0$ **then**
$Channel := Internal;$
**endif**

Figura 4.4: Algoritmo XY para mallas 2D

- Bloqueos en toros Los toros tienen ciclos. Si no es posible conseguir un grafo acíclico sin que la función deje de ser conexa, se procede a añadir arcos al diámetro D, asignando a cada canal físico un conjunto de canales virtuales. Cada canal físico $C_i$ se divide en dos canales $C_{0i}$ y $C_{1i}$ El algorítmo se muestra en la figura 4.5

### 4.5.4. Algoritmos parcialmente adaptativos

Se definen los algoritmos parcialmente adaptativos como algoritmos con la libertad de elegir caminos pero dentro de un subconjunto de caminos. Si estos algoritmos son totalmente adaptativos su libertad de elección es total, pueden utilizar cualquier camino mínimo entre origen y destino.

El objetivo de estos algoritmos es maximizar el rendimiento de la red (throughput).

**Algorithm: Dimension-Order Routing for Unidirectional 2-D Tori**
**Inputs:** Coordinates of current node $(Xcurrent, Ycurrent)$
and destination node $(Xdest, Ydest)$
**Output:** Selected output $Channel$
**Procedure:**
$Xoffset := Xdest - Xcurrent;$
$Yoffset := Ydest - Ycurrent;$
**if** $Xoffset < 0$ **then**
$Channel := c_{00};$
**endif**
**if** $Xoffset > 0$ **then**
$Channel := c_{01};$
**endif**
**if** $Xoffset = 0$ **and** $Yoffset < 0$ **then**
$Channel := c_{10};$
**endif**
**if** $Xoffset = 0$ **and** $Yoffset > 0$ **then**
$Channel := c_{11};$
**endif**
**if** $Xoffset = 0$ **and** $Yoffset = 0$ **then**
$Channel := Internal;$
**endif**

Figura 4.5: Algoritmo XY para toros 2D unidireccionales

#### Encaminamiento adaptativo por planos

En cada plano podemos elegir cualquier camino mínimo pero con una restricción: sólo vamos a poder movernos por planos. El encaminamiento es totalmente adaptativo pero en cada plano. En este encaminamiento son necesarios 3 canales virtuales por canal físico para evitar bloqueos en mallas y 6 canales virtuales para evitar bloqueos en toros.

### 4.5.5. Algoritmos completamente adaptativos

Estos algortimos están siempre libre de bloqueos. Permiten elegir cualquier camino mínimo. Como era de esperar, no todo son ventajas, estos algortimos consumen gran cantidad de recursos por lo que resultan caros.

El primer mecanismo que se utiliza es el algoritmo basado en clases. Se trata de añadir tantos canales virtuales como sean necesarios para obtener una red virtual totalmente adaptada. A cada nodo que atravieso utilizo un canal virtual superior.

#### Algoritmo completamente adaptativo dinámico

En esta ocasión se trata con una subred con canales deterministas y canales virtuales. Encaminamos los paquetes por los canales virtuales adaptativos, el paquete no elige el canal determinista, viaja siempre por los canales virtuales, eligiendo cualquiera de estos. En el caso de que un determinado

paquete sea susceptible de bloqueo, este irá a parar al canal determinista, y seguirá las leyes de este canal, y ya no saldrá del canal determinista.

### 4.5.6. Protocolo de Duato

Este método debe su nombre al Dr. José Duato [DYN97]. Se define una metodología para obtener mecanismos de encaminamiento basada en el siguiente teorema:
Resumiendo:

Según Duato, puedo añadir otros canales (virtuales) de manera que puedo encaminar un paquete que se encuentra en X, o bien excogiendo otro de los canales que me llevan de X a Y, o bien excogiendo el determinista.

Formulando:
si tenemos una red de interconexión $I_1$ y tenemos una función de encaminamiento $R1$ sin bloqueos mortales. Esta red permite conectar cualquier nodo de la red.
C1: Conjunto de canales de R1
C: Conjunto de canales virtuales de la Red
$C_{xy}$: cojunto de canales de salida de x que conectan x e y.

Podemos definir una función de encaminamiento R(x, y):

$$R(x, y) = R1(x, y) \cup (C_{xy} \cap (C - C1))$$

tenemos un conjunto de canales deterministas que estan libre de bloqueos.

Ej. Para aplicar este método al algorítmo de Lombriz primero hay que comprobar el grafo de dependencias para comprobar si hay ciclos.

### 4.5.7. Algunas máquinas comerciales

Por último, vamos a citar brevemente algunas de las máquinas comerciales que tienen la arquitectura de un multicomputador.Las máquinas de paso de mensajes más populares se realizaron por Intel, con dos arquitecturas sucesivas: los iPSC/l y los iPSC/2. Son máquinas de grano bastante grueso, que tienden a alcanzar los rendimientos de las supercomputadoras, utilizando tecnologías mucho menos agresivas, y por tanto con un coste menor.

El otro ejemplo típico de multicomputador es el construido a partir del Transputer, un microprocesador diseñado especialmente para computación paralela. Un Transputer es un microprocesador que reúne en un solo chip

un procesador, una memoria local y canales de comunicación que proporcionan una conexión punto-a-punto con otros Transputers. La arquitectura de la CPU es de tipo RISC. El control complejo se realiza por microcódigo. En particular, cada Transputer puede soportar varios procesos que se ejecutan concurrentemente. El ordenamiento de los procesos se lleva a cabo enteramente por microcódigo, lo que permite realizar una conmutación de contexto muy rápido (alrededor de 1 $\mu$s). El Transputer que más difusión ha tenido hasta el momento es el T800.

Para posibilitar las redes de comunicación de alta velocidad, hace pocos años se ha desarrollado la segunda generación de transputers, denominado T9000. Este nuevo transputers tiene un procesador superescalar, un planificador de tareas hardware, 16 Kbytes de memoria caché en el propio chip y un procesador de comunicaciones autónomo. Sobre todo a nivel de la Comunidad Económica Europea, el diseño de máquinas basadas en el Transputer es muy importante.

IBM tiene en marcha en el proyecto BlueGene[IBM05] (figura 4.6), toda una saga de máquinas que, en unos pocos años, alcanzarán la potencia de cálculo del PetaFLOP/s tomada como objetivo. La primera de estas máquinas, prevista para 2005, es el BlueGene/L (BG/L para abreviar). La información disponible sobre el BG/L es, de momento, escasa. No sabemos con detalle, por lo tanto, cómo está diseñada esta máquina, pero sí vamos sabiendo cosas, como que la red de interconexión es un toro 3D, con encaminamiento adaptativo, y que usa la técnica de canales de escape con control de flujo por burbuja para evitar interbloqueos.

La información sobre Red Storm (figura 4.7), sistema que está siendo construido por Cray[Cra05], es aún menor. En ella se indica que se va a optar por una interconexión específicamente diseñada para esta máquina, aunque se considera la posibilidad de recurrir a la tecnología QsNet de Quadrics (con modificaciones) si es necesario. En cuanto a la topología, optan por una malla 3D.

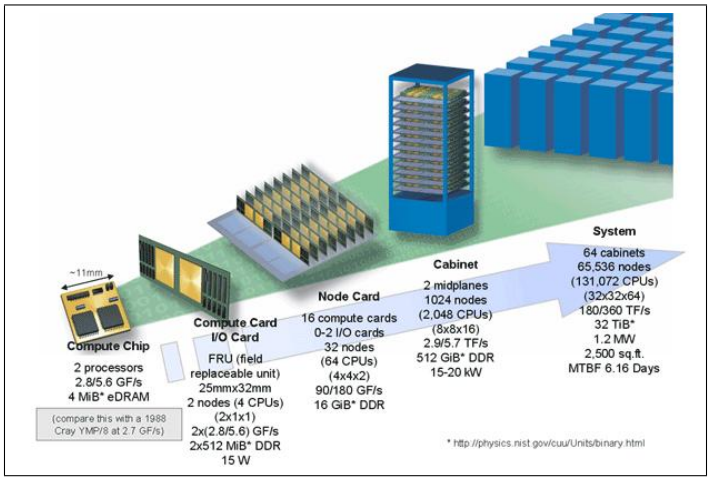


Figura 4.6: Esquema del BlueGene/L

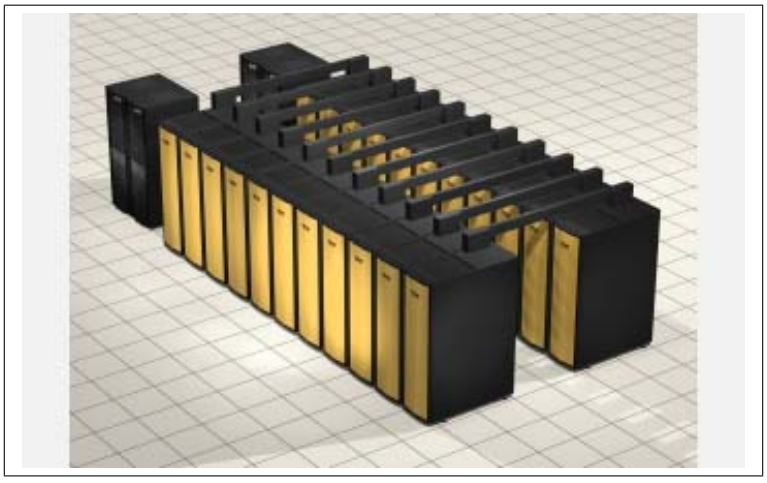

Figura 4.7: Renderización de ejemplo del Red Storm de Cray

# Parte II

# Análisis, Diseño e Implementación

# Capítulo 5

# Análisis

## 5.1. Introducción

Las aplicaciones escritas para sistemas de procesamiento paralelo son programas que requieren, para su ejecución, equipos dotados de N procesadores, de modo que en cada uno de ellos pueda correr simultáneamente un proceso creado en tiempo de ejecución, compartiendo eventualmente entre todos alguna zona de la memoria, con la esperanza de que el tiempo total que le tome correr a la aplicación completa se reduzca una cantidad de veces comprendida entre 1 y N con relación al que le tomaría correr en un equipo secuencial monoprocesador.

Un programa redactado para ser procesado en máquina paralela incluirá sentencias especiales (las que le darán precisamente el carácter paralelo), desde las encargadas de la creación de procesos y el lanzamiento de los mismos en distintas unidades de procesamiento, hasta los pedidos de acceso a zonas comunes, los pedidos de bloqueo y liberación de las zonas comunes, etc., insertadas dentro de código de programación convencional (no paralelo).

## 5.2. ¿Qué se puede paralelizar?

Esta es una pregunta que comúnmente se hace un programador con cualquier programa, y la respuesta no es muy sencilla, dado que debe analizarse si se cuenta con el hardware y software necesario y determinar si vale la pena paralelizar el programa o viceversa. En cuestiones de equipamiento el proceso de paralelizar un programa involucra, o mejor dicho, exige al programador conocer un poco más de la arquitectura de la supercomputadora o 'cluster' sobre el cual pretende paralelizar su código, conocer el número de procesadores con los que se cuenta, la cantidad de memoria, espacio en disco, los niveles de memoria disponible, el medio de interconexión, etc.

En términos de software el usuario debe conocer qué sistema operativo esta manejando, si los compiladores instalados permiten realizar aplicaciones con paralelismo, si se cuenta con herramientas como PVM o MPI (en Sistemas Distribuidos), Power C, Power Fortran u OpenMP (en sistemas de memoria compartida), etc. Para evaluar si vale la pena implementar paralelismo a un código, podemos identificar nuestra situación con las siguientes justificaciones:

1. Necesidad de respuesta inmediata de resultados. Si el usuario está experimentando con un programa en una máquina secuencial, y requiere ejecutarlo en varias ocasiones con diferentes datos de entrada, y cada tiempo de ejecución es considerable (consume horas o días) le es inapropiado o costoso esperar tanto tiempo para volver a realizar otro experimento o someterlos secuencialmente que al final disminuirá notablemente el desempeño de su máquina. Esto sin considerar las modificaciones al código o fallas en la ejecución del modelo. La paralelización y la ejecución del programa en una máquina paralela le permitirá realizar más análisis o experimentos en menos tiempo.

2. Es un problema de Gran Reto. Un programa puede presentar algoritmos de cálculo científico intensivo que demanden grandes recursos de cómputo (CPU, memoria, disco), en estos casos el dicho de "divide y vencerás"bien se puede aplicar, las tareas se dividen entre varios procesadores, se ejecutan en paralelo y se obtiene una mejora en la relación costo y desempeño. Hoy en día las arquitecturas de cómputo estan incorporando paralelismo en los más altos niveles de sus sistemas, para satisfacer las exigencias de los problemas de grandes retos.

3. Simplemente elegancia de programación. Es válido y se deja a voluntad del programador.

Para implementar paralelismo a un programa es muy importante que el código presente: 'Independencia de datos': El usuario debe identificar tareas independientes dentro del código, por ejemplo, revisar que existan ciclos for o do independientes, y rutinas o módulos independientes. De tal forma que no exista dependencias de datos que puedan obstruir la paralelización.

Que el programador identifique las zonas donde se efectúa *la mayor carga de trabajo* y que le consuma la *mayor parte de tiempo de ejecución*. Para eso existen herramientas que le permiten obtener una perspectiva o perfil de su programa.

## 5.3. Descripción de Simured C++

Como ya hemos visto, el Simured es un simulador de redes de computadores. Al estar escrito en Borland C++ existen 2 versiones, una para MS Windows y otra para Linux, además existe una versión no visual (por línea de comandos).

El simulador en sí se ha pensado que sea lo más portable posible y que compile con casi cualquier compilador de C++. El programa simured en realidad es un front-end para el módulo que hace la simulación. Las fuentes de la parte interna del simulador y un front-end de ejemplo se pueden decargar aquí mismo. Simured funciona tanto en Linux como en Windows.
La red que se simula es cualquier red estrictamente ortogonal toro n-cubo k-aria. En realidad también se pueden simular mallas pues la restricción de encaminamiento de las mallas respecto de los toros viene implementada en la propia función de encaminamiento.
El mecanismo de control de flujo es el Wormhole, si bien se está trabajando en otros mecanismos de conmutación segmentados. Mecanismos como conmutación de circuitos no son fácilmente implementables en el simulador, sin embargo el resto si.

Dispone de varias funciones de encaminamiento, deterministas, adaptativas, con bloqueos, etc. La implementación de nuevas funciones es relativamente simple.
Por el momento no se ha implementado la posibilidad de cambiar el arbitraje en el crossbar (actualmente el primero que llega es el que sale).

Permite modificar el número de canales virtuales de la red. Se puede decidir si estos canales son físicos o virtuales. También se pueden hacer bidireccionales o unidireccionales.

Permite modificar la longitud de las colas fifo y permitir o no adelantamiento (si la cola está vacía se pone el primero para salir).

Permite especificar los retrasos de las colas, de conmutación, de atravesar el crossbar, y el del canal.

Se puede modificar tanto la longitud del paquete como el de la cabecera.

