Universidad Complutense de Madrid

Facultad de Informática

Diseño e implementación del kernel de xDEVS, versión distribuida

Alumnos:

Guillermo Llorente de la Cita

Luis Lázaro-Carrasco Hernández

Director de proyecto:

José Luis Risco Martín

Trabajo de Fin de Grado

Grado en Ingeniería de Computadores

Curso académico: 2015/2016

Índice general

Palabras clave **5**

Resumen **7**

1. Introducción **9**
- 1.1. Antecedentes . . . 9
  - 1.1.1. Lista de herramientas DEVS . . . 10
- 1.2. Objetivos . . . 12
- 1.3. Plan de trabajo . . . 12

2. Sockets y JavaSockets **15**
- 2.1. Fundamentos . . . 15
- 2.2. Funcionamiento genérico . . . 15
- 2.3. JAVA Sockets . . . 17
- 2.4. Ventajas e Inconvenientes en el uso de Sockets . . . 18

3. El formalismo DEVS **21**
- 3.1. Introducción . . . 21
- 3.2. Definición del formalismo DEVS . . . 22
  - 3.2.1. Modelo DEVS atómico . . . 22
  - 3.2.2. Modelo DEVS acoplado . . . 23
  - 3.2.3. Simulación de modelos DEVS . . . 23

4. Diseño de xDEVS distribuido **25**
- 4.1. Introducción . . . 25
- 4.2. Estructura de diseño . . . 25
- 4.3. Pruebas de rendimiento . . . 31
  - 4.3.1. DEVStone . . . 31
  - 4.3.2. Pruebas en red virtual . . . 38
  - 4.3.3. Pruebas en LAN con Gigabit Ethernet . . . 38
  - 4.3.4. Comparativa según el tipo de red . . . 39
- 4.4. Conclusiones . . . 41

Bibliografía **43**

Agradecimientos **45**

Autorización de difusión **47**

Palabras clave

Palabras clave en Español

- Modelado y simulación
- Sistemas de eventos discretos
- Formalismo DEVS y xDEVS
- Sistemas Distribuidos
- Sockets y JavaSockets
- Benchmark DEVStone

Keywords in English

- Modeling and simulation
- Discrete events system
- DEVS formalism and xDEVS
- Distributed Systems
- Sockets and JavaSockets
- DEVStone benchmark

Resumen

Resumen en Español

Para entender nuestro proyecto, debemos comprender DEVS. Dentro de los formalismos más populares de representación de sistemas de eventos discretos se encuentra DES. En la década de los 70, el matemático Bernard Zeigler propuso un formalismo general para la representación de dichos sistemas. Este formalismo denominado DEVS (*Discrete EVent System Specification*) es el formalismo más general para el tratamiento de DES. DEVS permite representar todos aquellos sistemas cuyo comportamiento pueda describirse mediante una secuencia de eventos discretos. Estos eventos se caracterizan por un tiempo base en el que solo un número de eventos finitos puede ocurrir.

DEVS Modelado y Simulación tiene múltiples implementaciones en varios lenguajes de programación como por ejemplo en Java, C# o C++. Pero surge la necesidad de implementar una plataforma distribuida estable para proporcionar la mecánica de interoperabilidad e integrar modelos DEVS diversificados.

En este proyecto, se nos dará como código base el core de xDEVS en java, aplicado de forma secuencial y paralelizada. Nuestro trabajo será implementar el core de manera distribuida de tal forma que se pueda dividir un sistema DEVS en diversas máquinas.

Para esto hemos utilizado sockets de java para hacer la transmisión de datos lo más eficiente posible. En un principio deberemos especificar el número de máquinas que se conectarán al servidor. Una vez estas se hayan conectado se les enviará el trabajo específico que deberán simular.

Cabe destacar que hay dos formas de dividir un sistema DEVS las cuales están implementadas en nuestro proyecto. La primera es dividirlo en módulos atómicos los cuales son subsistemas indivisibles en un sistema DEVS. Y la segunda es dividir las funciones de todos los subsistemas en grupos y repartirlos entre las máquinas.

En resumen el funcionamiento de nuestro sistema distribuido será comenzar ejecutando el trabajo asignado al primer cliente, una vez finalizado actualizará la información del servidor y este mandara la orden al siguiente y así sucesivamente.

Summary in English

In order to understand our project, it is necessary to understand DEVS. DES is one of the most popular representation formalisms of discrete event systems. In the 70s, the mathematician Bernard Zeigler proposed a general formalism for the representation of such systems. This formalism called DEVS (*Discrete EVent System Specification*) is the most general formalism for the treatment of DES. DEVS allows to represent all those systems whose behaviour can be described by a sequence of discrete events. These events are characterized by a timebase in which only a finite number of events can occur.

DEVS Modeling and Simulation has multiple implementations with several computer languages such as Java, C# or C++. Therefore emerges the need of a stable distributed platform to provide interoperability mechanics and integration of diversified DEVS models.

In this project, we will be given the core of xDEVS in java as code base, applied sequentially and in a parallelized way. Our job is to implement the distributed form of the xDEVS core so that it can divide a DEVS system into different machines.

For this purpose, we have used java sockets for efficient data transmission. At first we specify the number of machines that will connect to the server. Once these have been connected, the server will send them the specific work to be simulated.

There are two ways of dividing a DEVS system which are implemented in our proyect. The first one is dividing into atomic modules which are indivisible subsystems. The second is to divide the functions of all subsystems in groups and spread them between machines.

To summarize the operation of our distributed system will be start executing the work assigned to the first worker, once completed will update the server and these will send the order to the next and so on.

Capítulo 1

Introducción

Mediante la simulación podemos entender el correcto funcionamiento de un sistema real a través de un modelo más sencillo y simplificado. De esta forma obtendremos resultados del sistema real que posteriormente analizaremos y nos ayudará a entender mejor el funcionamiento de este sistema en un determinado ámbito. No obstante los sistemas reales se tornan cada vez más complejos. En los últimos años han surgido conceptos como los sistemas ciberfísicos, sistemas cloud, sistemas sky, internet de las cosas, etc. Esto implica que los sitemas reales son cada vez más complejos y de mayores dimensiones, integrando no solamente componentes tipo máquina, sino integrando el comportamiento humano como parte de los mismos.

En un entorno simulado de un sistema distribuido como en nuestro proyecto hay que tener en cuenta diversas especificaciones de diseño como son la decisión de la existencia de un coordinador y cómo se realizará el intercambio de información entre los clientes que se conectan al servidor. La sincronización en este traspaso de información entre cliente y coordinador o entre los mismos clientes será fundamental para evitar bloqueos de información y garantizar el correcto funcionamiento de nuestro sistema distribuido.

1.1. Antecedentes

DEVS fue introducido en el primer libro de Bernard P. Zeigler *Theory of Modeling and Simulation* en el 1976, desde entonces se han desarrollado programas informáticos que implementan este formalismo aunque no se ha logrado desarrollar una versión distribuida estable, por lo que nuestro propósito es introducir una versión distribuida lo más eficiente posible.

En la década de los 70 el desarrollo de los sistemas distribuidos iba de la mano de las redes locales de alta velocidad. En la actualidad ha habido un mayor desplazamiento de computadoras personales de altas prestaciones, estaciones de trabajo y servidores hacia los sistemas distribuidos en detrimento de los ordenadores centralizados multiusuario. Por ello el desarrollo de software para sistemas distribuidos ha estado en auge durante las últimas décadas. A continuación vamos a nombrar tecnologías enfocadas en el desarrollo de los sistemas distribuidos.

Una tecnología recientemente utilizada en los sistemas distribuidos es **Java RMI** ("*Java Remote Method Invocation*"), el cual forma parte del entorno estándar de ejecución de Java y establece un mecanismo simple para que los servidores se comuniquen en aplicaciones distribuidas. Conceptualmente podríamos decir que RMI es un mecanismo que Java nos ofrece para invocar un método de manera remota. Este software permite a los sistemas compartir los recursos hardware, software y datos así como coordinar sus actividades.

Otra tecnología muy utilizada es **SOAP** ("*Simple Objet Access Protocol*") que es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. SOAP fue creado por Microsoft, IBM y terceros. Es uno de los protocolos más utilizados en los servicios web. Previamente existian los RPC("*Remote Procedure Calls*") usado en diversas tecnologías como **DCOM** y **CORBA** pero SOAP eliminó las complejidades de este tipo de interfaces. Cabe destacar que este protocolo permite la ejecución de un sistema distribuido en diferentes sistemas operativos, como por ejemplo Windows y Linux.

También existe una tecnología llamada **MPI** ("*Message Passing Interface*") el cual es un protocolo que define la sintaxis y la semántica de las funciones cuyo objetivo es el paso de mensajes en un sistema distribuido. El paso de mensajes es una técnica empleada en programación concurrente para aportar sincronización entre procesos que se ejecutan en diferentes máquinas y permitir la exclusión mutua de manera similar a como funcionaría un semáforo. En este proyecto usaremos una técnica similar utilizando *Sockets* y mensajes predefinidos.

1.1.1. Lista de herramientas DEVS

En la última década se han desarrollado diversos motores de simulación DEVS. Casi todos ofrecen una API para definir nuevos modelos usando un lenguaje de alto nivel y a excepción de uno o dos, la mayoría estan ligados a un lenguaje de programación. Además solo unos pocos contienen una interfaz de usuario gráfica para especificar los modelos. A contunuación mostraremos algunos de los más conocidos y utilizados.

DEVSJAVA

DEVSJAVA ha sido desarrollado por Bernard P. Zeigler y Hessam Sarjoughian (Universidad de Arizona, U.S.A.). Esta escrito en Java y soporta simulación virtual, en tiempo real, secuencial y paralela. En este simulador los modelos se especifican mediante una API y posee una interfaz de usuario. Este es uno de los más conocidos en la comunidad DEVS.