Se pueden generar los paquetes de prueba a partir de un fichero de trazas. El formato del fichero de trazas es texto. En cada línea se especifica el lanzamiento de un paquete de manera que el primer campo de la línea es el ciclo en el que se lanza el paquete, el segundo campo el nodo origen y el tercero el nodo destino; hay un cuarto campo opcional por si se quiere

especificar una longitud para el paquete. Lo paquetes deben estar ordenados según el ciclo en el que se lancen. Los campos van separados por espacios. Los nodos se numeran empezando por la dimensión 0 donde se encuentran los nodos 0, 1, 2...

Si no se tiene un fichero de trazas se pueden lanzar paquetes de forma automática especificando el número de paquetes a lanzar o los flits por nodo, y también la productividad que se desea en Flits/Ciclo/Nodo.

La simulación puede ser interactiva. En este modo se muestran las dos primeras dimensiones de la red con sus colas y canales, de manera que se puede ver la evolución de los paquetes por la red. Se puede especificar un tiempo entre ciclo y ciclo para poder ver el movimiento más o menos rápido, también se puede detener la simulación y ver la evolución paso a paso.

En simulación no interactiva se realiza una simulación con los parámetros que se hubieran especificado.
La simulación también se puede realizar de forma múltiple para poder obtener resultados en función de la variación de uno o dos parámetros. En la simulación múltiple simple se modifica el valor de la productividad de un inicio a un final. Se puede elegir una escala logarítmica o lineal para este cambio. Una simulación múltiple más compleja consta de varias simulaciones simples como la anterior pero en las que se va modificando algún otro parámetro.
Los resultados de la simulación múltiple se pueden guardar en un fichero en formato texto (CSV) que puede ser leído directamente por cualquier hoja de cálculo.

El simulador dispone de la posibilidad de visualizar gráficamente los resultados de la simulación a partir de la lectura del fichero CSV generado.

Ahora que ya se ha visto lo que puede realizar este simulador vamos a introducirnos en su funcionamiento interno, que es el que nos servirá de base en este proyecto.

### 5.3.1. Descripción interna

La versión que se va a analizar internamente es la versión visual para Windows. Se ha decidido estudiar esta versión por ser modular. Nuestro interés se centra en la simulación en si, y como esta está implementada para poder 'transformar' en lo posible el código actual hacia una versión paralela.

El simulador esta divido en clases, lo que nos ayudará a entender el código y separar las más relevantes. Esta estructura de clases la podemos

observar a continuación, por un lado tenemos las clases que definen la red: Como se observa en la figura 5.1 los dispositivos *Channel*, *CrossSwitch*, *Fi-*

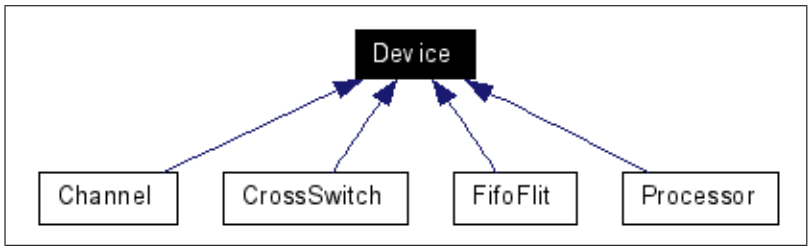


Figura 5.1: Clase Dispositivo

*foFlit* y *Processor* heredan de la clase *dispositivo*. Mediante la enumeración TypeDev se describe que tipo de dispositivo es el que se quiere crear (Fifo-Flit, Channel, Internal, Processor o CrossSwitch).

Estos dispositivos heredan estas variables. La clase Channel además añade su parámetro de Turn (para saber si ha llegado su turno y puede transmitir) y una variable booleana que nos dice si ha sido transmitido o no, Transmitted.

Otras clases que forman la red son los Buffer y los Nodos.
La clase *Buffer* contiene un puntero a la clase Red para conocer su estado(class Red *LaRed), y otro puntero para saber que flits hay en el buffer (FifoFlit * Fifo).
La clase *Node* además de apuntar a la Red, apunta a los Buffers de entrada y salida(Buffer * in, *out) y al CrossBar. Incluye también los procesadores de entrada y salida (Processor ElProcIn, ElProcOut).

Ya por último, la clase CrossBar está compuesta por un puntero a la Red(como todos los elementos de la red) y un CrossSwitch para cada Buffer de salida implementado mediante un puntero(CrossSwitch * output).

Por otro lado esta la clase que define el paquete:

```
class Packet{
public:
bool Transmitted; // Indica que el paquete llego entero
bool Blocked; // Indica si el paquete se encuentra bloqueado
int Length; // Flits del paquete
int Src,Dest; // nodo origen y destino
int Cycles; // ciclos de reloj consumidos
int CycBlock; // ciclos en los que esta bloqueado totales
int CycBlockEmit; // ciclos en los que esta bloqueado en el proc
```

```
emisor
int CycBlockRecep; // ciclos en los que esta bloqueado en el proc
receptor
int CycHead; // ciclos que tarda la cabeza en total
int PathNodes; // Nodos atravesados en los caminos recorridos
int Color; // Color RGB para pintar
Flit *ElFlit; // matriz de flits
class Red *LaRed;
Packet *Next;
Packet *Prev;
Packet(class Red *unared, int longit, int orig, int dest); // genera
un paquete
~Packet();
Packet* RunCycle(); // avanza el paquete un ciclo
Device* Mete(Device* disp, int i); // introduce un flit en un dispositivo
Device* Saca(Device* disp, int i); // saca un flit de un dispositivo
}
```

La clase Flit describe en su interior la posicion del flit en el paquete y en que dispositivo se encuentra.

Pero la más importante es la clase Red, que construye, gracias a todas las anteriores, la red que ha definido el usuario:

```
class Red{
     int Switching; // Tipo de control de flujo
     public:
        Channel *Channels; // Conjunto de Canales
        Node *Nodes; // Conjunto de Nodos
        Packet *Packets; // Puntero al inicio de la lista de paquetes
        int Cycles; // Cycles desde el inicio
        int CyclesEmit; // Cycles durante la emision de paquete
        int LatLast; // Latencia del último paquete
        int LatTrans; // Suma de latencias (solo movimiento)
        int LatBlock; // Suma de latencias (durante bloqueos)
        int LatBlockEmit; // Suma de latencias (durante bloqueos
en el p. emisor)
        int LatBlockRecep; // Suma de latencias (durante bloqueos
en el p. receptor)
        int LatTotal; // La suma de las anteriores
        int LatHead; // Suma de Latencias de la cabeza
        int EmiFlits; // Flits emitidos
        int NumPackTrans; // Packets Transmitidos completamente
        int NumPackEmit; // Packets Emitidos
        int PathNodes; // Nodos atravesados en los caminos recorridos
```

```
        int Routing; // Mecanismo de encaminamiento
        bool BloqueoTotal; // Indica si se produjo un bloqueo mortal
        int NumNodes; // Numero de nodos
        int NumBuf; // Numero de buffers por nodo
        int Dimensions; // Numero de dimensiones
        int NumNodesDim ; // Numero de nodos por dimension
        int NumVirtuals; // Numero de canales virtuales
        int NumFlitBuf; // Numero de flits en el buffer
        int Directions; // 1=unidireccional, 2=bidireccional
        int Forwarding; // 0= no se adelantan los flits en el fifo,
1= si
        bool PhysChannel; // Indica si los canales virtuales son
fisicos en realidad
        int FifoDelay; // Retraso del fifo
        int CrossDelay; // Retraso del crossbar
        int ChannelDelay; // Retraso del canal
        int SwitchDelay; // Retraso en la conmutacion
        Red(int dimen, int numdim, int numvirt, int numflitbuf,
int direcciones, int adelanto, bool fisicos, int routing,
int switching, int fiforetraso, int interretraso,
int canalretraso, int conmuretraso);
       ~Red();
        void RunCycle(); // Calcula un nuevo ciclo
        int ChannelId(int numnodo, int dimcanal, int direccion, int
numvirt);
        int Trans(int act, int Dim, int di); // devuelve el nodo
adyacente de act en la dimension Dim sentido di
        Device* RoutingOrderDimMesh(int nodo,int Dest);// Algoritmos
de routing
        Device* RoutingOrderDimTorus(int nodo,int Dest);
        Device* RoutingDuatoOrderDimMesh(int nodo,int Dest);
        Device* RoutingFullyAdaptiveMesh(int nodo,int Dest);// Completamente
adaptativo para mallas}
```

Vamos a ver como se interconectan todas las clases hasta ahora vistas en el siguiente diagrama de colaboración de la clase Red (figura 5.2): Ahora que ya se ha visto la estructura interna básica del simulador, se va a ver como se crea una simulación.

### 5.3.2. La simulación

Cuando el usuario se define la red, junto con los retrasos asociados a cada dispositivo, y decide simular, la aplicación construye la red creando en una primera instancia los dispositivos necesarios e interconectandolos más

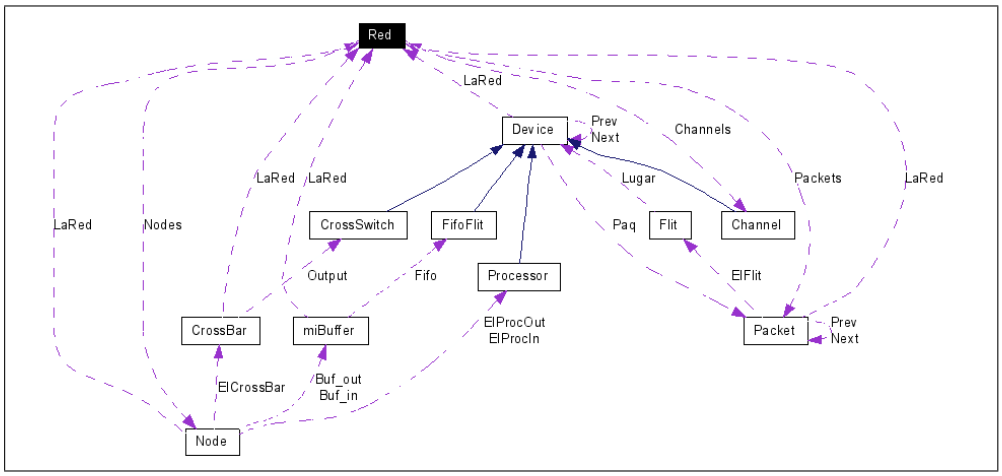


Figura 5.2: Diagrama de colaboración de la clase Red

tarde. A partir de ese instante ya se ha creado la estructura red, y se puede recorrer mediante los punteros de cada dispositivo. La Red no es más que una lista de dispositivos conectados. Una vez creada la red, el programa llama a la función **Simula(...)** que es la que realiza la simulación con los parámetros especificados

```
Simula(FILE * fich, int paqlong, int cablong, int Tope, float
        TasaEmit, int *continuar, int noleelong)
```

Esta función en una primera instancia crea los paquetes necesarios según la tasa de emisión(TasaEmit) y el número de paquetes que queremos enviar (Tope). Una vez creados esos paquetes se llama a RunCycle de Red. Este RunCycle se encarga de llamar a RunCycle de Paquete, que es el que mueve los paquetes por la Red. Se puede observar en el diagrama de secuencia resumido de la figura 5.3

### RunCycle: Un ciclo de simulación

La función RunCycle de Red funciona de la siguiente forma: mientras haya paquetes va llamando a la función RunCycle de Paquete.

Es aqui donde se observa que va lanzando de manera **secuencial** los paquetes siguiendo la lista de paquetes activos. Mientras la lista no esta vacía la red va moviendo un ciclo cada paquete.

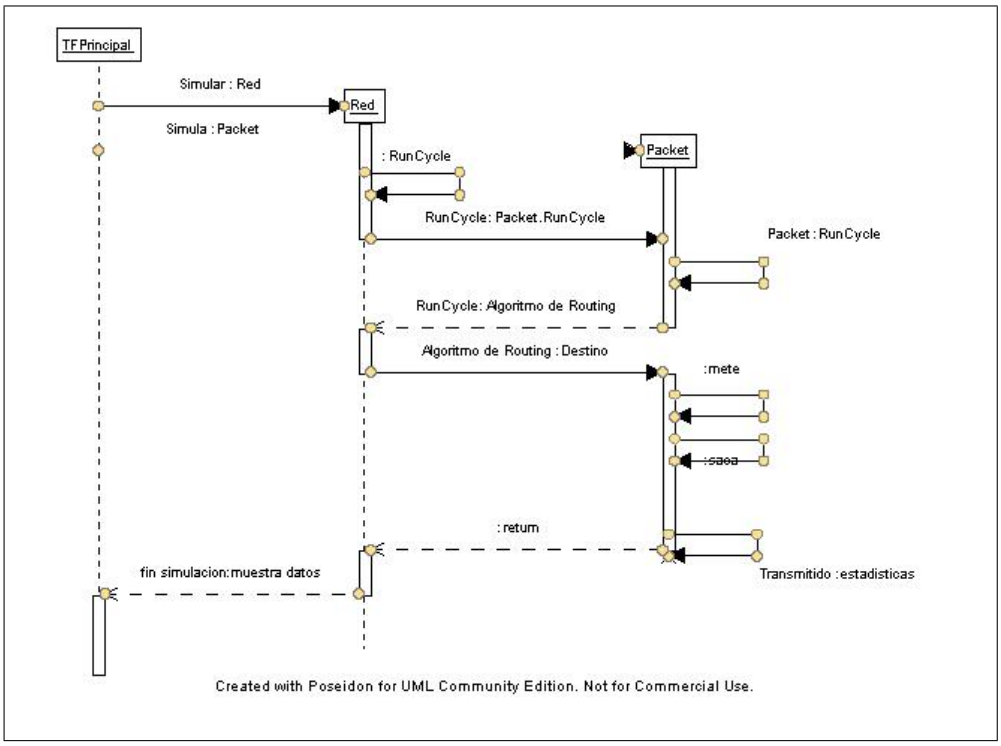


Figura 5.3: Diagrama de secuencia de una simulacion

```
   void Red::RunCycle()
{
Packet *aux,*uno;
int i;
     Cycles++;
     // Pone los canales a transmitido=false
     if (NumVirtuals>1)
     for (i=0;i<NumNodes*NumBuf;i++) Channels[i].Transmitted=false;
     // Mueve paquetes:
     aux=Packets;
     BloqueoTotal=true;
     while (aux!=NULL)
     {
        uno=aux→Next; // necesario pues el paquete puede desaparecer
        // ejecuta y/o borra paquete
        aux=aux→RunCycle();
        if (aux!=NULL)
        {
           if (aux→Blocked==false)
```

```
            BloqueoTotal=false;
        }
        else
        {
          BloqueoTotal=false; // si es que salio alguno hay vida
aun...
        }
        aux=uno;
        }
     }
```

### 5.3.3. Conclusiones sobre la herramienta actual

Para paralelizar esta aplicación nos deberemos centrar en intentar ejecutar 'RunCycle de paquete' de forma simultánea para distintos paquetes. Así se podrá agilizar el proceso de la lista con el consiguiente ahorro temporal.
Aqui radicará nuestro problema, en poder lanzar en procesadores distintos el cálculo de procesar un paquete dentro de un ciclo de red.
Si conseguimos procesar la lista de paquetes en distintos procesadores de forma simultánea estaremos ahorrando un tiempo que procesado de manera secuencial supone un retardo importante.