DEVS-Suite and COSMOS

DEVS-Suite es un simulador basado en el formalismo paralelo de DEVS diseñado con técnicas para visualizar la simulación mostrando la estructura de los modelos, animandolos y viendo los cambios en el tiempo. CoSMoS ("*Component-Based System Modeling and Simulation*") es un

framework orientado al desarrollo de modelos, configuración y recopilación de datos. La plataforma CoSMoS especifica los modelos con el formalismo DEVS mediante un esquema *XML*. Estos dos estan ligados y soportan simulación de modelos DEVS en paralelo.

CD++

CD++ ha sido desarrollado por Gabriel Wainer y sus estudiantes (Universidad de Carleton, Canadá; Universidad de Buenos Aires, Argentina). Escrito en C++, permite especificar modelos DEVS gráficamente a través de una API llamada CD++Builder. CD++ soporta simulación virtual, en tiempo real, secuencial, paralela y también distribuida.

xDEVS

xDEVS es el simulador que vamos a ampliar. Ha sido desarrollado recientemente por José L. Risco Martín y Saurabh Mittal. Está desarrollado en Java y soporta simulación virtual, en tiempo real, secuencial y paralela. También permite ejecutar modelos aplanados lo cual cambia la arquitectura del modelo y lo reduce a modelos atómicos para aumentar la eficiencia. También permite realizar co-simulaciones hardware/software. Esta diseñado a través del paradigma orientado a objetos. Además se encuentra bajo la licencia GPL lo cual facilita el desarrollo del mismo.

PyDEVS

PythonDEVS fue desarrollado como una herramienta para enseñar DEVS y también como base de investigación sobre detalles y variantes de DEVS. Cabe destacar que también existe una versión que permite simulaciones en tiempo real (*RT-PyDEVS*) desarrollada por Spencer Borland (Universidad McGill, Canadá). En PyDEVS los modelos son especificados a través de una API, permitiendo simulaciones virtuales y en tiempo real.

aDEVS

aDEVS es una librería C++ creada para modelar simulaciones de eventos discretos basadas en los formalismos DEVS Paralelo y DEVS dinámico. Desarrollada por Jim Nutaro, permite la implementación de simulaciones secuenciales y paralelas a través de su API. Esta plataforma es la que muestra mayor rendimiento.

JAMES-II

Desarrollado por la universidad de Rostock, JAMES II es una plataforma de simulación que soporta diferentes formalismos (Cellular Automata, Attributed Pi Calculus, ML-DEVS, ML-Rules...). Provee de una API para especificar modelos y una interfaz de usuario para observar los resultados de la simulación. Este motor soporta simulación secuencial y paralela.

DEVSim++

Desarrollado por el equipo de Tag Gon Kim en el Instituto Avanzado de Ciencia y Tecnología de Corea (KAIST), es un motor de simulación desarrollado en C++ usado para simulaciones extensas. La plataforma provee semanticas de especificación modular y jerarquica a nivel de sistema de eventos discretos como sistemas multinodo. Se enfoca en la interoperabilidad de modelos de simulación.

1.2. Objetivos

Los objetivos más destacados en la realización de este proyecto son:

1. Comprender el formalismo DEVS y su aplicación en el programa de simulación xDEVS.
2. Estudiar el diseño de plataformas distribuidas usando Sockets, sincronización y serialización.
3. Crear una o varias arquitecturas para aplicarlas en el desarrollo de la versión distribuida.
4. Extender la herramienta de simulación xDEVS para que soporte la simulación distribuida.
5. Desarrollar la arquitectura satisfactoriamente de la manera más eficiente posible.
6. Afianzar conocimientos de asignaturas como Programación de Sistemas Distribuidos y Tecnología de la Programación.
7. Realizar pruebas de rendimiento del sistema distribuido comparandolo con los modelos paralelo y secuencial.
8. Aprender a editar y crear textos científicos con LaTeX para la elaboración de esta memoria.

1.3. Plan de trabajo

La organización del proyecto ha consistido en reuniones quincenales con el director en las que se han debatido posibles restricciones del programa final, decisiones de diseño, cambios en la arquitectura y problemas que fueron sucediendo a lo largo del curso. A continuación mostramos las principales tareas que llevamos a cabo.

Tarea 1 (Objetivo 1) *Comprender el formalismo DEVS y su aplicación en el programa de simulación xDEVS.* El principio del proyecto consistió en estudiar el manual del formalismo DEVS y su aplicación. Para ello leímos el manual y descargamos el código en java para ejecutarlo y hacer pruebas. Posteriormente organizamos reuniones con el director para resolver las dudas pertinentes, esta tarea consistió en los primeros meses de proyecto (meses 1 a 3).

Tarea 2 (Objetivo 2) *Estudiar el diseño de plataformas distribuidas usando Sockets, sincronización y serialización.* Para crear un diseño consistente de la aplicación distribuida tuvimos que

estudiar la parte de la comunicación a través de Sockets ya que concluímos que era la opción más eficiente. Por lo que tuvimos que buscar información y ejemplos en internet que nos ayudarían a comprender su funcionamiento. Posteriormente realizamos pruebas de envío y recepción de datos. El objetivo se alcanzó en el segundo mes de proyecto (meses 2 a 3).

Tarea 3 (Objetivo 3,4 y 5) *Crear una o varias arquitecturas para aplicarlas en el desarrollo de la versión distribuida, extender la herramienta de simulación xDEVS con la arquitectura diseñada y realizarlo de la forma más eficiente posible.* Esta tarea consiste en crear un modelo de distribución del programa y un protocolo de comunicación así como aplicar los conocimientos de DEVS adquiridos para ello, además de llevar a cabo las fases de desarrollo, depuración y comprobacion de errores. Durante el proyecto acabamos creando tres modelos diferentes ya que algunos de ellos tuvieron demasiados fallos y complicaciones para proseguir con ellos. Por lo que la tarea se llevo a cabo durante todo el proyecto (meses 2 a 10).

Tarea 4 (Objetivo 7) *Realizar pruebas de rendimiento del sistema distribuido comparándolo con los modelos paralelo y secuencial.* Para llevar a cabo esta tarea tuvimos que estudiar el benchmark DEVStone para aprender a configurarlo correctamente y poder adaptarlo a las pruebas de nuestro protecto lo que nos llevó aproximadamente un mes (meses 7 a 8). Una vez funcionando el sistema distribuido, llevamos a cabo las pruebas de rendimiento en diferentes plataformas durante el último mes de proyecto (10º mes).

Capítulo 2

Sockets y JavaSockets

2.1. Fundamentos

Socket designa un concepto abstracto por el cual dos programas pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. El término socket es también usado como el nombre de una interfaz de programación de aplicaciones (API) para la familia de protocolos de Internet TCP/IP, provista usualmente por el sistema operativo.

Los sockets de Internet constituyen el mecanismo para la entrega de paquetes de datos provenientes de la tarjeta de red a los procesos o hilos apropiados. Un socket queda definido por dos direcciones IP y dos puertos, local y remoto y un protocolo de transporte.

A través de un puerto que este disponible pueden usar una serie de primitivas para establecer el punto de comunicación, de esta forma podrán escuchar en él, para leer, escribir, publicar información en el, y finalmente para desconectarse todo ello de manera bidireccional tanto para el servidor como para el cliente.

2.2. Funcionamiento genérico

El funcionamiento de un socket se basa en su respuesta en un puerto específico en un servidor que se ejecuta sobre una computadora determinada. El servidor se encarga de esperar la petición del cliente escuchando a través del puerto. Para realizar una petición de conexión el cliente debe conocer tanto la ip o la direccion del servidor como el puerto por el que está escuchando (figura 2.2).

Si todo funciona correctamente el servidor aceptará la conexión y obtendrá un nuevo socket el cual gestionará en un hilo secundario y le asignará un puerto diferente como se ve en la figura 2.3. Esto se debe a que si se conectan más clientes necesitará el puerto original libre. Por otro lado seguirá atendiendo las peticiones de conexión de nuevos clientes mientras atiende las necesidades del cliente que ya se conectó en el anterior socket. Por ejemplo en nuestro proyecto el conjunto no empieza a trabajar hasta que todos los clientes se han conectado.

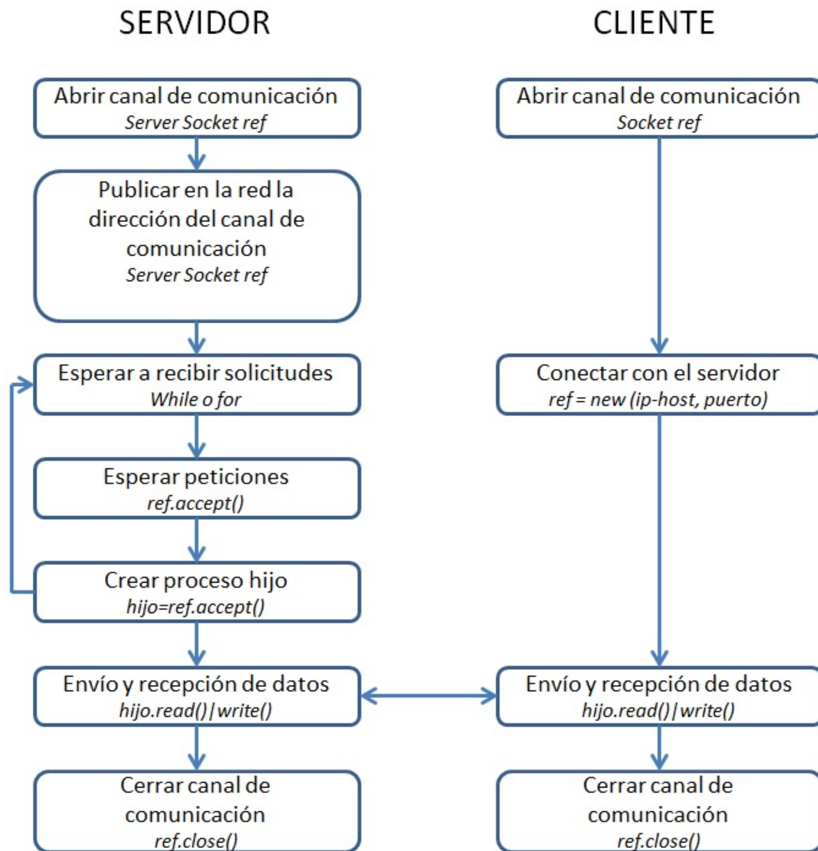

Figura 2.1: El servidor empieza abriendo el Socket en un puerto especifico y espera a la conexión de los clientes. A continuación los clientes se conectan con el servidor a través de la dirección ip y puerto designados por el servidor.