## 5.4. Análisis de las diferentes estrategias

El simulador Simured realiza un bucle mientras hayan paquetes en la lista pendientes de ser procesados. Este proceso se realiza de forma secuencial. Se deberán buscar métodos para poder procesar más de un paquete al mismo tiempo.

### 5.4.1. Identificación de alternativas

En primer lugar se nos presenta como paralelizar la aplicación.
Existen dos posibilidades, hacer una aplicación multiproceso o una aplicación multihilo.

#### Aplicación multiproceso

Los procesos son lentos de crear, tienen independencia de datos por consiguiente se requeriría muchos cambios de contexto para comunicar los datos de la simulación. El sistema operativo debería soportar este mecanismo.

### Aplicación multihilo

Los hilos son entidades ligeras, comparten los datos del proceso y son rápidos de crear. Si un hilo modifica una variable del proceso, el resto de hilos verán esa modificación cuando accedan a esa variable.

Los cambios de contexto entre hilos consumen poco tiempo de procesador.
Las diferentes alternativas que se presentan con los hilos tienen que ver con dos lenguajes de programación, el lenguaje C++ y el lenguaje Java.

### Realización en C++ con Kilyx

Como se vió en el estado del arte, el lenguaje C/C++ nos da una gran velocidad de procesado, no requiere transformar el código existente, y además mediante Kilyx se podría conseguir que la aplicación funcionase tanto en Windows como en Linux. Tiene la desventaja que requeriría modificar y crear una nueva librería de hilos y por otro lado, intentar retomar un proyecto olvidado como Kilyx para C++, con los consiguientes fallos sin corregir que puede tener (y tiene...). Posiblemente esta solución requiera el pago de alguna licencia(para Windows).

### Realización en Java

La plataforma Java nos asegura esa portabilidad de código que se nos exige. Al utilizar la Java Virtual Machine nos cercioramos que la aplicación que se creará funcionará en cualquier sistema operativo. La utilización de hilos es bastante sencilla y está documentada. Está en constante desarrollo, por lo que nos asegura un soporte adecuado. Tiene la desventaja de tener que realizar un 'port' a Java de la herramienta actual escrita en Borland C++. Java es bastante lento para procesar gráficos, pero nos permite poder realizar miniaplicaciones Web (applets), con lo que se podría integrar este proyecto en una página web.

Java dispone también de un método para realizar el acceso exclusivo a una zona de código. En Java, todo objeto tiene asociado un cerrojo. Cuando un método es declarado **synchronized** el hilo que lo ejecuta debe obtener el cerrojo asociado con el obejto antes de que pueda continuar. Cuándo el método finaliza, ese cerrojo es liberado automáticamente. Si dos hilos intentan acceder al cerrojo al mismo tiempo sólo uno lo logrará. El otro hilo debe esperar a que el primero libere el cerrojo.

Para proteger las variables Java proporciona una solución bastante elegante: la palabra reservada **volatile**. Si una variable es declarada *volatile*,

cada vez que esa variable es usada será leída de la *main memory*. Igualmente, cada vez que la variable seá modificada, su valor deberá ser escrito en la memoria principal. Además, cada objeto proporciona un mecanismo para mantenerse en espera y que ayuda asi la comunicación entre hilos.
La idea es simple: un hilo necesita cierta condición para ejecutar su código y asume que otro hilo creará esa condición. Cuando otro hilo crea esa condición avisa al hilo que estaba esperando por esa condición.
Este mecanismo es implementado por las funciones wait (espera una condición) y notify (avisa) o notifyall (avisa a todos los que esperaban esa condición). Con este método se consigue gran parte de la sincronización entre objetos.

### 5.4.2. Elección de la alternativa a desarrollar

Pese a tener que reescibrir todo el programa, la alternativa que se presenta más conveniente es realizar el proyecto en Java. Tener que reescribir todo el código se puede considerar interesante para comprender mejor el funcionamiento del simulador secuencial y poder realizar mejoras para su paralelización. Además esta solución será multiplataforma sin tener que realizar ningún cambio en el código. No nos requerirá el pago de ninguna licencia al ser Java *Open Source* .

Considerando el balance ventajas contra desventajas el utilizar Java requerirá un poco más de tiempo de desarrollo, pero sus ventajas son superiores a esa perdida de tiempo en realizar el cambio de lenguaje, que no es realmente una pérdida pues nos permite analizar el simulador actual.

### 5.4.3. Metodología y Material necesario

Como se ha podido comprobar el proyecto se realizará en dos etapas:

- En una primera parte se realizará la aplicación secuencial en Java, es decir, se realizará el 'port' a Java de la aplicación actual.
- En una segunda parte se desarrollará la paralelización de esta nueva aplicacón Java.
- Ya por último se integrarán los módulos visuales (generación de gráficas).

Como se observa se utilizará una metodología de desarrollo incremental. En la parte de paralelización del código, al ser una tarea de investigación se utilizará una *metodología en espiral*, realizando continuas pruebas de las alternativas, se irán documentando las soluciones que se utilicen, tanto las que se rechacen como las que se acepten como solución final.

### Material necesario

Para desarrollar la solución adoptada se necesitarán los siguientes materiales:

- **Un ordenador Pc** Para desarrollar la aplicación, a ser posible potente para que realice los cálculos rapidamente y soporte el entorno de desarrollo que hemos elegido.
- **Un ordenador multiprocesador** Se necesitará un ordenador de más de un procesador para poder comprobar la paralelización.
- **Sistemas operativos** Necesitaremos un sistema operativo Windows y otro Linux para comprobar su funcionamiento multiplataforma. Se ha elegido Windows XP (con licencia de la Universidad de Valencia) y Linux Suse 9.1 y Linux RedHat.
- **Compilador de Java** Utilizaremos la versión 5 del entorno de desarrollo de aplicaciones Java de Sun (jdk 1.5) junto con su JVM (descargable desde [Mic05]).
- **Entorno de desarrollo para Java** Se ha pensado en Netbeans (versión 4.0) por ser gratuito y ser recomendado por Java Sun(descargable desde [Net05]).
- **Compilador de Latex**Para redactar la memoria utilizaremos MikTex [Sch05] por ser actualizable on-line y dar un gran soporte junto con el editor TexnicCenter.(descargable desde [Too05]). Ambos son open source.

# Capítulo 6

# Diseño

Para diseñar la aplicación como se ha visto se dividirá en dos el diseño.

En una primer estudio se diseñará la versión secuencial. Esta versión será simple, presentando las tareas básicas para simular y comprobar los resultados, en formato texto y mediante el paso de paquetes de forma visual.

En un segundo estudio se partirá de la versión secuencial realizada en Java para diseñar la manera de paralelizar la ejecución de la simulación.

## 6.1. Versión Secuencial

Para diseñar esta versión secuencial se realizará un 'clon' de la aplicación realizada en C++. En una primera etapa al ser Java un lenguaje de objetos puro el primer paso consistirá en dividir en clases esta versión en C++. Como Java no separa la definición de la clase con la implementación de sus métodos se reorganizará el código. Se obtendrán entonces las clases que definen los elementos de la red:

- **Device** Describirá el tipo dispositvo de la cual heredan CrossSwitch, Processor, Channel y FifoFlit
- **Node**. Describirá el tipo nodo junto con sus métodos
- **Processor**. Describirá el tipo procesador junto con sus métodos
- **TypeDev**. No es más que una enumeración que definirá el tipo de dispositivo.
- **Channel**. Describirá el tipo canal junto con sus métodos
- **CrossBar**. Describirá el tipo crossbar junto con sus métodos
- **CrossSwitch**. Describirá el tipo crossSwitch junto con sus métodos

- **miBuffer**. Describirá el tipo Buffer junto con sus métodos
- **FifoFlit**. Describirá el tipo FifoFlit junto con sus métodos
- **Flit**. Describirá el tipo Flit junto con sus métodos
- **Packet**. Describirá el tipo Packet junto con sus métodos
- **Red**. Describirá el tipo Red junto con sus métodos
- **simuredJava**. Este clase es el form principal, aqui se implementarán los métodos para ejecutar el programa.
- **Dibujo**. Esta clase es donde se dibujarán las simulaciones

En esta transformación a Java se perderá la interactividad con la simulación al no corresponder las funciones De Borland C++ con las de Java. Una vez se ha portado todo a Java se necesitará pues que la aplicación no se quede bloqueada al llamar a Simular, y se pueda interactuar con el interfaz para interrumpir, reescalar la ventana, y otras acciones. Para ello se definirán dos clases más que heredarán de la clase Thread. Es decir, se crearán dos hilos:

1. un primer hilo que controlará la función *Simular*, que como hemos visto es la que crea la Red, y al ser ejecutado este hilo no bloquerá el form de Dibujo.
2. un segundo hilo que será la función *Simula*, que es la que realiza el movimiento de paquetes por la red. Al ser un hilo se podrá actuar sobre el y **sincronizar** su ejecución.

Mediante una variable booleana controlaremos la ejecución de la simulación, que podremos decidir que ha finalizado o no con solo cambiar su estado.

Para Pausar y reanudar la simulación se utilizarán las sentencias wait() y notify(), que pausaran y reanudarán el hilo deseado.

Para el Paso a Paso se utilizará una variable booleana que nos indicará si esta activo el paso y sentencias wait/notify. El mecanismo será el siguiente: al pulsar sobre el botón *'paso a paso'* se activará la variable booleana realizará un paso de simulación y si esta variable esta activa llamará a wait(), al volver a pulsar sobre el botón realizará un notify(), realizará otro paso y volverá a llamar a notify(), y así succesivamente.
Para salir de 'Paso a Paso' el botón *'Continuar'* pondrá la variable paso a paso a false y notificará.

Para el diseño de los interfaces con el usuario utilizaremos la librería Swing de Java, que ofrece una gran cantidad de componentes desde botones

hasta tablas. Nos será de gran utilidad para realizar un interfaz amigable y estructurado. Utilizaremos 'NullLayout' para dar un aspecto lo más parecido al simulador anterior.

## 6.2. Versión Paralela

Se utilizará la versión secuencial para realizar la versión paralela. Para realizar la paralelización se estudiarán las siguientes alternativas:
Se podría pensar en crear un hilo por paquete a procesar, es decir, que la función *run* del hilo fuese la que mueve un paquete por la red. Esta solución sólo funcionaría para pocos paquetes a la vez, pues introduce mucha sobrecarga de hilos.

Las soluciones que se van a presentar tendrán limitados el número de hilos. Para las labores de sincronismo se dispone de la libreria *java.util.concurrent* que implementa la mayoría de mecanismos para el manejo de hilos.

### 6.2.1. Cola de Trabajo (WorkQueue) y un contador

Nos crearemos una cola de trabajo e iremos insertando paquetes (hilos) (figura 6.1). Con un contador sincronizado esperaremos a que acaben todos los paquetes de un ciclo de Red.
La cola de trabajo aseguran los expertos es una manera rápida de pasar tareas a los procesadores. Se tratará de una FIFO de tareas por ser procesadas que se irán asignando a los hilos que se hayan definido. Los paquetes deberán heredar de la clase hilo para poder ser procesados en paralelo.

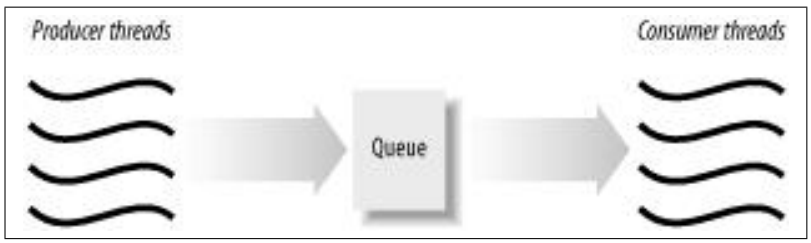


Figura 6.1: Cola de Trabajo

### 6.2.2. Conjunto de Hilos y Sincronización mediante Barrera

La idea es crear un conjunto fijo de hilos (un pool de threads) de tamaño fijo e ir asignando a esos hilos tareas(paquetes). Para ello el paquete deberá heredar de hilo, como en el caso anterior.Un esquema explicativo se puede ver en la figura 6.2. Para Sincronizar este método se recorrerá en un

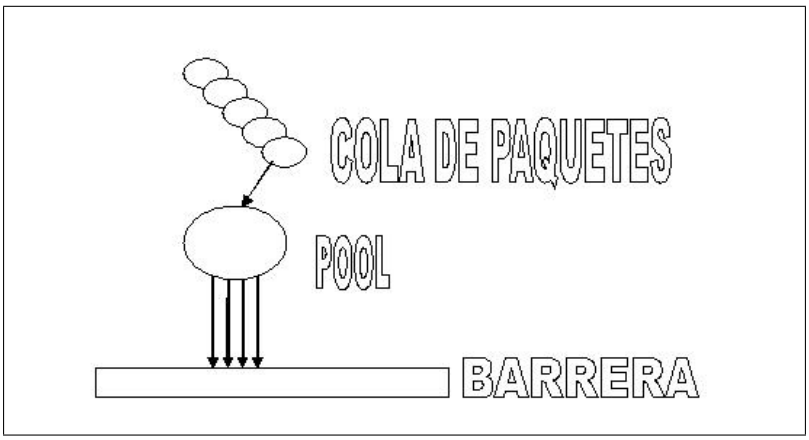


Figura 6.2: Pool de Hilos y sincronización mediante Barrera

principio la lista de paquetes por procesar en un ciclo de Red y se creará una Barrera para esperar a que finalicen todos los paquetes de ese ciclo.

### 6.2.3. Creación de un número fijo de hilos y sincronización con wait y notify

Este es un diseño propio. Se trata de crear una nueva clase que cree un número fijo de hilos y lance los hilos. A cada ciclo de Red se le irán asignando a esos hilos paquetes para ser procesados. El funcionamiento que se quiere conseguir es el siguiente y se puede observar en la figura 6.3:
En el constructor de Red se creará una nueva instancia de esta nueva clase,

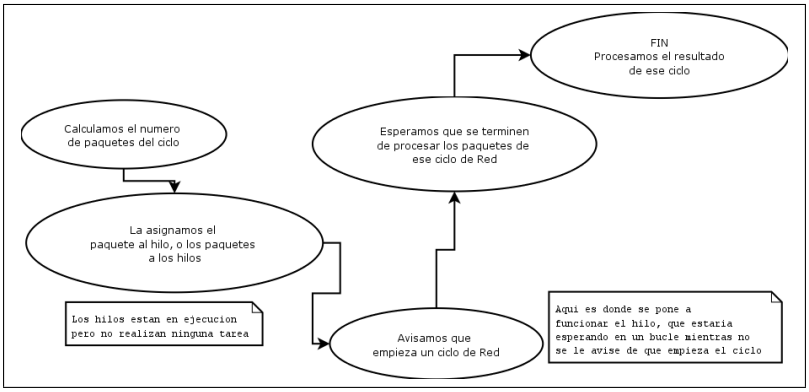


Figura 6.3: diagrama del comportamiento deseado de la simulación

que creará un número fijo de hilos trabajadores y lanzará su ejecución. En un primer recorrido de la lista conoceremos los paquetes a lanzar en ese

ciclo. Al no tener ningún paquete asignado estos hilos estarán esperando hasta que se asigne un paquete al hilo.
En RunCycle de Red se asignará el paquete y se avisará del comienzo de un ciclo. Como el número de hilos esta limitado, se asignarán en principio los paquetes posibles y mediante objetos de sincronización se esperará a que los hilos esten libres, es decir hayan terminado de procesar el paquete que tenían asignado. Cuando ese suceso ocurre se llamará a una función que liberará un objeto de sincronización para así poder asignar un nuevo paquete a un hilo sin tarea.
Mientras estemos en un mismo ciclo de red no se podrá salir de este mecanismo, y los paquetes que estén en la lista de ese ciclo se irán asignando a medida que se liberen hilos. Si no hay hilo libre el paquete esperará su turno. Se tratará de un mecanismo que irá liberando paquetes a medida que puedan ser procesador por un número fijo de hilos.
El diagrama de flujo de un hilo se puede observar en la figura 6.4. Para evi-

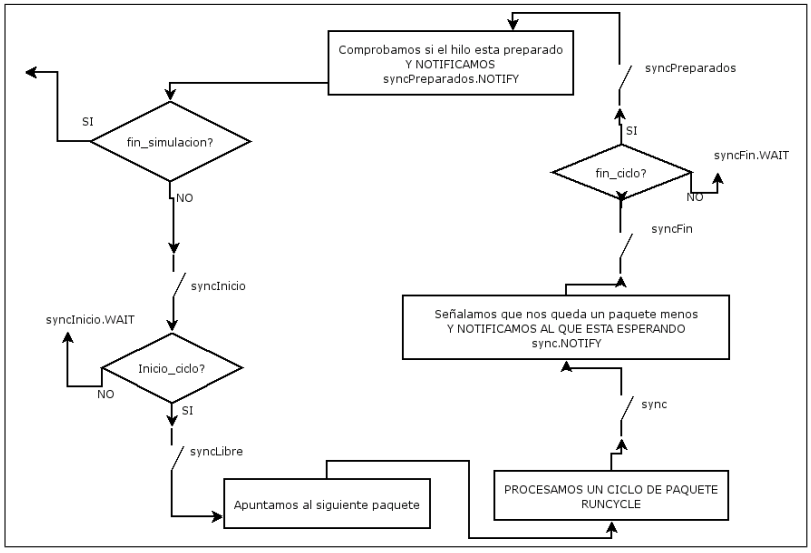


Figura 6.4: Diagrama de flujo de un hilo

tar la desincronización se utilizarán zonas protegidas (cerrojos) dónde solo se podrá acceder de forma exclusiva. Un objeto de sincronización alertará el fin de la simulación y matará los hilos para que no queden en memoria.
Las variables se protegerán para ser leídas y escritas en la memoria principal y asi asegurarnos que se lee y escribe la última modificación (mediante la palabra reservada *volatile*). Con este método se espera no encontrar ningún tipo de sobrecarga con la creación y destrucción de hilos, la única sobrecarga que se puede dar está en el cambio de paquetes por procesar y en la sincronización de esperas.

## 6.3. El applet

Un applet es una pequeña aplicación software diseñada para ser incorporada a una página Web y hacerla interactiva. Necesita que el navegador soporte Java. Al ejecutarse del lado del cliente, tienen algunas restricciones de seguridad, como es que no puede leer y menos escribir en un archivo. Es por esto que nuestro applet estará limitado a realizar simulaciones, y no podrá guardar los resultados en un fichero. Para realizar un applet no hay más que heredar de la clase JApplet y comentar el metodo main().

## 6.4. Diseño del banco de pruebas (TestBench)

Debido a la gran cantidad de parámetros que pueden variar en nuestro simulador se pensó en diseñar un banco de pruebas centrándonos en las variables que podián influir en el rendimiento. No se trata por lo tanto de un banco de pruebas para demostrar el correcto funciomaniento del simulador en sí, si no por contra de comprobar el rendimiento de este proyecto en base a los parámetros que pueden hacernos ganar en tiempo de ejecución, por tanto nuestro principal estudio será el tiempo de simulación. De todas maneras en unas primeras pruebas se crearán unos ficheros con simulaciones predefinidas y se comprobarán los resultados con las mismas simulaciones ejecutadas en el simulador Simured en C++.

Debido a la imposiblidad de ejecutar las pruebas en una máquina de caracter local, se optó por realizar este estudio en una máquina remota, por lo que al tener un acceso compartido(en ocasiones se podría estar ejecutando la simulación junto con otras aplicaciones) la solución elegida fue la de realizar las pruebas un conjunto de veces y realizar la media temporal de esta, eliminando los tiempos extremos, para así obtener un tiempo estimado con cierta coherencia.

### 6.4.1. La máquina multiprocesador

Para realizar las pruebas se pidió al Departamento de Informática de la Universidad de Valencia un ordenador multiprocesador. El departamento nos facilitó una cuenta en 'sonoma'. 'sonoma' es un clúster de 13 máquinas. El servidor tiene un punto de acceso al mundo, conocido como 'sonoma.quifis.uv.es' .Esta es una máquina de 32 bits que sólo utilizaremos para albergar los ficheros de usuario y como puente de acceso al clúster. Esta máquina es conocida 'desde dentro' como 'sonoma00'. Por supuesto, las sonomaXX están aisladas del exterior, así que todo se debe hacer utilizando el puente.

Utilizaremos las máquinas sonoma17 a sonoma20. Son máquinas Dual Amd64 . Están interconectadas por la Myrinet y cuentan con compilador PGI 5.2 y LAM/MPI 7.1.1.

Como se observa, solo podremos comprobar el rendimiento en una **máquina Dual**.

# Capítulo 7

# Implementación

## 7.1. Implementación de la versión secuencial

Como se ha visto en la parte de diseño, se ja intentado realizar un 'clon' de la aplicación anterior. Al tener muchas similitudes el C++ y el Java se ha podido realizar esta transformación conservando la esencia del simulador anterior. Mediante la librería Swing se realiza un interfaz casi idéntico al anterior.

El form principal es un JFrame con un panel con pestañas, se puede observar en la figura 7.1

El form Dibujo es otro JFrame con un JPanel para el dibujo de la versión visual, se puede observar en la figura 7.2

Al pulsar el botón simular (dentro del form de dibujo) se crea el hilo Simular:

```
   Simular hilo;
...
hilo =new Simular(this, null);
hilo.start();
...
```

La implementación de la llamada a Simula dentro del hilo Simular se ha implementado de la siguiente manera:

```
   HiloSim = new Simula(Princ,in, paqlong, pp, numpaq, (float)tasaemit,
Princ.Continuar, noleelong, this);
HiloSim.start();
try{
      HiloSim.join();// Esperamos a que termine la simulacion para
```

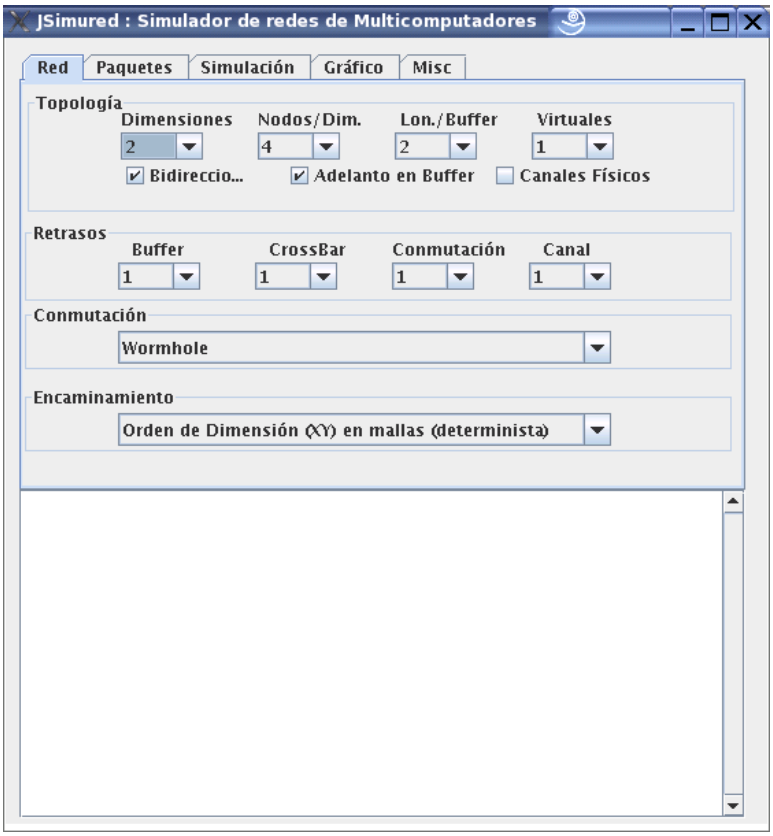


Figura 7.1: Form principal del simulador JSimured

```
continuar
}
catch(java.lang.InterruptedException e){
      System.err.println(.Exception en llamada a Join simula"+e);
}
Princ.ShowStats(); // y mostramos estadísticas
```

### 7.1.1. Prueba de funcionamiento

Para probar el correcto funcionamiento se ha comprobado con ficheros de trazas en el simulador en C++ y en nuestra versión secuencial obteniendo los mismos resultados lo que nos asegura que el simulador secuencial en Java, base para realizar nuestra paralelización funciona correctamente. Los ficheros de prueba se pueden consultar en el CD adjunto.

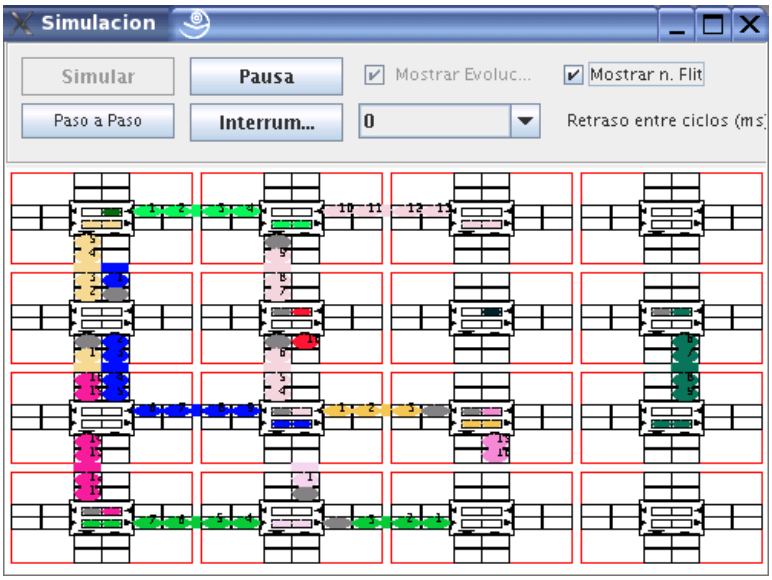


Figura 7.2: Form Dibujo del simulador JSimured

## 7.2. Impementación de la versión paralela

Para implementar la versión paralela se habían distinguido tres posibilidades:

- Cola de Trabajo
- Pool de Threads con Barrera
- Lanzador de Paquetes y sincronización de hilos mediante wait y notify

### 7.2.1. La Cola de Trabajo

Para implementar la cola de trabajo se ha definido una clase WorkQueue. Esta clase tiene una lista donde se le asignan paquetes.

```
public WorkQueue(int nThreads, int totalThreads) {
this.nThreads = nThreads;
queue = new LinkedList();// esta es la cola de trabajo
threads = new PoolWorker[nThreads];
nTotales = new Integer(totalThreads);
for (int i=0; i<nThreads; i++) {
     threads[i] = new PoolWorker();
     threads[i].start();
}
}
```

Tenemos un pool de trabajadores que irá sacando de la lista los paquetes por procesar, e irá lanzando su método run().

```
private class PoolWorker extends Thread {
  public void run() {
  Runnable r;
  while (true) {
     synchronized(queue) {
        while (queue.isEmpty()) {
        try
        {
           queue.wait();
         }
        ...
     }
     r = (Runnable) queue.removeFirst();
  }
```

En la función RunCycle de Red lanzaremos los paquetes de cada ciclo en paralelo, en un principio habremos recorrido ya la lista de paquetes para conocer cuantos debemos esperar para pasar de ciclo, es decir cuantos paquetes debera esperar :

```
Cola.hilosEsperar(nHilos); // Esperaremos que terminen los nHilos,
es decir los paquetes de ese ciclo aux=Packets;
while(aux!=null){// Mientras la lista no esta vacia se procesan
     try
     {
        h = new hiloPacket(aux, this);
        Cola.execute(h);
        ...
        aux = aux.Next; // siguiente paquete
     }
     Cola.esperaTerminar();
```

#### Pruebas de funcionamiento

Comprobamos que este método se bloquea, no funciona adecuadamente, por lo que no continuaremos con esta vía de investigación. Parece que no es conveniente crear paquetes de tipo hilo en cada ciclo.

### 7.2.2. El Pool de Threads

Para implementar este método nos hemos creado un objeto de tipo ThreadPool, este tipo es una clase que define un conjunto de hilos a los que se les puede asignar trabajo, si hacen falta más hilos los creará automática-

mente.

Se ha implementado este método de la siguiente manera:

```
    pool = new ThreadPool(poolSize); // poolSize = numero de hilos
que queramos
hilosLatch = new CountDownLatch(nHilos); // Barrera del número de
paquetes de ese ciclo.
aux=Packets;
while(aux!=null){
      try
      {
       pool.addWorker(aux);          aux=aux.Next;
      }
....
// espera a que los hilos terminen de calcular un ciclo
//////////////////////////////////////////////////////
hilosLatch.await();
```

junto con el método run de packet que realiza la llamada al RunCycle y calcula los tiempos:

```
public void run(){
        Packet aux;
        aux = RunCycle();
       LaRed.pool.workerDone(this,false);
          if(Blocked == false)
                    LaRed.BloqueoTotal = false;
                if (Transmitted) {
                    Calcula();
                    Delete();
                }
        LaRed.hilosLatch.countDown();
}
```

Dentro del método run de paquete como vemos deberemos decrementar el número de hilos a esperar una vez haya finalizado de procesar.

```
...
hilosLatch.countDown();
....
```

También se han probado métodos utilizando la interfaz *Executor* de la librería *java.util.concurrent*.

```
public static ExecutorService newFixedThreadPool(int nThreads)}
private final ExecutorService pool; // nos definimos el servicio
pool = Executors.newFixedThreadPool(poolSize);
```

**Pruebas de funcionamiento**

Al realizar las pruebas nos ocurre lo mismo, tiende a quedarse bloqueado y consumir toda la CPU. Pese a probar diferentes variantes sobre este método no se consigue hacerlo funcionar correctamente, y cuando funciona emplea un tiempo desmesurado. No seguiremos con este método.