Figura 2.2: Petición de conexión por parte del cliente conociendo la ip y el puerto por el que escucha el servidor.

En el lado del cliente, si la conexión es aceptada, se creará un socket de forma satisfactoria que podrá utilizar para comunicarse con el servidor. El cliente asignará un número de puerto local a la maquina en la cual está siendo ejecutado de manera que no utilizará el número de puerto usado para realizar la petición al servidor. En este momento el cliente y el servidor podrán comunicarse escribiendo o leyendo en o desde sus respectivos sockets.

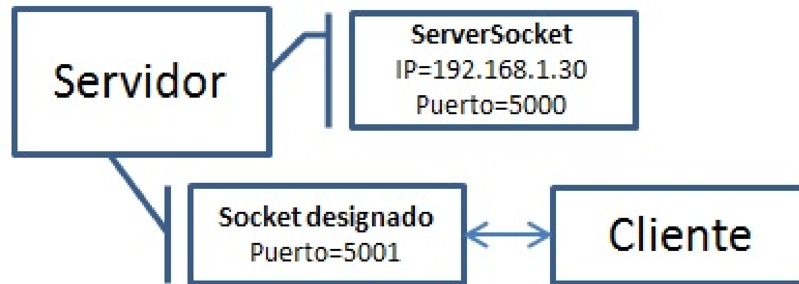

Figura 2.3: Una vez que el cliente se ha conectado se le asigna un puerto diferente para dejar libre el puerto principal para posteriores conexiones de otros clientes.

2.3. JAVA Sockets

Introduccion

El paquete java.net de la plataforma Java proporciona una clase Socket, la cual implementa una de las partes de la comunicación bidireccional entre un programa Java y otro programa en la red. La clase Socket se sitúa en la parte más alta de una implementación dependiente de la plataforma, ocultando los detalles de cualquier sistema particular al programa Java. Usando la clase java.net.Socket en lugar de utilizar código nativo de la plataforma, los programas Java pueden comunicarse a través de la red de una forma totalmente independiente de la plataforma.

De forma adicional, java.net incluye la clase ServerSocket, la cual implementa un socket el cual los servidores pueden utilizar para escuchar y aceptar peticiones de conexión de clientes. Nuestro objetivo será conocer cómo utilizar las clases Socket y ServerSocket.

Por otra parte, si intentamos conectar a través de la Web, la clase URL y clases relacionadas (URLConnection, URLEncoder) son probablemente más apropiadas que las clases de sockets. Pero de hecho, las clases URL no son más que una conexión a un nivel más alto a la Web y utlilizan como parte de su implementación interna los sockets.

Modelo de comunicación con Java

El servidor establece un puerto y espera durante un cierto tiempo (*timeout* segundos) o espera hasta que se conecten un cierto número de clientes. Cuando el cliente solicite una conexión, el servidor abrirá la conexión *socket* con el método *accept()*.

El cliente establecerá una conexión con la máquina host a través del puerto asignado por el servidor. Y posteriormente se comunicarán con manejadores InputStream y OutputStream (figura 2.4).

Para poder enviar objetos por el canal de comunicación es necesario serializarlos. La serialización de un objeto consiste en obtener una secuencia de bytes que represente el estado de dicho objeto. Esta secuencia puede utilizarse de varias formas. Por ejemplo puede enviarse a través de la red, guardarse en un fichero para su uso posterior y utilizarse para recomponer el objeto original. El estado de un objeto viene dado por el estado de sus campos. Por lo que serializar un objeto es guardar ese estado. Si el objeto a serializar tiene campos que a su vez son objetos, estos también tendrán que ser serializados y así sucesivamente.

Un objeto serializable es un objeto que se puede convertir en una secuencia de bytes. Para que un objeto sea serializable, debe implementar la interfaz java.io.Serializable. Esta interfaz no define ningún método. Simplemente se usa para marcar aquellas clases cuyas instancias pueden ser convertidas a secuencias de bytes y posteriormente reconstruídas.

Para serializar un objeto no hay más que hacer que este implemente la clase Serializable. Aunque Java lo pueda hacer de forma automática también es conveniente asignarle una ID ya que en un modelo distribuido con múltiples clientes, deben tener la misma ID para cada clase.

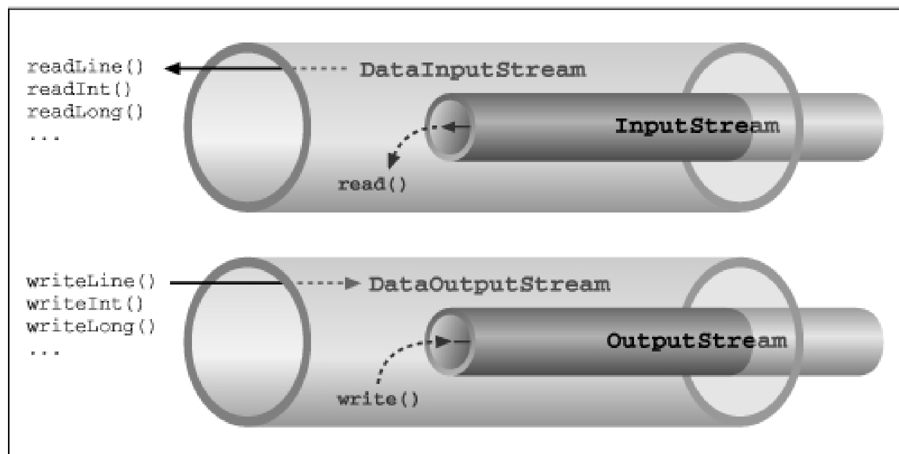

Figura 2.4: En el canal de comunicación se pueden escribir tanto tipos primitivos como clases de java siempre y cuando esten serializadas.

2.4. Ventajas e Inconvenientes en el uso de Sockets

Las tablas 2.1 y 2.2 resumen las ventajas e inconvenientes del uso de sockets en computación distribuida. Hay que tener en cuenta que el rendimiento del programa distribuido es mucho mayor al no utilizar capas auxiliares tales como servicios **SOAP**, **REST**, etc. Cabe destacar también que Java soporta sockets de forma nativa, por lo que las bibliotecas especializadas no son necesarias.

Cuando se utilizan sockets se sabe exactamente qué datos van a ser enviados y de qué tipo son estos, por lo tanto con su uso se evita enviar información adicional tanto para minimizar la sobrecarga en el traspaso de información como para ganar una mayor flexibilidad y control sobre los datos que se envían. Para enviar adecuadamente datos estructurados es necesario serializar el objeto que se desea enviar, esto quiere decir convertirlo en un conjunto de bytes para que posteriormente después de su envío pueda ser reconstruido al otro lado de la red.

En nuestro caso hemos utilizado los sockets en Java en un sistema operativo Windows pero estos pueden ser usados incluyendo su correspondiente librería tanto en otros lenguajes de programación tales como C, C++, Ruby o Python como en cualquier sistema Unix, esto garantiza una gran disponibilidad en el uso de los mismos. Por último cabe destacar que su uso no es excesivamente complicado si se conocen las técnicas de sincronización adecuadas.

Ventajas	**Explicación**
Rendimiento y mínima sobrecarga (control total sobre los datos enviados)	Esto se debe a que no se envía ninguna información adicional a parte de los datos.
Máxima flexibilidad a la hora de desarrollar aplicaciones (control total sobre los datos enviados).	Se puede enviar cualquier tipo de dato ya sean objetos o tipos primitivos.
Disponibilidad: interfaz socket disponible en múltiples sistemas operativos y lenguajes de programación.	Los sockets estan disponibles en multitud de lenguajes de programación y también pueden comunicar sistemas operativos diferentes.

Cuadro 2.1: Ventajas en el uso de sockets.

<table>
<tr><th>Desventajas</th><th>Explicación</th></tr>
<tr><td>Necesidad de realizar un manejo explícito de los datos (formateo, codificación, etc).
▪ Requiere acordar/negociar los formatos de transmisión.
▪ Requiere aplanar/desaplanar los datos estructurados antes de ser enviados/recibidos (marshaling o serialización).</td><td>Para poder enviar objetos se requiere una serialización previa de los mismos, tanto por parte del servidor como del cliente. Si el cliente no posee esta serialización el servidor debe enviarsela al cliente y este debe procesarla.</td></tr>
<tr><td>Complejidad y dificultad de programación.</td><td>La complicación en el desarrollo de aplicaciones usando sockets es la parte de sincronización entre servidor y múltiples clientes.</td></tr>
</table>

Cuadro 2.2: Desventajas en el uso de sockets.

Capítulo 3

El formalismo DEVS

3.1. Introducción

Durante las últimas décadas, la rápida evolución de la tecnología ha producido una proliferación de nuevos sistemas dinámicos. Ejemplos de ellos son las redes de computadoras, sistemas de producción automatizados y sistemas en general de comando, de control, de comunicaciones y de información. Todas las actividades en estos sistemas se deben a la ocurrencia asincrónica de *eventos discretos*, algunos controlados (como por ejemplo la pulsación de un botón) y otros no (como la generación de una señal en una máquina). Esta característica es la que lleva a definir el término de *Sistemas de Eventos Discretos*.