### 7.2.3. El lanzador de hilos

Para implementar este método se ha creado una clase lanzadorpacket, en esta clase se manejarán los hilos y se sincronizarán.

Se han implementado también 6 objetos de sincronización para acceder a zonas exclusivas.

```
private Object sync, syncLibre, syncInicio, syncFin, syncPreparados,
syncTermina;
```

- el objeto *sync* sincroniza la espera de un paquete para ser procesado.
- el objeto *syncLibre* sincroniza la lista de paquetes, es decir, sincroniza el movimiento por la lista mediante sigLibre, el siguiente paquete libre
- el objeto *syncInicio* sincroniza los comienzos de un nuevo ciclo en Red
- el objeto *syncFin* sincroniza el fin de un ciclo
- el objeto *syncPreparados* sincroniza la lista de paquetes pendientes
- el objeto *syncTermina* sincroniza el fin de la simulación

Al llamar al constructor de lanzadorpacket se crean tantos trabajadores como hilos hayamos definido y se lanzan:

```
public lanzadorPacket(int n) {
  trabajador = new trabajadorPacket[n];
  ...
  for (int i=0; i<n; i++){
    trabajador[i] = new trabajadorPacket();
    trabajador[i].start();
}
```

Como no tienen en principio trabajo asignado se quedaran esperando que se les asigne un paquete. El diagrama de colaboración de las clases implicadas

se puede observar en la figura 7.3

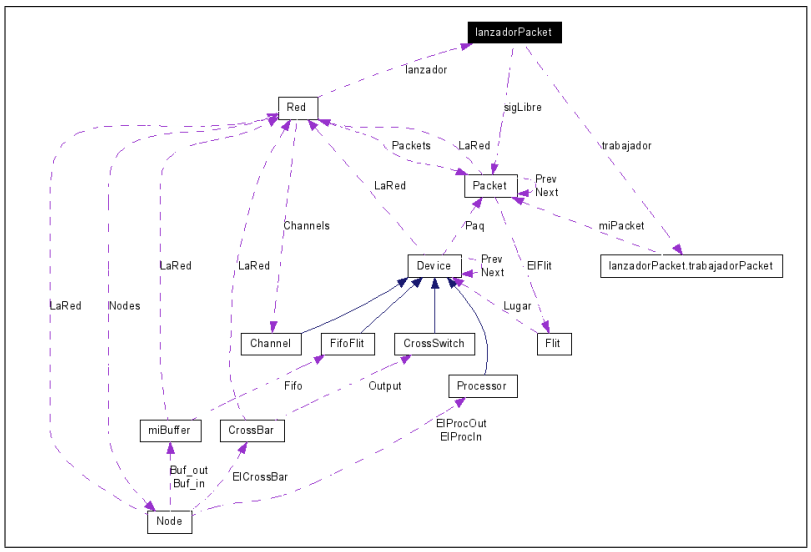


Figura 7.3: Diagrama de colaboración de la clase lanzadorpacket

La clase RunCycle de Red queda de la siguiente manera:

```
   public void RunCycle() throws InterruptedException{
Packet aux, tmp, p;
int Bloqueados;
Terminados = 0;
Lanzados = 0;
Cycles ++; // un nuevo ciclo de simulacion
// Pone los switches segun algun arbitraje
// Pone los canales a transmitido = false
if(NumVirtuals >1)
     for(int i=0; i <NumNodes*NumBuf; i++)
        Channels[i].Transmitted = false;
// mueve paquetes
//////////////////
aux = Packets;
// calculamos el numero de paquetes a lanzar
nHilos = 0;
while(aux!=null) {
     if (aux.idpaq==0){
        aux.idpaq=++nPaq;
}
     nHilos++; aux=aux.Next;
```

```
}
BloqueoTotal = true;
// Le decimos cuantos paquetes tiene que esperar, y asigamos paquete
al hilo
lanzador.Prepara(Packets);
// Avisamos de que ya se puede procesar
lanzador.InicioCiclo(nHilos);
// Esperamos a que los hilos terminen de procesar la lista
lanzador.Espera();
// procesa el resultado de los hilos
//////////////////////////////////////////
aux=Packets;
BloqueoTotal = true;
Bloqueados = 0;
while(aux!=null){
     tmp = aux.Next;
     if(aux.Blocked == false)
       BloqueoTotal = false;
     else Bloqueados++;
     if (aux.Transmitted) {
       aux.Calcula();
       aux.Delete();
     }
     aux=tmp;}
}
```

El método run del trabajador es el encargado de ejecutar un ciclo del paquete, y es dónde se mantiene la sincronización para que los paquetes no accedan a zonas protegidas y se mantengan los ciclos de simulación.

```
   public void run(){
Packet aux;
while(!fin_simulacion){
     // PUNTO DE SINCRONISMO
     ///////////////////////
     synchronized(syncInicio){
       while(!inicio_ciclo){
         try{
           syncInicio.wait();
         }catch(Exception e) {
           System.out.println(Ëxcepcion en syncInicio "+e);}
       }
      }
     while(miPacket!=null){
       synchronized(syncLibre)
```

```
        { if (sigLibre != null)
          sigLibre = sigLibre.Next;}
        synchronized(miPacket){
          miPacket.RunCycle();
       }
        synchronized(syncLibre)
          {miPacket = sigLibre;}
       }
      synchronized(sync){
        Faltan--;
        sync.notify();
       }
      // PUNTO DE SINCRONISMO
      ////////////////////////
      synchronized(syncFin){
        while(!fin_ciclo){
          try{
            syncFin.wait();
          }catch(Exception e) {
            System.out.println(Ëxcepcion en syncFin"+e);}
        }
       }
      synchronized(syncPreparados){
        if (Preparados>0) Preparados--;
        syncPreparados.notify();
       }
       }
}
}
```

**Pruebas de funcionamiento**

La aplicación termina de ejecutarse, no se bloquea, y a simple vista parece dar unos resultados correctos.
En esta implementación ya no creamos y destruimos hilos en cada ciclo, por lo que esperamos que sea más rápida que las anteriores.

## 7.3. Implementación del applet

El applet requiere unos mínimos cambios en nuestra aplicación. El primero es cambiar en nuestro form principal (simuredJava) la clase de la que hereda por JApplet.

```
public class simuredJava extends javax.swing.JApplet{...}
```

y comentamos la clase main.

Después se realiza una página web (html) dónde se aloja el applet, en ella se le da el espacio que requiere el applet para ser visualizado y la clase principal de la aplicación, en nuestro caso simuredJava y el archivo .jar donde estará nuestro código.

```
<HTML>
<HEAD>
   <TITLE>JSIMURED: Simulador de redes de Multicomputadores</TITLE>
</HEAD>
<BODY>
<H3><HR WIDTH="100%">PFC: Simulador de redes de Multicomputadores<HR
WIDTH="100%"></H3>
<P>
<APPLET code="simuredJava.class"
  archive="SimuredJava.jar"
 width="550" height="600"></APPLET>
</P>
</BODY>
</HTML>
```

## 7.4. Implementación de los módulos

### 7.4.1. Movimiento visual de paquetes

El dibujado de los paquetes se ha realizado mediante la función Dibuja() dentro del form principal. Se han creado 3 imágenes :

```
   ImgBuf = createImage(DibujoWidth, DibujoHeight);
ImgNodoBaseBmp = createImage(AnchoNodo, AltoNodo);
ImgNodoBmp = createImage(AnchoNodo, AltoNodo);
NodoBaseBmp = ImgNodoBaseBmp.getGraphics(); // espacio de dibujado
para el nodo base
NodoBmp = ImgNodoBmp.getGraphics(); // espacio de dibujado para
los cambios del nodo
```

En el NodoBaseBmp se dibuja el nodo vacio y con NodoBmp se van superponiendo las diferencias. Se utiliza ImgBuf para dibujar todo la imagen final.

### 7.4.2. Creación de Gráficas

Para implementar las gráficas se ha utilizado la librería gráfica desarrollada por Joseph A. Huwaldt (jhuwaldt@knology.net). Se han limitado los colores a 5 gráficas en un mismo gráfico. La librería se puede descargar desde [Huw05] y es de licencia GPL.
Los datos se cargan en un *ArrayList* de Simulaciones a partir del fichero CSV seleccionado de la siguiente manera:

```
   while((linea = in.readLine())!= null){
     if (linea.startsWith('"'))// Si la linea empieza con comillas
es que es el texto {
        Variables = linea.split(";");// nos quitamos los ;
        simulaciones++; // es una nueva simulación
        Bloque = new ArrayList();
        Simulaciones.add(Bloque);
     }
     else
     {
        String [] datos = linea.split(";");
        Vector simulacion = new Vector();
        for (int i = 0; i <datos.length; i++)
          simulacion.add(Float.parseFloat(datos[i]));
        Bloque.add(simulacion); // lo añadimos al bloque
     }
}
```

Para dibujar el gráfico se ha creado un nuevo JFrame dónde se añade el panel generado con la librería anterior.

```
   Plot2D aPlot = new SimplePlotXY(valoresX,valY,' ',
padre.Xtxt.getSelectedItem().toString(),
padre.Ytxt.getSelectedItem().toString(),
null, null, new CircleSymbol());
...
PlotPanel panel = new PlotPanel(aPlot);
getContentPane().add(panel);
```

Se obtienen unas gráficas del tipo de la figura 7.4:

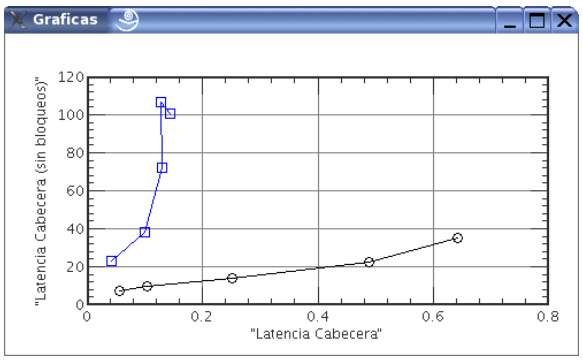


Figura 7.4: Gráficas generadas por nuestra apliación

# Parte III

# Resultados y Conclusiones

# Capítulo 8

# Pruebas y Resultados

## 8.1. Pruebas

En una primera parte se va a comprobar si el simulador funciona correctamente, para ello se utilizarán los ficheros predefinidos y comprobaremos los resultados con la aplicación realizada en C++. Una vez probados los ficheros se procederá a probar las simulaciones con generación aleatoria de paquetes.
Las pruebas que se realizarán harán variar el tamaño de la red, los retardos en los paquetes, el número de paquetes, en resumen todos los parámetros que varían en el simulador. Se comprobará tanto en un monoprocesador como en la máquina Dual. Una vez se haya comprobado el correcto funcionamiento del simulador utilizaremos nuestra versión secuencial para contrastar los tiempos de simulación con la versión paralela. En una segunda parte se probará el rendimiento variando los parámetros que se consideran interesantes. Estos parámetros que vamos a comprobar son:

- Tamaño del paquete
- Productividad
- Número de Hilos

Se comprobará el simulador paralelo en redes grandes y un gran número de paquetes, pues para redes pequeñas el simulador secuencial cumple sobradamente su función. Lo que realmente nos interesa al realizar estas pruebas es saber cuando el simulador paralelo es rentable.

### 8.1.1. Versión Secuencial

El funcionamiento se ha comprobado ejecutando los ficheros de prueba en el simulador en C++ y en nuestro simulador en Java. Como se observa en la figura 8.1 los resultados son los mismos. En el fichero de prueba prueba1.trc

se lanzan 16 paquetes en un mismo ciclo(el ciclo 2) desde todos los nodos de la red y en el ciclo siguiente se lanza otro paquete.

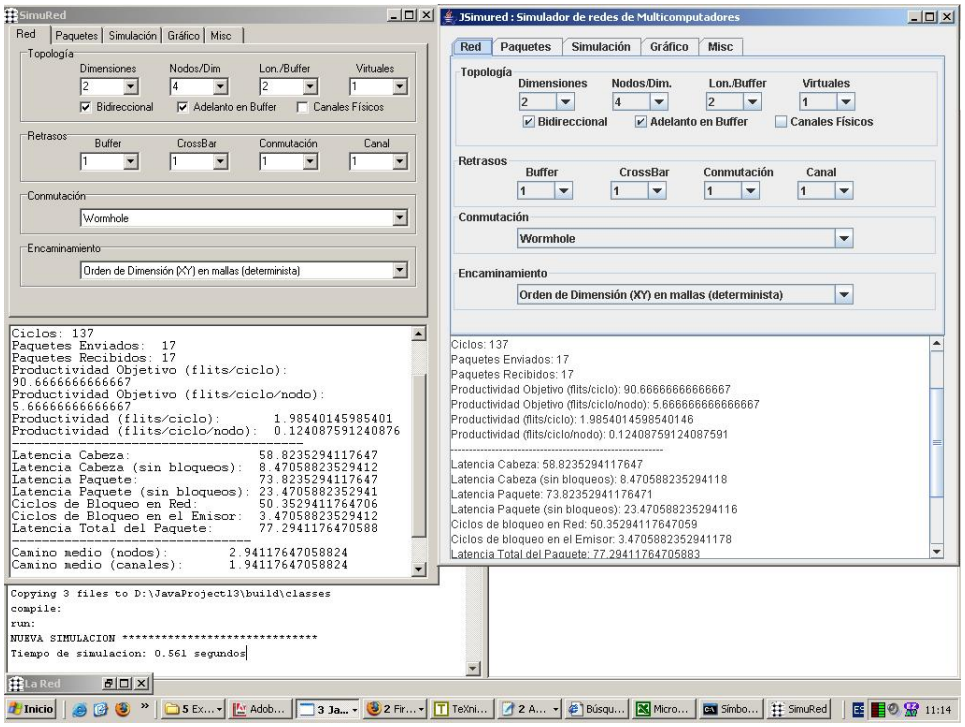


Figura 8.1: Resultados de la ejecución con el fichero de prueba prueba1.trc

Para todos los ficheros de prueba,obtenemos también los mismos resultados en los dos simuladores.

Para simulaciones aleatorias de gran dimensión se obtienen resultados válidos. Un ejemplo se puede ver en la figura 8.2

### 8.1.2. Versión paralela

Para la versión paralela también se ha comprobado primero los ficheros de traza predefinidos, primero en un monoprocesador y después en la máquina Dual. Los resultados son análogos como se puede comprobar en las figuras 8.3 y 8.4

## 8.2. Resultados

En la tabla 8.1 se ha variado el tamaño de paquete en una red de dimensión 6, con 4 nodos por dimensión, una longitud de buffer de 2 flits y 1 solo

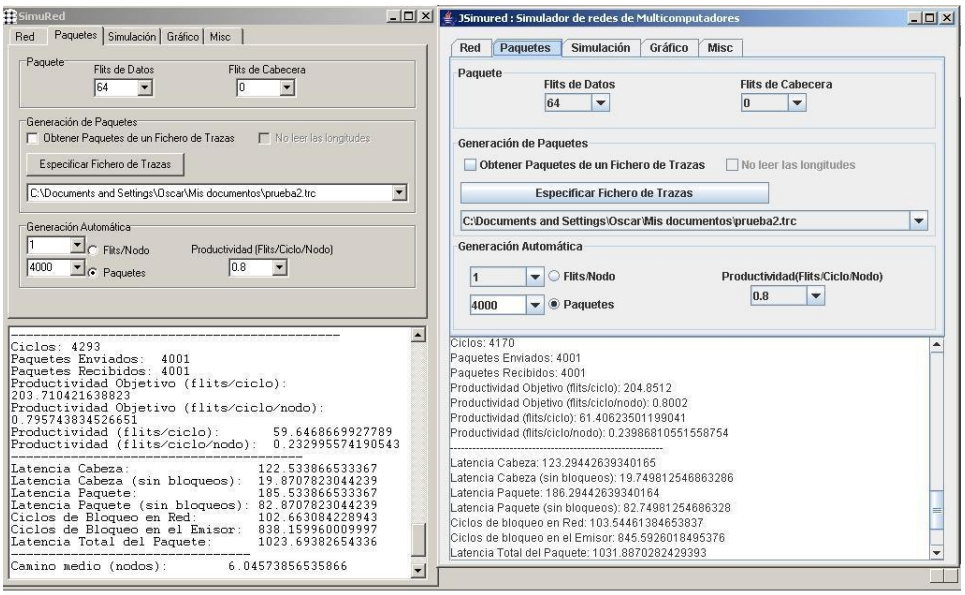


Figura 8.2: Resultados de la ejecución para red de 4 dimensiones

canal virtual, se han lanzado 8000 paquetes con productividad 0.8

Como se observa en la figura 8.5 el tiempo de simulación del simulador

| **tam.paquete** | **Tpo Secuencial** | **Tpo Paralelo** | **Ganancia** |
|---|---|---|---|
| 64 flits | 18,8 | 25,317 | 0,742584034 |
| 128 flits | 40,245 | 53,448 | 0,752974854 |
| 256 flits | 98,446 | 114,992 | 0,85611173 |
| 512 flits | 253,255 | 260,557 | 0,971975422 |
| 1024 flits | 674,031 | 638,105 | 1,056301079 |
| 2048 flits | 2.154 | 1.134 | 1,900021871 |

Cuadro 8.1: Tiempos (segundos) en función del tamaño de paquete

paralelo es inferior al del secuencial a partir de un tamaño de paquete de 1024 flits, llegando a ser de la mitad para un paquete de 2048 flits.

Si variamos la productividad a 0,3 para la red anterior obtenemos los siguientes resultados mostrados en la tabla 8.2

Observando ahora la figura 8.6 nos damos cuenta de que conseguimos aumentar el rendimiento del simulador mucho antes, ahora a partir de 256

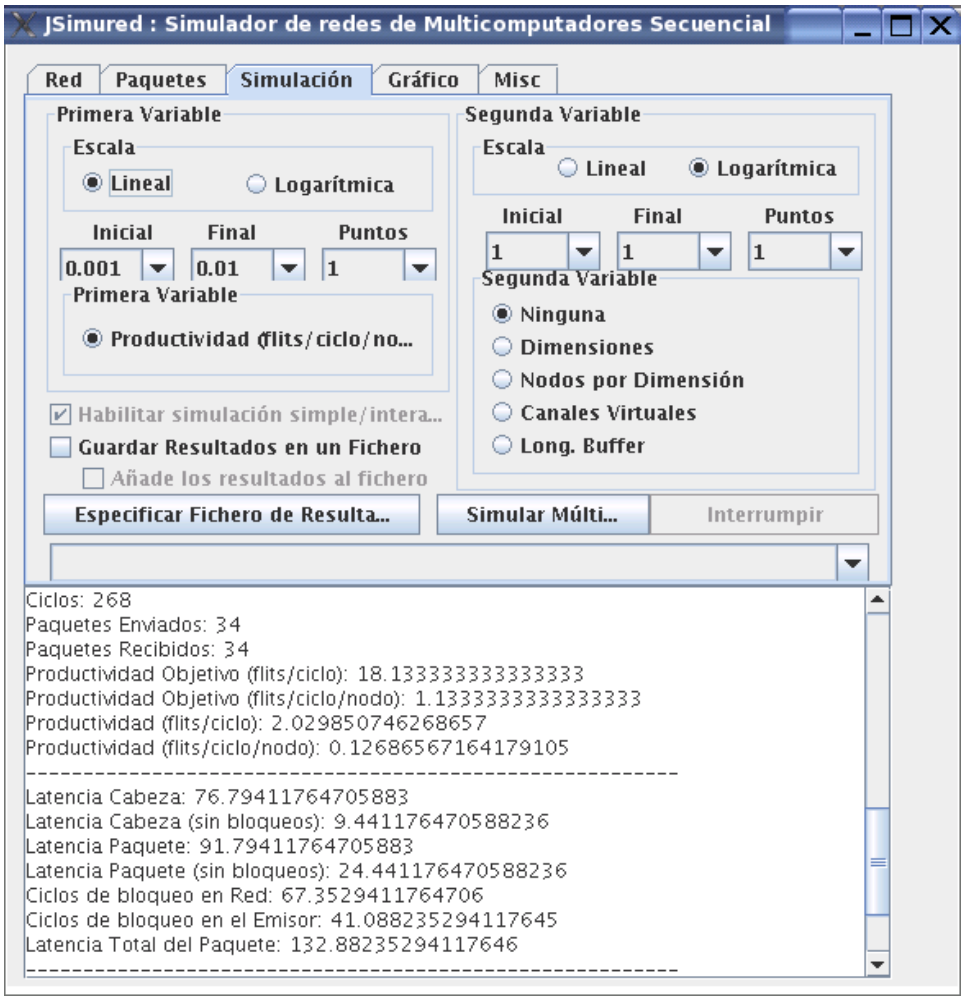


Figura 8.3: Resultado ejecución fichero prueba2.trc en Simulador Secuencial

flits ya notamos que el simulador paralelo ejecuta la simulación en menos tiempo, llegando a doblar el rendimiento a partir de un paquete de 512 flits.

Si ahora nos centramos en la productividad y escogemos un paquete de tamaño usual (128 flits) y lanzamos en esa misma red 80000 paquetes observamos la siguiente gráfica (figura 8.7)

Se observa que para productividad inferior a 0,3 el simulador paralelo es más rapido, pero a partir de esa productividad el simulador paralelo es más lento.

En cuanto al número de hilos que creamos para asignarles trabajo los

| **tam.paquete** | **Tpo Secuencial** | **Tpo Paralelo** | **Ganancia** |
|---|---|---|---|
| 16 | 6,043 | 5,7045 | 1,059339118 |
| 32 | 9,602 | 7,07 | 1,358132956 |
| 64 | 15,01 | 11,623 | 1,291404973 |
| 128 | 30,176 | 21,626 | 1,39535744 |
| 256 | 73,372 | 46,602 | 1,574438865 |
| 512 | 206,251 | 109,127 | 1,890008889 |
| 1024 | 658,729 | 296,173 | 2,224135894 |
| 2048 | 2299 | 1080,244 | 2,128222883 |

Cuadro 8.2: Tiempos (segundos) en función del tamaño de paquete para productividad de 0,3

resultados han sido los que se muestran en la tabla 8.3.

| **tam.paquete** | **3 hilos** | **30 hilos** | **100 hilos** | (2 hilos) |
|---|---|---|---|---|
| 64 | 13,891 | 14,14 | 16 | 16.745 |
| 128 | 26,482 | 26,747 | 30,642 | 34,166 |
| 256 | 55,19 | 56,667 | 68,517 | 69,719 |
| 512 | 127,242 | 139,771 | 160,413 | 155,254 |

Cuadro 8.3: Tiempos (segundos) en función del número de hilos y del tamaño de paquete (productividad de 0,8)

En la gráfica 8.8 observamos como a medida que se aumenta el número de hilos el tiempo de la simulación va en aumento. Si nos centramos en la simulación con 2 hilos y con 3 hilos (figura 8.9) observamos algo curioso, para 2 hilos se tarda más en simular que para 3 hilos, y eso que estamos simulando en una máquina Dual.

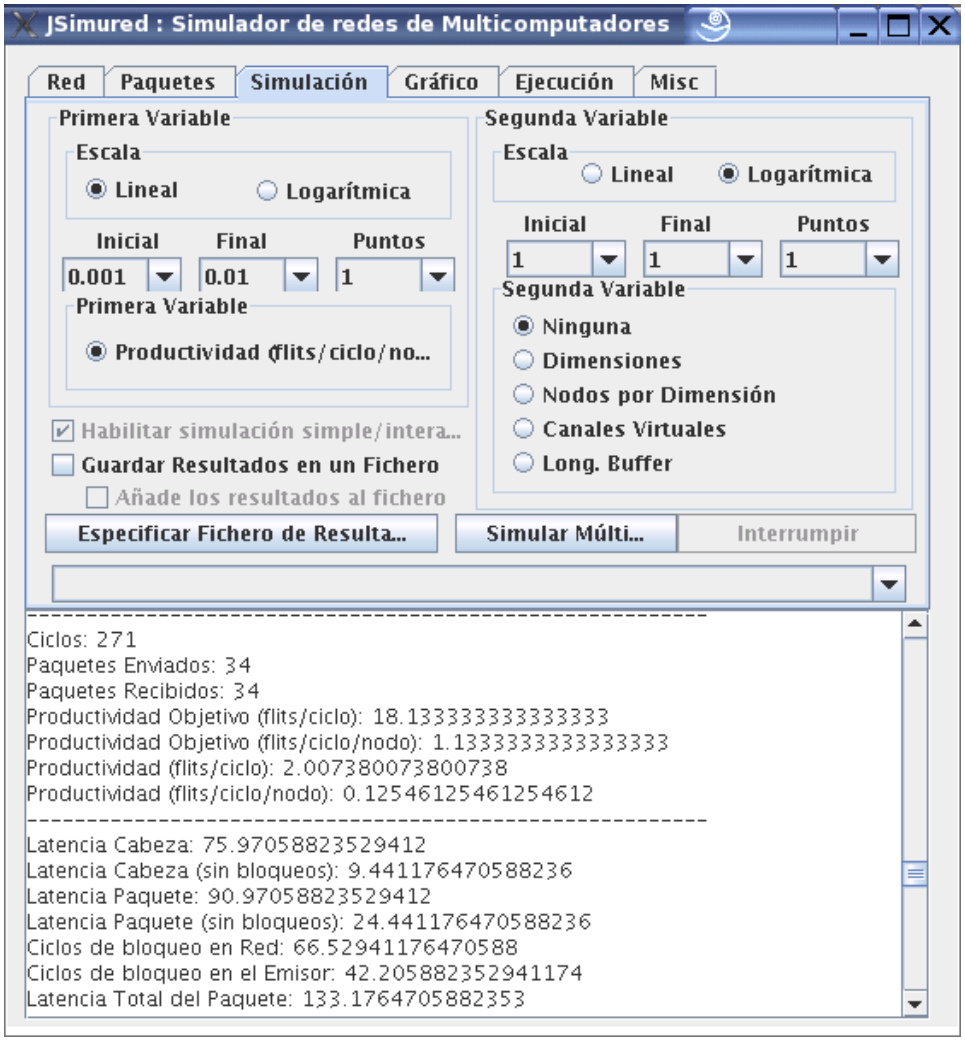


Figura 8.4: Resultado ejecución fichero prueba2.trc en Simulador Paralelo

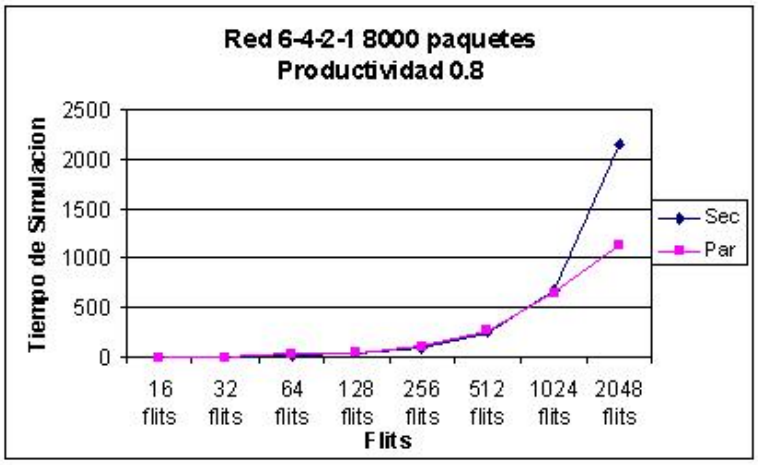


Figura 8.5: Gráfica de tiempos de simulación en función del tamaño del paquete(productividad de 0.8)

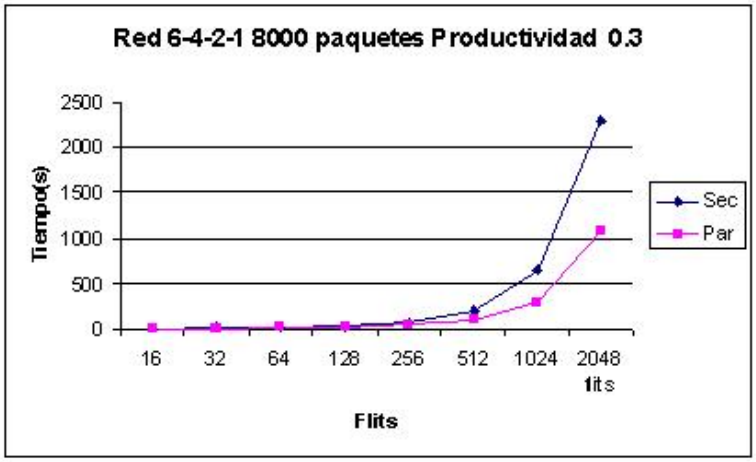


Figura 8.6: Gráfica de tiempos de simulación en función del tamaño del paquete (productividad de 0.3)