Las herramientas matemáticas que hoy disponemos fueron desarrolladas para modelar y analizar los procesos *conducidos por el tiempo* que generalmente uno encuentra en la naturaleza. El proceso de adaptar estas herramientas y desarrollar nuevas para los sistemas *conducidos por eventos* tiene solo unos pocos años. Por este motivo, encontramos en la teoría de los sistemas de eventos discretos no sólo una serie de herramientas para resolver problemas de modelización, simulación y análisis de sistemas, sino también un campo inexplorado para el desarrollo de nuevas técnicas y teorías debido a la cantidad de problemas aún abiertos en el área.

En la década de los 70, el matemático Bernard Zeigler propuso un formalismo general para la representación de sistemas. Este formalismo denominado DEVS es el formalismo más generico para el tratamiento de sistemas *conducidos por eventos*.

En el formalismo DEVS, un sistema está formado por subsistemas, los cuales se corresponden con modelos indivisibles llamados *atómicos*. Estos en su conjunto pueden formar modelos *acoplados* que en resumen son modelos atómicos y/o acoplados interconectados entre sí.

Bajo un punto de vista general, un modelo DEVS está caracterizado por generar eventos de salida Y, en relación con el estado en el que se encuentre S y las entradas recibidas X, cada cierto tiempo.

3.2. Definición del formalismo DEVS

DEVS define el comportamiento de un sistema real, este comportamiento es descrito utilizando señales de entrada y salida, y transiciones entre estados concretos asemejandose a una máquina de Moore. La descripción del sistema puede variar según su naturaleza.

3.2.1. Modelo DEVS atómico

Un modelo DEVS atómico se define como una estructura:

M = $\{X.S$,Y,δ_{int},δ_{ext},λ,$ta\}$ donde:

- X: conjunto de valores de las entradas
- S: conjunto de valores de los estados
- Y: conjunto de valores de las salidas
- δ_{int}: función de transición interna: δ_{int}:$S \rightarrow S$
- δ_{int}: función de transición externa: δ_{ext}:$Q \times X \rightarrow S$, donde

  $$Q = \{(s, e)|s \in S, 0 \leq e \leq ta(s)\}$$

  es el conjunto de estado total y e es el tiempo transcurrido desde la última transición
- λ : función de salida: $\lambda : S \rightarrow Y$
- ta : es la función de avance de tiempo: ta : $S \rightarrow R_0^+ \infty$

La interpretación de estos elementos es la siguiente: En algún instante de tiempo el sistema llega al estado s. Si no ocurre ningún evento externo el sistema permanecerá en dicho estado durante un tiempo ta(s). Cuando el tiempo transcurrido, e, supera a ta(s), el sistema produce un evento de salida con valor $\lambda(s)$ y cambia al estado $s` = \delta_{int}(s)$ que es el estado de transición interna, en el que sabemos que no ha ocurrido ningún evento externo durante el tiempo pasado.

Si antes que ocurra la transición interna ocurriese un evento externo con valor x mientras el sistema está en el estado total (s,e), siendo $e \leq ta(s)$, el sistema cambia al estado δ_{ext}(s,e,x), pero en este caso no se produce ningún evento de salida.

Como puede verse en la definición, ta(s) puede ser un número real, inclusive 0 ó ∞. Si este tiempo es 0, diremos que s es un estado *transitorio*. Si en cambio, ta(s) es ∞, si el sistema llega a ese estado, permanecerá en el para siempre, a menos que ocurra algún evento externo que le haga ejecutar su $\delta_{ext}(s)$. En este caso nombraremos s como estado *pasivo*.

3.2.2. Modelo DEVS acoplado

El modelo acoplado no es más que un grupo de modelos atómicos y/o acoplados cuyas entradas y salidas han sido interconectadas entre ellos, por supuesto pueden quedar entradas y salidas sin conectar y que quedarían externas al sistema. Cada modelo interno tiene un identificador y si este es acoplado también un conjunto de componentes.

N = $\{X$,Y,M,EIC,EOC,$IC\}$ donde:

- X : conjunto de pares $(puerto, valor)$ de los eventos de entrada
- Y : conjunto de pares $(puerto, valor)$ de los eventos de salida
- M : nombres de los componentes de cada modelo (atómicos y/o acoplados)
- EIC : función de acoplamiento de las entradas externas al modelo
- IC : función de traslación de las salidas de M_i a las entradas de M_j
- EOC : función de acoplamiento de las salidas externas al modelo

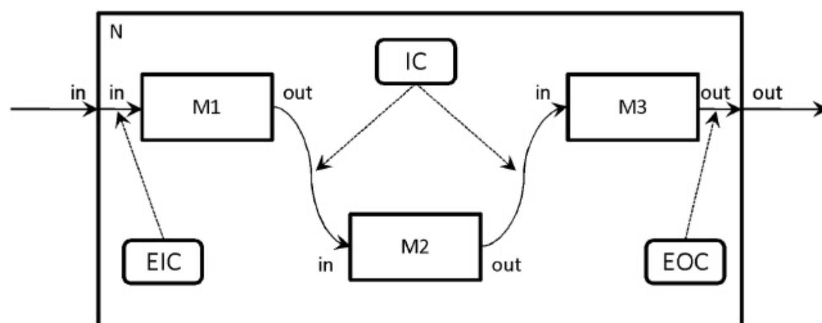

Figura 3.1: Ejemplo de modelo acoplado siendo cada M un componente de tipo átomico o acoplado, EIC y EOC las entradas y las salidas respectivamente e IC las conexiones internas.

3.2.3. Simulación de modelos DEVS

DEVS trata al modelo y al simulador como dos elementos diferentes. Cada componente tiene un protocolo de simulación y cada protocolo es impuesto por un proceso, el cual puede ser un coordinador o un simulador como podemos ver en la figura 3.2.

Primero se construye el modelo acoplado y se crea un coordinador junto con el modelo. A continuación el coordinador divide el modelo acoplado en componentes atómicos y/o acoplados. En la figura 3.2 en el caso de que un componente fuera un modelo acoplado entonces su simulador debería ser un coordinador.

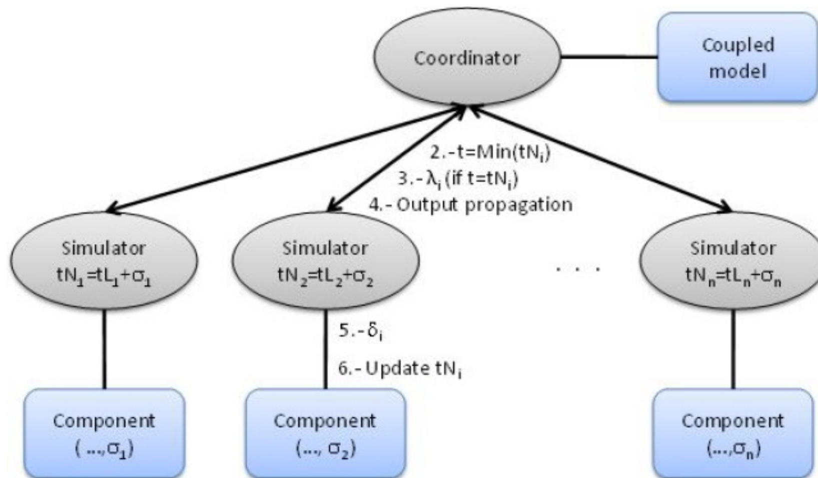

Figura 3.2: Estructura de simulación: cada componente atómico tiene su simulador, pero este en caso de ser acoplado tendría un coordinador. Podemos decir que cada atómico tiene su simulador propio y cada acoplado tiene un coordinador.

Una vez con el sistema construido, el coordinador raíz se encargará de encontrar el modelo atómico M_d que según su función ta() y el tiempo transcurrido, deba ser el próximo en ejecutar su función δ_{int} (se ejecutará primero el que tenga un ta() menor). Después se le sumará al tiempo transcurrido el tiempo t_n de la transición δ_{int} y se ejecutarán las funciones λ y δ_{int} de M_d.

Tras ejecutar la función λ se propagarán las nuevas salidas a todos los componentes conectados con ellas y se ejecutarán las funciones δ_{ext} de los mismos. A continuación se limpiarían los puertos X/Y y se volvería a buscar el atómico que le toca realizar su δ_{int}. Todo esto es un bucle que según los parámetros del usuario, simula durante un número de iteraciones, o por otro lado durante un tiempo específico.

Capítulo 4

Diseño de xDEVS distribuido

4.1. Introducción

Para crear xDEVS distribuido hemos optado por el esquema de diseño más sencillo y robusto para evitar posibles problemas que nos surgieron con otras arquitecturas diferentes. Este se basa en utilizar un servidor como centro de conexión entre los clientes de manera que este haga de coordinador de xDEVS. El servidor enviará un mensaje al cliente que tenga el simulador que le toque ejecutar su transición δ_{int} y este le devolverá el modelo actualizado. En el caso de que se produzca una transición del tipo δ_{ext}, sucederá lo mismo y el servidor siempre mantendrá el modelo actualizado.

4.2. Estructura de diseño

Para comenzar hemos creado un nuevo paquete llamado *xdevs.core.simulation.distributed* en el que hemos implementado las clases necesarias para llevar a cabo nuestra idea.

Como vemos en el diagrama UML (figura 4.1) tenemos un servidor el cual después de iniciarse, espera la conexión de los clientes, y crea una hebra ClientHanlder por cada uno de ellos. En el lado del cliente se crea una hebra ServerHandler la cual se encarga de la comunicación con el servidor. Entre los dos manejadores se envían mensajes del tipo enumerado Orders. El primer mensaje que se manda es "*el trabajo*"que tiene que realizar cada cliente con la orden *sendLamWork* y *sendDelWork*.