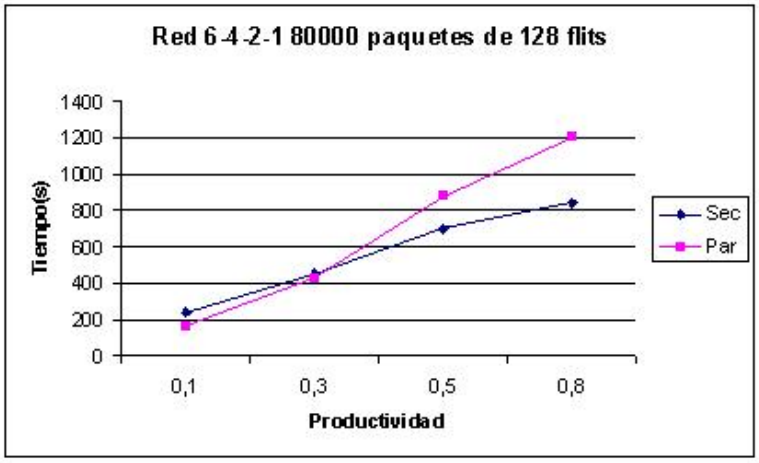


Figura 8.7: Gráfica de tiempos de simulación en función de la productividad

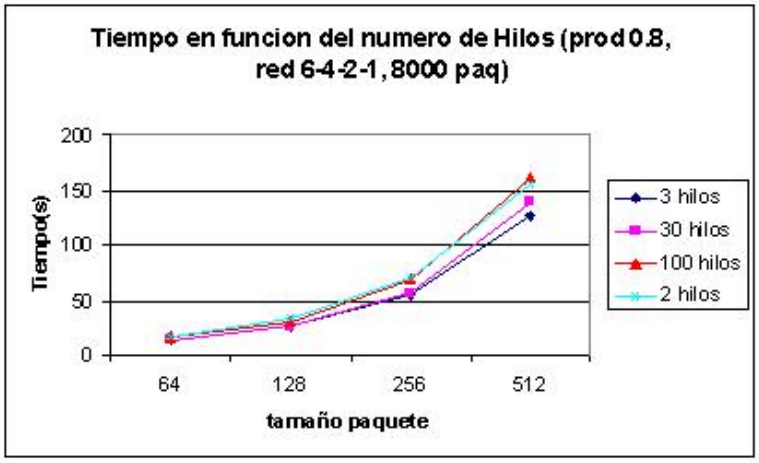


Figura 8.8: Gráfica de tiempos de simulación en función del tamaño de paquete para diferente número de hilos

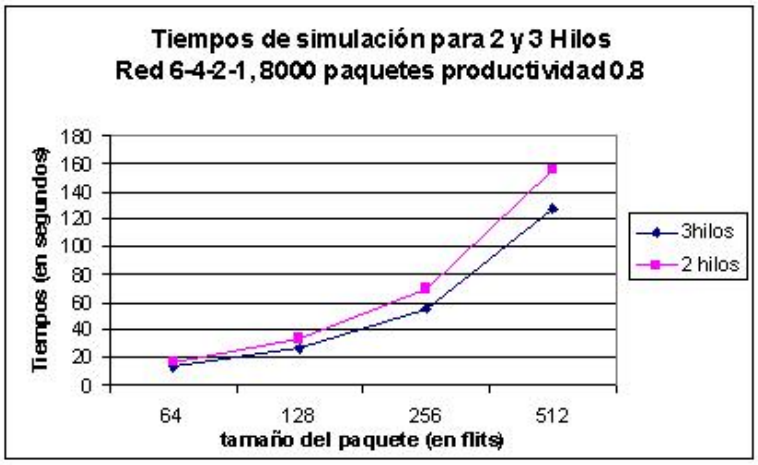


Figura 8.9: Gráfica de tiempos de simulación en función del tamaño de paquete para 2 y 3 hilos

# Capítulo 9

# Conclusiones

## 9.1. Simured C++ vs JSimured secuencial

El principal objetivo a cumplir de este proyecto era conseguir una herramienta multiplataforma. Gracias a la implementación en Java se ha conseguido que funcione en cualquier sistema operativo que tenga soporte en la máquina virtual de java. En nuestra versión secuencial tenemos las mismas características que la versión en C++, pero al tener que correr en la máquina virtual, y no liberar memoria (java utiliza el garbage collector de forma automática) observamos que esta nueva versión es un poco más lenta en ejecución, y sobretodo a la hora de dibujar.

## 9.2. JSimured secuencial vs JSimured paralelo

En cuanto a nuestra versión paralela muchas son las conclusiones a sacar. Una de las principales es que se ha conseguido aumentar la velocidad de simulación hasta el doble. El tiempo de simulación para paquetes grandes disminuye con el tamaño de paquete. Como era de esperar, para obtener un gran rendimiento el código a procesar debe ser **intensivo en cálculo** y no en comunicaciones. Nosotros realizamos un bucle del tamaño del paquete para moverlo un ciclo, si este paquete es pequeño no nos conviene lanzarlo en paralelo, pues las operaciones de comunicación y sincronización van a ser más costosas que los cálculos a realizar en el bucle. Por contra, si este paquete es grande, el bucle va a ser muy costoso de realizar, y si podemos lanzar estos bucles en diferentes procesadores estaremos ahorrando un tiempo que contrarestará al tiempo necesario para sincronización.

Como se ha observado en los resultados, para un paquete superior a 256 flits ya se empiezan a obtener resultados ventajosos para productividades bajas. Para productividades altas el simulador paralelo obtiene mejor rendimiento para paquetes superiores a 1024 flits.

Podemos sacar a partir de los resultados que a baja productividad obtenemos esa ganancia temporal mucho antes. Si miramos como funciona nuestro simulador, la operación más costosa que realizamos en paralelo es el movimiento de un paquete. Al mover este paquete si la productividad es baja el paquete va a moverse sin conocer bloqueos, si la productividad es alta, el paquete no podrá avanzar, y es ahi dónde esta el problema. Al no avanzar la función que realizamos en paralelo es menos costosa, y por lo tanto el simulador secuencial la realiza en menos tiempo.

El simulador construído obtiene una mejor respuesta para paquetes grandes, para paquetes pequeños el simulador secuencial obtiene el resultado en un tiempo inferior ya que no requiere tiempo para sincronizar.

Para obtener resultados óptimos para cualquier tamaño de paquete se tendría que tener una gran independencia en los datos. En una red todo esta interconectado, por lo que la mayoría de parámetros dependen de otros y supone labores de sincronización para actualizar las variables, y por lo tanto perdemos la velocidad que podríamos ganar al lanzar en paralelo.

Otro dato significativo es el número de hilos, a mayor número de hilos, mayor tiempo de simulación. Con el método implementado la sobrecarga por la creación y destrucción de hilos ya no supone un problema. Se crean de una sola vez todos los hilos que van a ejecutar la simulación, y cuando esta termina se destruyen, sin suponer un tiempo extra en cada ciclo. La sobrecarga como hemos observado es debida a la sincronización, y es algo inevitable.

Teniendo en cuenta que estamos simulando en una máquina dual, cabe destacar que el mejor tiempo lo deberiamos encontrar para 2 hilos, pues sólo se podrían procesar 2 tareas simultanéamente. Pero observamos en la gráfica 8.9 que para 3 hilos obtenemos un menor tiempo de simulación. ¿Por qué? La respuesta esta en la sincronización. Al sincronizar 2 hilos no hay otro hilo esperando, y en caso de cambio de tarea no se puede aprovechar ese tiempo para lanzar tareas que no se superpongan. Con 3 hilos ya se puede aprovechar ese tiempo, y siempre habrá un hilo listo en caso de bloqueo de alguno de los otros.

En cuanto al tamaño de la red, una red de tamaño grande evita los bloqueos, y en consecuencia acelera la simulación paralela, eso si, para productividad baja.

En resumen, se ha conseguido realizar un simulador de redes de multicomputadores paralelo que presenta ciertas limitaciones impuestas por la naturaleza del problema. Para simular con paquetes pequeños(del orden de

128 flits) y altas productividades (del orden de 0,8) seguiremos utilizando la versión secuencial. Para paquetes grandes y baja productividad convendrá utilizar el simulador paralelo, siempre que se disponga del hardware necesario!

## 9.3. Trabajo futuro

La aplicación realizada cumple con los objetivos del simulador, pero aún requiere de algunos retoques para ser utilizada plenamente a nivel comercial o educacional. Los mensajes de error son enviados a la salida estándar de error, como trabajo futuro se pretende realizar ventanas de aviso según la gravedad del error.
Además no se comprueba si los datos introducidos por el usuario son coherentes, lo que sería interesante verificar antes de lanzar la ejecución y aparezca el error.

Una buena prestación que también se podría implementar es poder elegir el tipo de arbitraje en el conmutador, actualmente el primero que entra es el primero que sale, y se podrían incorporar nuevos arbitrajes. Incluso se podrían incorporar otros algoritmos de conmutación aparte de la lombriz(WormHole).
Falta por implementar el retraso entre ciclos, que se utiliza para ralentizar la simulación de paquetes visual. Sería una buena idea ya que hoy en día las máquinas son cada vez más potentes y puede ser imposible ver el paso de paquetes en las máquinas actuales si no se habilita un timer para ralentizar.

A nivel de programación paralela, sería interesante probar el simulador en una máquina multiprocesador de más de dos procesadores para comprobar el funcionamiento. En una máquina Dual hemos conseguido aumentar el rendimiento hasta el doble: ¿si dispusieramos de una máquina con 3 procesadores obtendríamos el triple?

# Apéndice A

# Manual de Usuario

## A.1. Requisitos del sistema

La aplicación funciona en cualquier sistema operativo. Requiere un buen procesador aunque ha sido testeado en un Pc Pentium 3 a 800 MHz, y utiliza unos 64 Megas de memoria para su ejecución (ejecución de la *Java Virtual Machine*), aunque puede requerir mucha más memoria según la red que se quiera simular. Para ejecutar este simulador se necesita tener instalada en el sistema la máquina virtual de Java (Java 2 Runtime Environment) en su versión 1.5 o superior. Esta se puede descargar desde [Mic05].

## A.2. Ejecución

Para ejecutar la aplicación existen 2 posibilidades:

1. Ejecutar el archivo .jar
2. Ejecutar simuredJava.class

### A.2.1. Ejecutar el archivo.jar

Para ejecutar un archivo .jar desde Windows simplemente haremos doble-click sobre este archivo y si tenemos el software necesario la aplicación será lanzada sin problemas. Desde Linux deberemos escribir por consola la siguiente línea desde el directorio donde se encuentre nuestro simulador:

```
java -jar simulador.jar
```

Esta instrucción también es válida para Windows desde una ventana de línea de comandos.

### A.2.2. Ejecutar simuredJava.class

Para ejecutar el archivo simuredJava.class deberemos escribir en la ventana de comandos la instrucción:

```
java -cp . simuredJava
```

## A.3. Funcionamiento

Una vez tenemos el interfaz de nuestra aplicación lanzado el manejo es muy sencillo.
En una primer pestaña (figura A.1)llamada Red elegiremos la topología de nuestra red, los retrasos de los dispositivos, el tipo de conmutación (actualmente solo esta implementado el Wormhole) y el encaminamiento.

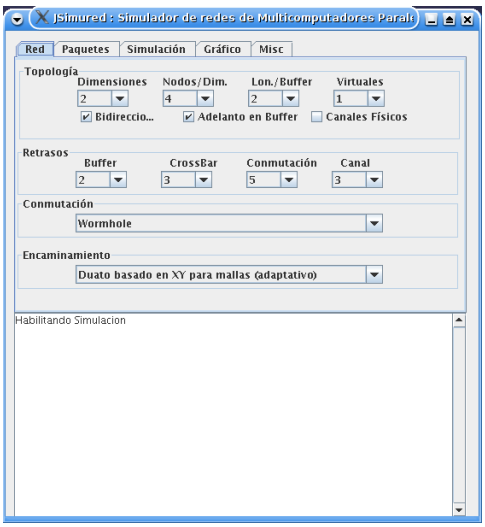


Figura A.1: JSimured: La Pestaña de Red

En una segunda pestaña (figura A.2) elegiremos el tamaño del paquete, y los flits de cabecera.
También se puede cargar un fichero de trazas con los paquetes que vamos a lanzar, o lanzar de forma automática los paquetes elgiendo o bien el número de paquetes, o el número de flits por nodo, y dar una productividad a la red.
En una tercera pestaña (figura A.3) elegiremos (si deseamos una simulación multiple) las variables de simulación junto con los tiempos y la escala. Si deseamos una simulación simple y observar el movimiento de paquetes habilitaremos el check-box 'Habilitar simulación simple/interactiva', esto lanzará la ventana de simulación.

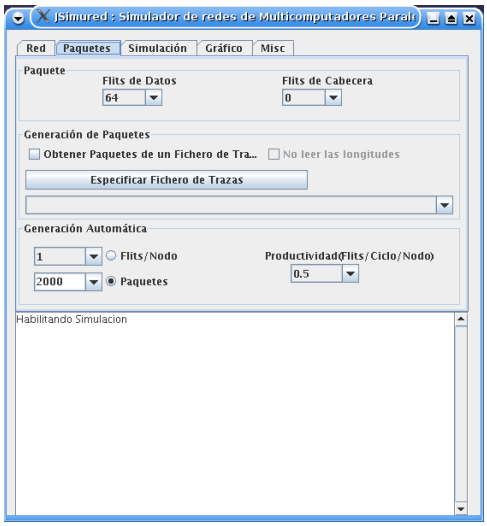


Figura A.2: JSimured: La Pestaña de Paquetes

Si deseamos una simulación múltiple elegiremos los paramétros y mediante el botón 'Simulación múltiple' lanzaremos la simulación.
Los resultados se pueden guardar especificando un fichero y habilitando el check-box 'Guardar resultados en un fichero'.
La pestaña Gráfico (figura A.4)nos permite ver los datos guardados en un fichero en forma de un gráfico. Para ello primer debemos cargar el fichero con los resultados desde el botón 'Cargar fichero CSV'. Una vez cargado elegimos los ejes que deseamos observar y pulsamos el botón 'Recargar(Mostrar Gráfico)'.
Una última pestaña, Misc (figura A.5), permite elegir el idioma.

### A.3.1. La ventana de simulación

La ventana de simulación se muestra en la figura A.6.
Si deseamos observar el movimiento de los paquetes deberemos activar el check-box 'Mostrar evolución'.
Si además queremos ver el número del flit activaremos el check-box 'Mostrar n. flit'.
Si deseamos simular pulsaremos el botón 'Simular', que lanzará la simulación.
Para pausar existe el botón 'Pausar', este cambiará su aspecto a 'Continuar' que pulsaremos para reanudar. Para interrumpir y terminar la simulacíon existe el botón 'Interrumpir'. Interrumpir o cerrar la ventana termina con la simulación actual y muestra los resultados hasta el momento.

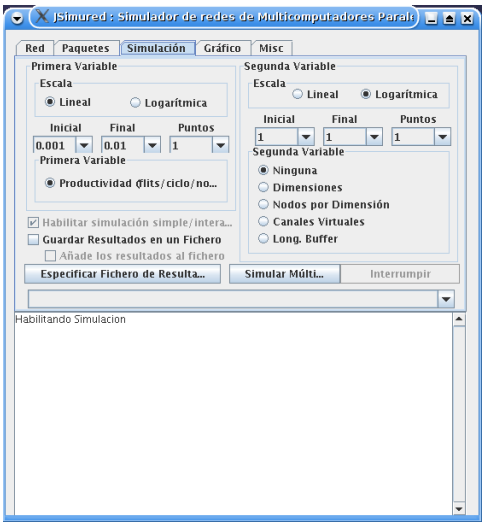


Figura A.3: JSimured: La Pestaña de Simulación

Podemos ver el transcurso de la simulación paso a paso, para pasar un paso pulsaremos el botón 'paso a paso' que ejecutará un sólo ciclo de simulación. Si queremos salir del modo 'paso a paso' pulsaremos sobre 'Continuar'.
El retraso entre ciclos está por desarrollar, por lo que no tiene ningún efecto, por ahora, si se desea ralentizar la aplicación se puede aumentar el tamaño de la ventana.