Por otro lado tenemos el CoordinatorDistributed el cual es capaz de ejecutar una sola transición por cliente, dependiendo del mensaje que le haya enviado el servidor al principio de la simulación. Cuando el ta() mínimo sea superado por el momento actual, el servidor enviará una orden runLambda a todos los clientes de tal forma que el cliente en estado de transición (δ_{int}) le enviará el modelo actualizado al servidor.

A continuación el servidor propagará las salidas de la transición previa y se encargará de enviar una orden runDelta junto con el modelo actualizado a todos los clientes. Y en caso de que el cliente posea una entrada con algún valor realizará su transición externa δ_{ext}.

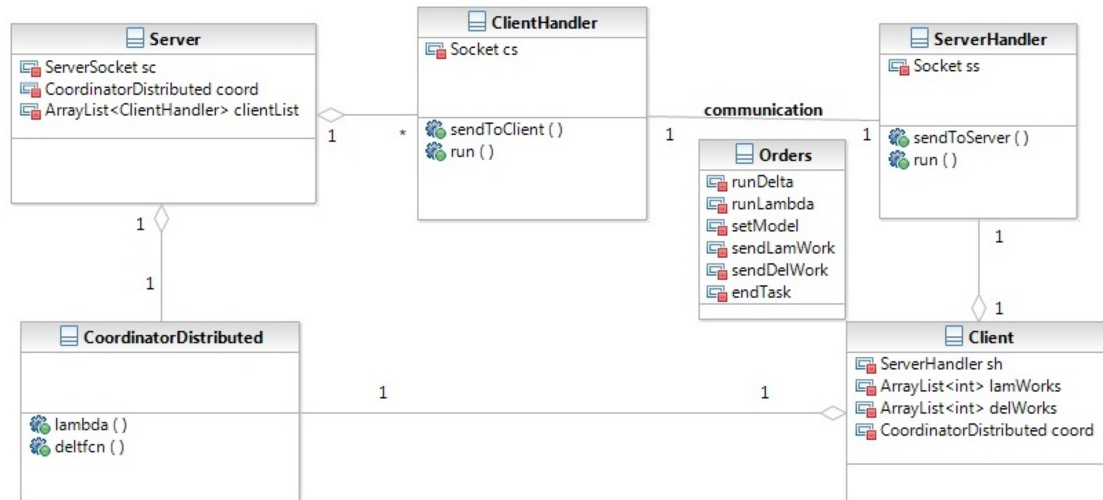

Figura 4.1: Arquitectura del sistema distribuido: el servidor tiene una hebra ClientHandler por cada cliente y el cliente tiene una sola hebra ServerHandler.

Métodos de conexión con los clientes y de simulación respectivamente:

Server.java

```
public void connect(){
  for(int i=0;i<numClients;i++){
    so = new Socket();      //creamos el socket servidor
    try {
      so = sc.accept(); //escuchamos hasta que se conecte un cliente
      System.out.println("Client connected " + so.toString());
    } catch (IOException e) {
      //nothing
    }
    clientList.add(new ClientHandler(so,this)); //anyadimos el cliente
  }
}

public void simulate(long numIterations) {
  logger.fine("START SIMULATION");
  coord.getClock().setTime(tN);
  long counter;
  for (counter = 1; counter < numIterations
    && coord.getClock().getTime() < Constants.INFINITY; counter++) {
    lambda();
    propagateOutput();
    propagateInput();
    deltfcn();
    coord.clear();
    coord.getClock().setTime(coord.getTN());
```

```
26     }
27     for(ClientHandler c : clientList){
28       c.sendToClient(Orders.endTask);
29       c.endTask();
30     }
31     try {
32       sc.close(); //Cerramos el socket servidor
33     } catch (IOException e) {
34     }
35     coord.exit();
36   }
37   public synchronized void lambda() {
38     for(ClientHandler c : clientList){
39       c.sendToClient(Orders.setModel, this.coord);  //Enviar el modelo
40       c.sendToClient(Orders.runLambda);   //Enviar la orden
41       try {  //Esperar la respuesta
42         wait();
43       } catch (InterruptedException e) {
44       }
45     }
46   }
47   public synchronized void deltfcn() {
48     for(ClientHandler c : clientList){
49       c.sendToClient(Orders.setModel, this.coord);
50       c.sendToClient(Orders.runDelta);
51       try {
52         wait();
53       } catch (InterruptedException e) {
54       }
55     }
56   }
```

El servidor hace de coordinador y de punto de conexión entre los clientes. Por un lado añade los clientes a una lista por orden de conexión y crea un manejador para cada uno. Por otro lado se encarga de mandar las ordenes de transición, mantener el modelo actualizado y propagar las salidas del modelo. Esta es la parte más crítica de la simulación distribuida, ya que en ella recae gran parte de la eficiencia. Cabe destacar que mantener el modelo actualizado en todos los componentes del sistema distribuido consume gran cantidad de tiempo de ejecución.

Método del manejador que se ocupa de recibir el modelo actualizado de los clientes:

ClientHandler.java

```
1  public synchronized void run() {
2    while(!Thread.currentThread().isInterrupted()){
3    //Leemos la orden
4    try {
5      if(in==null) in = new ObjectInputStream(cliente.getInputStream());
6        orden = in.readObject();
7        if(orden.equals(Orders.setModel)){
```

```java
        CoordinatorDistributed model = (CoordinatorDistributed) in.readObject()
            ;
        this.server.setModel(model);
        synchronized (server) {
          this.server.notify(); //Avisamos que se ha actualizado el modelo
        }
      }
  } catch (IOException | ClassNotFoundException e) {

  }
  }
  try {
    in.close();
    out.close();
    cliente.close();
  } catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }
}
```

Cada manejador asignado a un cliente se encargará de notificar al servidor cuando el modelo haya sido actualizado.

Coordinador distribuido en el que se sobreescriben las funciones *lambda* y *deltfcn*:

CoordinatorDistributed.java

```java
public class CoordinatorDistributed extends Coordinator{
  private static final long serialVersionUID = 4283308305613014 26L;
  public CoordinatorDistributed() {
    super(null);
  }
  public CoordinatorDistributed(Coupled model) {
    super(model, true);
  }
  public CoordinatorDistributed(Coupled model, boolean flatten) {
    super(new SimulationClock(), model, flatten);
  }
  public CoordinatorDistributed(SimulationClock clock, Coupled model, boolean
      flatten) {
    super(clock, model, flatten);
  }
  public void lambda(ArrayList<Integer> lamb) {
    for(Integer x: lamb){
      this.simulators.get(x).lambda();
    }
  }
  public void deltfcn(ArrayList<Integer> delt){
    for(Integer x: delt){
      this.simulators.get(x).deltfcn();
    }
  }
}
```

Como vemos en el CoordinatorDistributed, es necesario serializar las clases que participan en el intercambio de mensajes ya que una vez establecida la conexión entre cliente y servidor, el receptor tiene que ser capaz de reconocer la clase que está recibiendo mediante su "*serialVersionUID*".

Las clases que necesitan ser serializadas por el usuario son tanto la clase *Atomic* como la clase *Coupled*, que por supuesto deben tener la misma id tanto en el cliente como el servidor. Para comprender la serialización vamos a poner un modelo de ejemplo el cual va a ser *MyPulseGeneratorExample* que es un generador de pulsos en el que se especifica la amplitud, el ancho de pulso, el periodo y la fase. Este se compone de dos modelos, uno se dedica a generar la señal y otro se dedica a mostrar el tiempo actual y el valor del pulso por pantalla.

Modelo acoplado del generador de pulsos

```
public class MyPulseGeneratorExample extends Coupled{

  private static final long serialVersionUID = -2358243267068712108L;

  public MyPulseGeneratorExample() {
    super("PulseGeneratorExample");
    MyPulseGenerator pulse = new MyPulseGenerator("Pulse", 10, 3, 5, 5);
    super.addComponent(pulse);
    MyCsvConsole scope = new MyCsvConsole("CSV");
    super.addComponent(scope);
    super.addCoupling(pulse, pulse.oOut, scope, scope.iIn);
    }
}
```

Generador de pulsos

```
public class MyPulseGenerator extends Atomic{
  private static final long serialVersionUID = -252236766591516943L;
  public OutPort<Double> oOut = new OutPort<>("out");
  protected double amplitude;
  protected double pulseWidth;
  protected double period;
  protected double phaseDelay;
  public MyPulseGenerator(String name, double amplitude, double pulseWidth,
      double period, double phaseDelay) {
    super(name);
    super.addOutPort(oOut);
    this.amplitude = amplitude;
    this.pulseWidth = pulseWidth;
    this.period = period;
    this.phaseDelay = phaseDelay;
  }
  public void initialize() {
     super.holdIn("delay", 0);
```

```
18      }
19       public void exit() {
20      }
21      public void deltint() {
22          if (super.phaseIs("delay")) {
23            super.holdIn("high", phaseDelay);
24        } else if (super.phaseIs("high")) {
25            super.holdIn("low", pulseWidth);
26        } else if (super.phaseIs("low")) {
27            super.holdIn("high", period - pulseWidth);
28        }
29      }
30      public void deltext(double e) {
31      }
32      public void lambda() {
33        System.out.println("Phase: "+ super.phase);
34        if (super.phaseIs("delay")) {
35          oOut.addValue(0.0);
36        } else if (super.phaseIs("high")) {
37          oOut.addValue(amplitude);
38        } else if (super.phaseIs("low")) {
39          oOut.addValue(0.0);
40        }
41      }
42    }
```

Consola externa

```
1   public class MyCsvConsole extends Atomic {
2     private static final long serialVersionUID = -8768429093916352817L;
3     public InPort<Number> iIn = new InPort<>("in");
4     protected double time;
5     public MyCsvConsole(String csvPath) {
6       super("CsvConsole");
7       super.addInPort(iIn);
8     }
9     public void initialize() {
10      this.time = 0.0;
11      super.passivate();
12    }
13    public void exit() {
14    }
15    public void deltint() {
16      time += super.getSigma();
17      super.passivate();
18    }
19    public void deltext(double e) {
20      time += e;
21      if (!iIn.isEmpty()) {
22        System.out.println(time + ";" + iIn.getSingleValue().doubleValue());
23      }else{
24        System.out.println("in esta vacio");
```

```
25        }
26    }
27    public void lambda() {
28    }
29 }
```

4.3. Pruebas de rendimiento

Para evaluar el rendimiento de la arquitectura de simulación DEVS desarrollada en este trabajo, hemos optado por el benchmark DEVStone. En primer lugar porque se está convirtiendo en el estándar para evaluar el rendimiento de simuladores DEVS. En segundo lugar porque queríamos evaluar la bondad de DEVStone para ejecutar simulaciones distribuidas, donde este benchmark no se ha probado aún. DEVStone es un benchmark diseñado para comprobar tiempos de ejecución de modelos DEVS el cual realiza cierto número de iteraciones en cada una de las cuales aumenta la complejidad del sistema, añadiendo componentes y simulandolos junto con los anteriores. Y lo vamos a distribuir usando un servidor como coordinador y dos clientes como trabajadores. A continuación vamos a profundizar en el benchmark DEVStone que hemos utilizado para realizar las pruebas de rendimiento.

Además hemos usado diferentes plataformas de prueba para abarcar la mayor cantidad posible de escenarios. No sólo compararemos el rendimiento de nuestro sistema en todos estos escenarios sino que también compararemos este con el rendimiento del sistema paralelo y secuencial, estas plataformas serán una red LAN y una red virtual.

4.3.1. DEVStone

DEVStone es un benchmark sintético (diseñado para medir el rendimiento de un componente especifico de un sistema) que ha sido usado en los años recientes para evaluar la actuación de diferentes simuladores DEVS. DEVStone puede ser usado para generar autómaticamente una gran variedad de modelos con diferentes tamaños y formas. Estos modelos pueden entonces ser simulados para comprobar diferentes características con respecto a un determinado simulador.

Un benchmark DEVStone se define con 5 parámetros:

- **Tipo**: Diferentes esquemas de estructura y de interconexión entre los componentes en el modelo.
- **Anchura**: Este parámetro se basa en el número de componentes en cada modelo acoplado.
- **Profundidad**: El número de niveles en la jerarquía del modelo.
- **Tiempo de transición interna**: El tiempo de ejecución dedicado por cada función de transición interna.

- **Tiempo de transición externa**: El tiempo de ejecución dedicado por cada función de transición externa.

DEVStone define cinco tipos de modelos: LI (Low level of interconnections), HI (High Input couplings), HO (HI model with numerous Outputs), HOmod y HOmem (variantes más complejas de HO). A continuación serán presentados y se utilizarán diferentes ecuaciones para computar el número de funciones de transición externas e internas.

LI (Low level of Interconnections)

La figura 4.2 muestra la estructura general del modelo LI. Con d capas de profundidad, la primera $d - 1$ capa (con $d \geq 1$) tiene la estructura de la figura 4.2(a). Todas estas capas tienen un modelo acoplado y $w - 1$ (con $w \geq 2$) modelos atómicos (donde w es el ancho). Por otro lado podemos ver la estructura de la figura 4.2(b) que tiene la capa más simple en cuánto a profundidad, la cual posee únicamente un modelo atómico. Las flechas en las figuras representan la conexión entre los puertos de entrada y salida en todo el modelo.

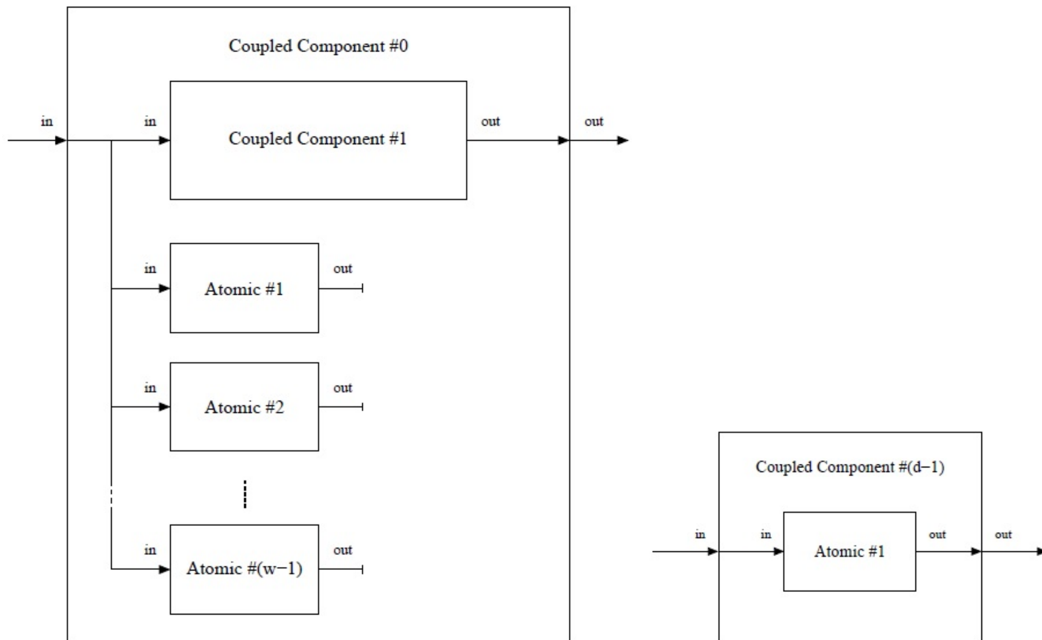

Figura 4.2: DEVStone modelo LI figuras a y b respectivamente.

Como se ha indicado anteriormente, se miden dos parámetros, tiempo de ejecución y consumo de memoria. Obviamente estos dos parámetros dependen del número de modelos atómicos, número de transiciones internas, número de transiciones externas y el número total de eventos generados internamente. Además, el consumo de memoria depende de la concurrencia del modelo, es decir, del número de eventos pendientes que están en espera al mismo tiempo en los puertos de entrada.

Para simplificar el cálculo del total de número de eventos disparados asumiremos que tanto el tiempo de transición externa como el de transición interna son 0 segundos, y que todos los eventos inyectados en el benchmark son separados por un tiempo mayor a 0 segundos. Dado que la estructura del modelo es conocida, el tiempo de ejecución teórico y el número total de eventos generados se pueden calcular fácilmente.

En primer lugar, hay que tener en cuenta el modelo de $d-1$ capas con $w-1$ modelos atómicos, y que conocemos que una capa contiene un modelo atómico. Con esto podemos saber que el número total de modelos atómicos es $(w-1) \times (d-1) + 1$.

En segundo lugar, los modelos LI producen una transición externa, un evento de salida, y una transición interna por cada modelo atómico y mensaje externo. Por lo tanto en los modelos LI, el número de transiciones y eventos generados es igual al número de modelos atómicos multiplicado por el número total de eventos externos N:

$\#\delta_{int} = \#\delta_{ext} = \#Eventos = N \times ((w-1) \times (d-1) + 1)$

En los siguientes benchmarks de DEVStone, derivamos las ecuaciones para el número de funciones de transición y eventos generados internamente dado un solo evento externo, es decir, para este punto de referencia:

$\#\delta_{int} = \#\delta_{ext} = \#Eventos = (w-1) \times (d-1) + 1$

HI (High Input couplings)

La figura 4.3 muestra la estructura general de un modelo HI. Es igual al modelo LI, pero el puerto de salida de un componente i atómico está conectado al puerto de entrada del siguiente componente atómico $i+1$, tal y como se ve en la figura 4.3(a).

Por lo tanto, el número de modelos atómicos es igual al del modelo LI. Sin embargo, el número de funciones de transición y eventos generados son bastante diferentes, ya que para cada entrada externa, el conjunto de $w-1$ modelos atómicos actúa como un registro de desplazamiento, lo que genera un evento adicional para cada evento externo. Como resultado, el número de modelos atómicos, funciones de transición, y eventos generados se calcula de la siguiente manera:

$$
\begin{aligned}
\#\text{Atomic} &= (w-1) \cdot (d-1) + 1 \\
\#\delta_{\text{int}} &= \left((w-1) + \sum_{i=1}^{w-2} i\right) \cdot (d-1) + 1 = \left(\frac{w^2-w}{2}\right) \cdot (d-1) + 1 \\
\#\delta_{\text{ext}} &= \left((w-1) + \sum_{i=1}^{w-2} i\right) \cdot (d-1) + 1 = \left(\frac{w^2-w}{2}\right) \cdot (d-1) + 1 \\
\#\text{Events} &= \left((w-1) + \sum_{i=1}^{w-2} i\right) \cdot (d-1) + 1 = \left(\frac{w^2-w}{2}\right) \cdot (d-1) + 1
\end{aligned}
$$