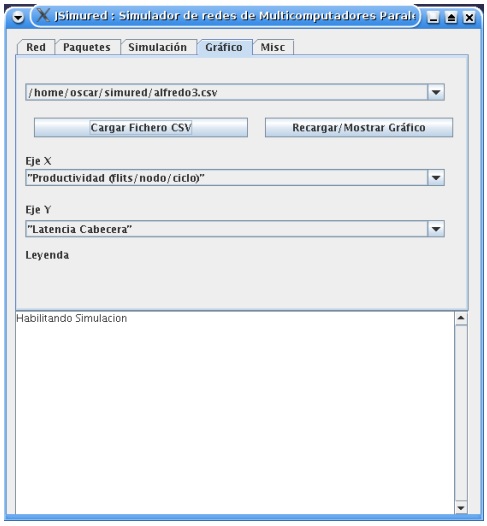


Figura A.4: JSimured: La Pestaña de Gráfico

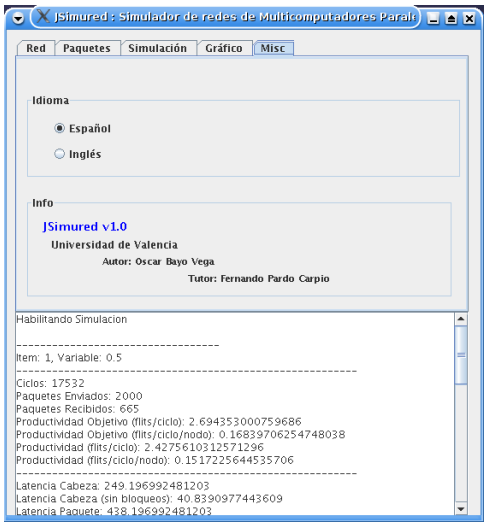


Figura A.5: JSimured: La Pestaña Misc

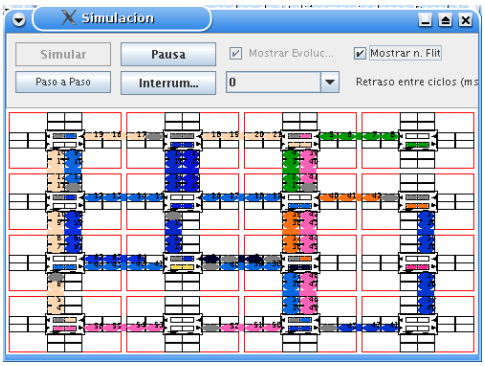


Figura A.6: JSimured: La ventana de simulación

# Apéndice B

# Planificación del Proyecto

## B.1. Planificación del proyecto

Para desarrollar esta aplicación paralela se pensó como solución más acorde con nuestros objetivos el utilizar el lenguaje Java. Las tareas se desglosaron en cuatro fases importantes:

1. **Análisis** del sistema actual. En una primera fase identificaremos el problema central de nuestro proyecto y se decidirá si es posible realizar tal tarea. Se verá que bloques se pueden ejecutar en paralelo y se tendrá una primera visión del problema.

2. **Port a Java** y primera versión secuencial en dicho lenguaje. Mientras se realiza la transformación al lenguaje Java de toda la aplicación se continuará con el análisis, pero esta vez de forma más profunda para comprender el funcionamiento del simulador.

3. **Paralelización** de la versión Java. Esta es la fase más importante. Se intentará desarrollar la paralelización del código para obtener el mejor rendimiento posible, utilizando las técnicas que tenemos a nuestro alcance

4. **Redacción de la memoria**. Esta fase se irá desarrollando a medida que surjan comentarios importantes o resultados relevantes

5. Preparación de la **presentación**. Esta última fase se intentará dar la visión global del proyecto y dar los aspectos más relevantes de nuestro trabajo

Al abarcar un campo de investigación, el diagrama de Gantt (figura B.1) se ha realizado de forma general, sin desglosar en subtareas. Se ha intentado estimar el tiempo de cada fase, pero como es habitual se trata de tiempos orientativos.

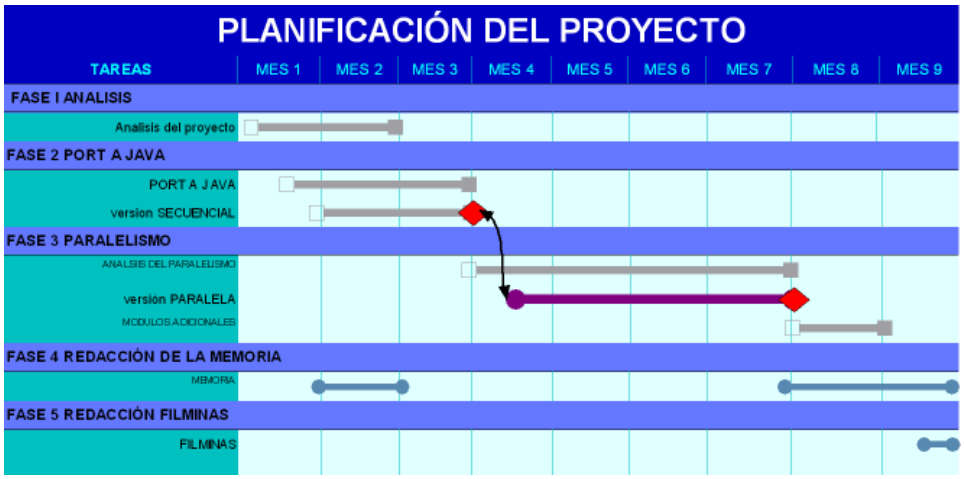


Figura B.1: Diagrama de Gantt del proyecto

El análisis de costes del proyecto no se ha considerado importante al no tratarse de una aplicación comercial, si no de una aplicación de uso educacional, pese a todo se han utilizado herramientas con licencia pública, por lo que el coste de licencias es nulo. El único coste es el hardware utilizado (y las horas invertidas...)

# Apéndice C

# Código del método implementado

```
/*
 * lanzadorPacket.java
 *
 * Created on 23 de diciembre de 2004, 14:22
 */

/**
 *
 * @author  Oscar
 * Esta clase es la que lanza trabajadores para procesar los
 * paquetes
 */
public class lanzadorPacket {

    private trabajadorPacket trabajador[];
    private int nHilos;
    private int nPaquetes;
    private Packet sigLibre;
    private Object sync, syncLibre, syncInicio, syncFin, syncTermina, syncPreparados;
    private int Faltan;
    private int Procesados;
    private boolean fin_simulacion;
    private boolean inicio_ciclo, fin_ciclo;
    private int Preparados;

    /** Creates a new instance of lanzadorPacket */
    public lanzadorPacket(int n) {
        trabajador = new trabajadorPacket[n];
```

```
        nHilos = n;
        nPaquetes = 0;
        sigLibre = null;

        sync = new Object();
        syncLibre = new Object();
        syncInicio = new Object();
        syncFin = new Object();
        syncTermina = new Object();
        syncPreparados = new Object();

        fin_simulacion = false;
        inicio_ciclo = false;
        fin_ciclo = false;

        for (int i=0; i<n; i++)
        {
            trabajador[i] = new trabajadorPacket();

            trabajador[i].start();

        }

    }
    /*
     * Asigna paquetes a los trabajadores
     */
    public void Prepara(Packet p)
    {
        Packet aux = p;

        Faltan = nHilos;
        Preparados = nHilos;

        Procesados = 0;

        for (int i=0; ((i<nHilos) && (aux!=null)); i++)
        {
            trabajador[i].setPriority(trabajador[i].NORM_PRIORITY);
            trabajador[i].Asigna(aux);
            synchronized(syncLibre){
                sigLibre = aux;
```

```
            aux=aux.Next;
        }
        if(aux== null)
            break;
    }
}
/*
 * Avisa de un inicio de ciclo de Red
 */
public void InicioCiclo(int n)
{
    nPaquetes = n;
    // Los hilos esperaran tras terminar de procesar la lista
    synchronized(syncFin)
    {
        fin_ciclo = false;
    }

    // y avisamos de que ya pueden empezar a procesarla
    synchronized(syncInicio){
        inicio_ciclo = true;

        syncInicio.notifyAll();
    }


}
/* Avisa del fin de una simulacion */
public void Termina()
{
    synchronized(syncTermina){
        fin_simulacion = true;
    }

    synchronized(syncInicio)
    {
        inicio_ciclo = true;
        syncInicio.notifyAll();

    }
    synchronized(syncFin)
    {
        fin_ciclo = true;
        syncFin.notifyAll();
```

```
    }
}

/*
 * Espera a que todos los paquetes
 * de un ciclo sean procesados
 */
public void Espera()
{
    // Espera a que los hilos terminen de procesar la lista
    synchronized(sync)
    {
        while(Faltan>0)
            try{
                sync.wait();
            }catch(Exception e){
                System.out.println("Error en wait!");
                System.exit(-1);
            }
    }

    // Los hilos no empezaran hasta que se de la señal
    synchronized(syncInicio){
        inicio_ciclo = false;
    }

    // y permitimos la finalizacion del proceso de la lista de los hilos
    synchronized(syncFin)
    {
        fin_ciclo = true;
        syncFin.notifyAll();
    }

    // y esperamos a que los hilos esten de nuevo listos para trabajar

    synchronized(syncPreparados)
    {
        while(Preparados>0)
            try{
                syncPreparados.wait();
            }catch(Exception e){
                System.out.println("ï¿½Error en wait!");
                System.exit(-1);
```

```
            }
        }
    }

    /*
     * Clase trabajador, es la que ejecuta el runCycle de
     * paquete y sincroniza el acceso
     */
    private class trabajadorPacket extends Thread
    {
        private Packet miPacket;
        private boolean activo;

        public trabajadorPacket() {miPacket = null; activo = false;}
        public trabajadorPacket(Packet p) {miPacket = p;}

        public void Asigna(Packet p) { miPacket = p; }

        public void run()
        {
          Packet aux;
            while(!fin_simulacion)
            {
                // PUNTO DE SINCRONISMO
                ///////////////////////
                synchronized(syncInicio){
                    while(!inicio_ciclo)
                    {
                        try{
                            syncInicio.wait();
                        }catch(Exception e)
                        {System.out.println("Excepcion en syncInicio "+e);}
                    }
                }
                while(miPacket!=null)
                {
                    synchronized(syncLibre)
                        { if (sigLibre != null) sigLibre = sigLibre.Next;}

                  synchronized(miPacket){
                           miPacket.RunCycle();
                  }

                    synchronized(syncLibre)
```

```
                        {miPacket = sigLibre;}
                }

                synchronized(sync)
                {
                    Faltan--;
                    sync.notify();
                }

                // PUNTO DE SINCRONISMO
               ///////////////////////
                synchronized(syncFin){
                    while(!fin_ciclo)
                    {
                            try{
                                syncFin.wait();
                            }catch(Exception e)
                            {System.out.println("Excepcion en syncFin"+e);}

                    }

                }

                synchronized(syncPreparados)
                {
                    if (Preparados>0) Preparados--;
                    syncPreparados.notify();
                }
            }
        }
    }

}
```

# Bibliografía


[ADZM91] Cengiz Alaettinoglu, Klaudia Dussa-Zieger, and Ibrahim Matta. Mars, maryland routing simulator, 1991. ftp://ftp.cs.umd.edu/pub/MaRS (abril 2005).

[But97] David R ButenHof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Cra05] Inc. Cray. Red storm specifications, april 2005. http://www.cray.com/products/programs/red_storm/index.html (abril 2005).

[Dup88] Alex Dupuy. Nest, network simulation testbed, 1988. http://ftp.cs.columbia.edu/nest/ (abril 2005).

[DYN97] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks; An Engineering Approach*. IEEE Computer Society, 1997.

[Eck00] Bruce Eckel. *Thinking in Java, 2nd Edition*. Prentice Hall, 2000.

[Fuj83] Richard Fujimoto. Simon: a simulator of multicomputer networks. *Technical Report: CSD-83-140*, 1983.

[HC03] Cay S. Horstmann and Gary Cornell. *Java 2*. Prentice Hall, Pearson Educación S.A., 2003.

[Huw05] Joseph A. Huwaldt. Joes java homepage, 2005. http://homepage.mac.com/jhuwaldt/java/ (abril 2005).

[IBM05] Corp. IBM. Bluegene specifications, april 2005. http://www.research.ibm.com/bluegene/ (abril 2005).

[Kes97] S. Keshav. Real, realistic and large network simulator, 1997. http://www.cs.cornell.edu/skeshav/real/overview.html (abril 2005).

[Mat00] Norman Matloff. Mulsim, multiprocessor simulator, 2000. http://heather.cs.ucdavis.edu/~matloff/MulSim/MulSimDoc.html (abril 2005).

[MD02] Gregorio Martin and Christian W. Dawson. *El Proyecto Fin de Carrera en Ingeniería Informática, una guía para el estudiante*. Prentice Hall, 2002.

[Mic05] Sun MicroSystems. Java technology, april 2005. http://java.sun.com/ (abril 2005).

[Net05] NetBeans. Netbeans, april 2005. http://www.netbeans.org/ (abril 2005).

[oB96] University of Belgrade. Limes, a multiprocessor simulation tool for pc platforms, 1996. http://galeb.etf.bg.ac.yu/~igi/ (abril 2005).

[ONC99] Andy Ogielski, David Nicol, and Jim Cowie. S3 project, 1998 - 1999. http://www.ssfnet.org/homePage.html (abril 2005).

[oSC95] University of South California. Ns2 network simulator, 1995. http://www.isi.edu/nsnam/ns/ (abril 2005).

[OW04] Scott Oaks and Henry Wongs. *Java Threads, 3rd Edition*. O'Reilly, 2004.

[Par01] Fernando Pardo. Apuntes de ampliación de arquitectura de computadores, 2001. http://www.uv.es/~pardo (abril 2005).

[Par03] Fernando Pardo. Simured: Simulador de multicomputadores, Noviembre 2003. http://simured.uv.es (abril 2005).

[RFDS97] Jennifer Rexford, Wu-chang Feng, James Dolter, and Kang G Shin. Pp-mess-sim: A flexible and extensible simulator for evaluating multicomputer networks. *IEEE Transactions on parallel and distributed systems, vol. 8, nº 1*, 1997.

[Sch05] Christian Schenk. Miktex project, 2005. http://www.miktex.org/ (abril 2005).

[Sta03] William Stallings. *Computer Organization and Architecture, 6rd Edition*. Prentice Hall, 2003.

[Too05] ToolsCenter.org. Toolscenter, 2005. http://www.toolscenter.org/ (abril 2005).

[TPGCSFQA03] J. Tomás Palma, M. Garrido Carrera, F. Sanchez Figueroa, and A. Quesada Arencibia. *Programación Concurrente*. Thomson editores Paraninfo S.A. Spain, 2003.

[VR02] Miguel Angel Vega Rodríguez. Simulador de sistemas de memoria caché en multiprocesadores simétricos, 2002. http://arco.unex.es/smpcache/SMPCacheSpanish.htm (abril 2005).

[yT99] Hung ying Tyan. J-sim, java simulator, 1999. http://www.j-sim.org/ (abril 2005).