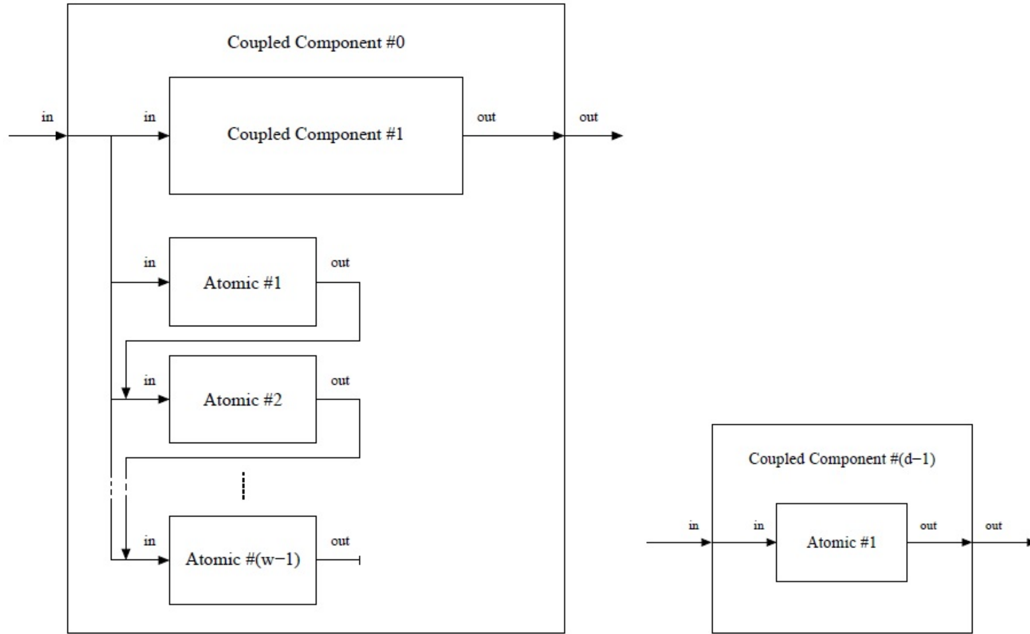

Figura 4.3: DEVStone modelo HI figuras a y b respectivamente.

HO (HI model with numerous Outputs)

La figura 4.4 muestra la estructura general de un modelo HO. Este tiene un mapa de interconexión más complejo con el mismo número de componentes atómicos y acoplados. Por ejemplo, los componentes acoplados de HO tienen 2 entradas y 2 puertos de salida en cada capa. La principal diferencia con HI es que el segundo puerto de entrada de cada modelo acoplado está conectado a la entrada de cada modelo atómico. Además, la salida de cada modelo atómico está conectada a la segunda salida de su modelo acoplado padre.

Vale la pena mencionar que el número de modelos atómicos, funciones de transición, y eventos generados en los modelos HO son exactamente los mismos que en el modelo HI exceptuando el tiempo de ejecución y el consumo de memoria, que son más altos debido a las conexiones de entrada externa adicionales.

$$
\begin{aligned}
\#\text{Atomic} &= (w-1)\cdot(d-1)+1 \\
\#\delta_{\text{int}} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \\
\#\delta_{\text{ext}} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \\
\#\text{Events} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1
\end{aligned}
$$

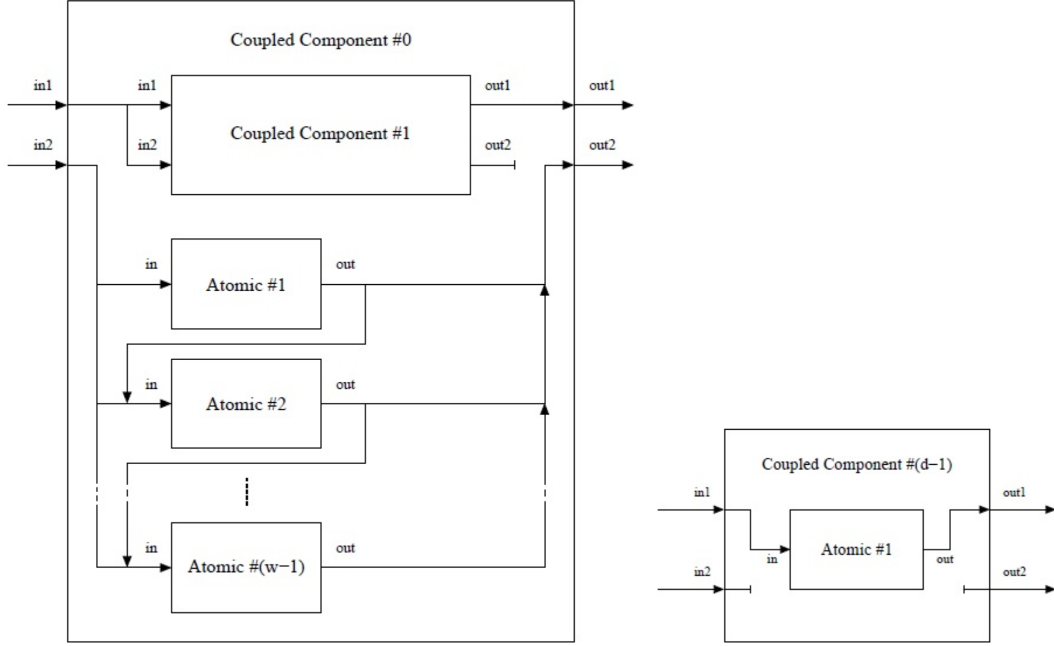

Figura 4.4: DEVStone modelo HO figuras a y b respectivamente.

HOmod

La figura 4.5 representa la estructura de un modelo HOmod. Como es usual, el modelo acoplado más profundo está formado por un solo modelo atómico. Los modelos acoplados restantes están constituidos por un modelo acoplado, una cadena de $w-1$ modelos atómicos, y un conjunto de $k = 1 \ldots w-1$ cadenas formadas por $\sum_{i=1}^{k} i$ modelos atómicos. El segundo puerto de entrada externo está conectado a toda la primera fila y sólo al primer componente atómico en las filas restantes. Además, todos los modelos atómicos en la segunda fila están conectados a la primera fila, que a su vez envía toda la salida directamente al componente acoplado. Por último cada componente atómico restante está conectado a su componente superior.

El cálculo de número de modelos atómicos y transiciones se realiza de la siguiente manera:

$$\begin{aligned}
\#\text{Atomic} &= \left((w-1) + \sum_{i=1}^{w-1} i\right) \cdot (d-1) + 1 \\
\#\delta\text{int} &= (d-1)\cdot(w-1)^2 + \left((d-1) + (w-1)\cdot\sum_{i=1}^{d-2} i\right) \times \left((w-1) + \sum_{i=1}^{w-1} i\right) + 1 \\
\#\delta\text{ext} &= (d-1)\cdot(w-1)^2 + \left((d-1) + (w-1)\cdot\sum_{i=1}^{d-2} i\right) \times \left((w-1) + \sum_{i=1}^{w-1} i\right) + 1
\end{aligned}$$

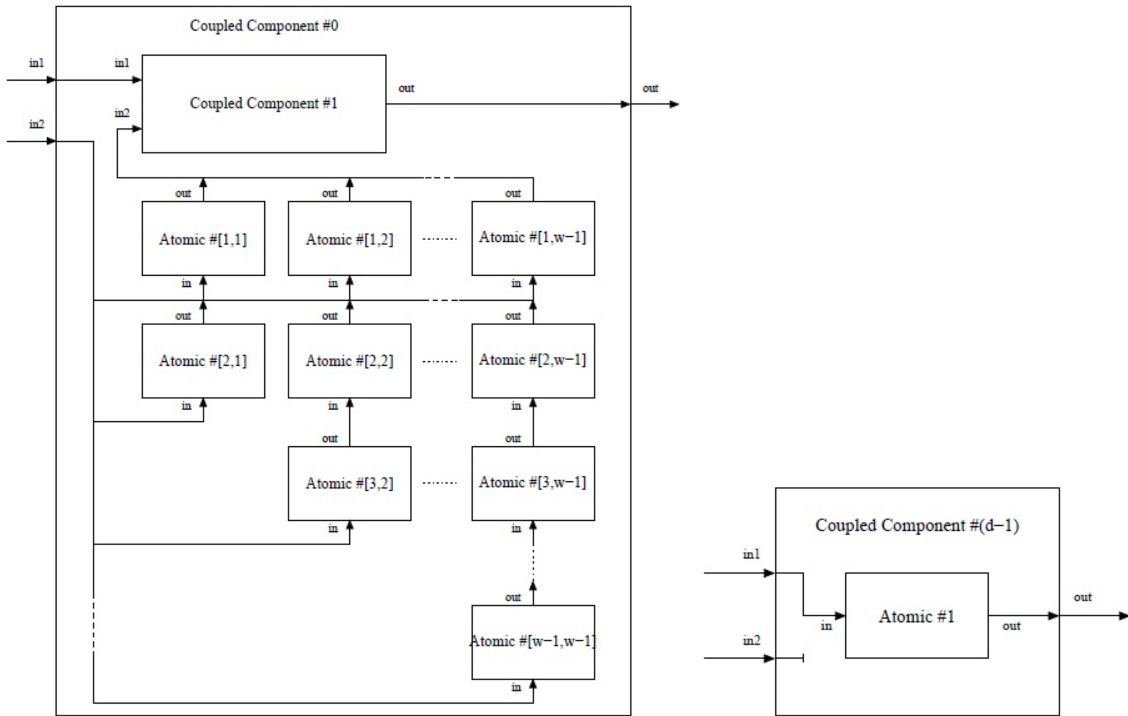

Figura 4.5: DEVStone modelo HOmod figuras a y b respectivamente.

De manera similar, el cálculo de la cantidad de eventos sigue una ecuación recursiva que se define a continuación:

$$\#\text{Events} = \sum_{l=1}^{d-1} \left(\sum_{c=1}^{K_l+w-1} \left(W_1 \times \sum_{i=1}^{w} P_l^{c-i+1} + \sum_{i=1}^{w} \left(W_i \cdot P_l^{c-i+1} \right) \right) \right) + 1$$

Como puede verse, la complejidad de las ecuaciones que describen las métricas de HOmod es alta. La inclusión de estas ecuaciones en un simulador es dificil, y el análisis teórico se convierte en prohibitivo. Por estas razones, se ha definido un nuevo benchmark DEVStone llamado HOmem que, proporcionando el mismo esfuerzo de cálculo que el HOmod, en los diferentes marcos de simulación, muestra una formulación matemática sencilla.

HOmem

HOmem propuesto como un mecanismo para incrementar el tráfico de mensajes con respecto al HO, es equivalente al HOmod, pero con una estructura y descripción matemática más sencilla.

La figura 4.6 muestra la estructura del benchmark HOmem de DEVStone. Como puede verse, el modelo acoplado más profundo es idéntico al de HOmod. En cuanto a los restantes modelos

acoplados, cada uno de ellos está formado por un modelo acoplado y $2 \times (w-1)$ modelos atómicos. La segunda cadena $w-1$ recibe la entrada a través de conexiones de entrada externas, y se propagan estas entradas a la primera cadena $w-1$ de los modelos atómicos. Estos, a su vez, envían todas las entradas recibidas al modelo acoplado.

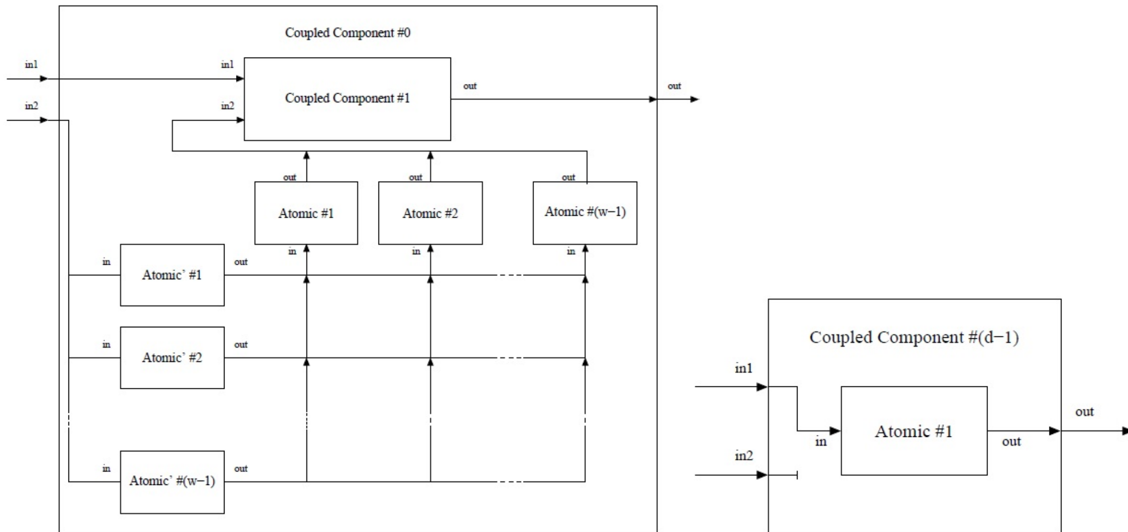

Figura 4.6: DEVStone modelo HOmem figuras a y b respectivamente.

El número de funciones de transición son fáciles de calcular, ya que es igual al número de modelos atómicos. Sin embargo, para calcular el número de eventos se debe tener en cuenta que cada evento único es enviado $w-1$ veces a toda la segunda cadena de modelos atómicos. Esto crece exponencialmente con la profundidad del modelo de la siguiente forma:

$$\begin{aligned}
\#\text{Atomic} &= 2 \cdot (w-1) \cdot (d-1) + 1 \\
\#\delta_{\text{int}} &= 2 \cdot (w-1) \cdot (d-1) + 1 \\
\#\delta_{\text{ext}} &= 2 \cdot (w-1) \cdot (d-1) + 1 \\
\#\text{Events} &= \sum_{l=1}^{d-1} \left((w-1)^{2 \cdot l} + (w-1)^{2 \cdot l - 1} \right) + 1
\end{aligned}$$

Los resultados experimentales muestran que esta especificación simple conduce a un tiempo de ejecución y consumo de memoria similar cuando se compara con HOmod.

4.3.2. Pruebas en red virtual

En esta prueba hemos creado una red virtual con el programa tunngle, que es un cliente $p2p$ VPN que ofrece redes $ipv4$ virtuales, para probar el sistema en redes lentas. En este caso la red tiene una velocidad aproximada de 5Mbps y una latencia de 60ms. Hemos realizado esta prueba para demostrar que un modelo DEVS distribuido no funciona a pleno rendimiento en este tipo de redes. Para todas las pruebas vamos a usar el modelo HOmod ya que es el que más interconexión de atómicos tiene y el que va a dar más sufrimiento a nuestro sistema.

La primera columna muestra el número máximo de eventos simultáneos, la segunda y la tercera el ancho y la profundidad del sistema respectivamente. La cuarta el número de transiciones internas, la quinta el número de transiciones externas y por último el número de eventos y el tiempo de los sistemas secuencial paralelo y distribuido respectivamente. Para comprender mejor los valores de la tabla hemos usado colores fríos para tiempos más pequeños y colores cálidos para tiempos mayores, también cabe destacar que el tiempo esta dado en segundos. Cada fila se corresponde con una simulación de una iteración concreta, siendo el número de componentes mayor que la anterior. Como vemos en la tabla 4.1 el sistema paralelo tarda un poco más que el secuencial, esto se debe a la virtualización que realiza la máquina virtual de Java cuando se crean varios procesos simultáneos.

Como se aprecia en la tabla 4.1 los tiempos de la simulación distribuida son mucho mayores con respecto a los otros dos sistemas, esto es debido a que en cada transición se produce un envío para actualizar el modelo del servidor. También se puede observar que en las filas donde el número máximo de eventos simultaneos es dos, se duplica el tiempo de simulación distribuida. Esto se debe a que se duplica el número de envíos necesarios y en consecuencia el tiempo.

4.3.3. Pruebas en LAN con Gigabit Ethernet

Para esta prueba hemos utilizado la red local de nuestro hogar en la que no sucederán congestiones y en la que la infraestructura permite alcanzar velocidades de hasta un gigabit por segundo. Para ello también vamos a usar el modelo HOmod para mantener la consistencia con la prueba anterior. Como vemos en la tabla 4.2 los tiempos siguen siendo muy altos en comparación con los otros sistemas de simulación, esto se debe a que se producen demasiados envíos ya que en cada transición se actualiza el modelo del servidor y este tiene que esperar. Por otro lado podemos percibir que el tiempo de simulación distribuida se ha reducido considerablemente con respecto a la prueba anterior, debido a que la velocidad de transmisión es mucho mayor y a que se no se producen congestiones en la red. Hay que considerar que aunque utilizáramos otro modelo de DEVStone, los resultados serían similares ya que despliegan muchos componentes y el computo por cada uno es mínimo.

max events	width	depth	δ_{int}	δ_{ext}	events	secuencial	paralelo	distribuido
1	2	2	4	4	4	**0.002**	**0.003**	**2.978**
2	2	2	8	8	8	**0.0**	**0.002**	**2.61**
1	3	2	10	10	10	**0.0**	**0.001**	**0.49**
2	3	2	20	20	20	**0.0**	**0.002**	**0.903**
1	4	2	19	19	19	**0.0**	**0.002**	**0.517**
2	4	2	38	38	38	**0.001**	**0.003**	**2.823**
1	5	2	31	31	31	**0.0**	**0.002**	**2.578**
2	5	2	62	62	62	**0.001**	**0.006**	**5.744**
1	2	3	9	9	10	**0.0**	**0.001**	**1.612**
2	2	3	18	18	20	**0.0**	**0.001**	**4.519**
1	3	3	29	29	64	**0.001**	**0.002**	**0.64**
2	3	3	58	58	128	**0.001**	**0.003**	**5.283**
1	4	3	64	64	235	**0.0**	**0.002**	**0.684**
2	4	3	128	128	470	**0.0**	**0.004**	**4.209**
1	5	3	117	117	631	**0.001**	**0.003**	**1.166**
2	5	3	234	234	1262	**0.002**	**0.006**	**5.42**
1	2	4	16	16	22	**0.0**	**0.002**	**1.161**
2	2	4	32	32	44	**0.0**	**0.002**	**6.389**
1	3	4	58	58	388	**0.0**	**0.002**	**3.617**
2	3	4	116	116	776	**0.0**	**0.004**	**5.454**
1	4	4	136	136	2827	**0.001**	**0.005**	**1.134**
2	4	4	272	272	5654	**0.01**	**0.008**	**10.752**
1	5	4	259	259	12631	**0.002**	**0.01**	**3.8**
2	5	4	518	518	25262	**0.005**	**0.019**	**12.016**
1	2	5	25	25	46	**0.001**	**0.002**	**1.344**
2	2	5	50	50	92	**0.001**	**0.002**	**8.06**
1	3	5	97	97	2332	**0.0**	**0.003**	**0.863**
2	3	5	194	194	4664	**0.001**	**0.004**	**7.987**
1	4	5	235	235	33931	**0.003**	**0.007**	**11.373**
2	4	5	470	470	67862	**0.007**	**0.014**	**9.471**
1	5	5	457	457	252631	**0.025**	**0.029**	**19.861**
2	5	5	914	914	505262	**0.044**	**0.057**	**33.265**

Cuadro 4.1: DevStone: HOmod comparación en red virtual

4.3.4. Comparativa según el tipo de red

En esta sección vamos a comprobar el impacto del tipo de red en el que trabaja nuestro nuestro sistema distribuido, con el objetivo de conocer el tipo de red necesaria para exprimir el máximo rendimiento del sistema. Para ello vamos a calcular la media de tiempos del benchmark DevStone en las dos plataformas probadas y los vamos a comparar en un gráfico circular.

max events	width	depth	δ_{int}	δ_{ext}	events	secuencial	paralelo	distribuido
1	2	2	4	4	4	**0.002**	**0.003**	**0.175**
2	2	2	8	8	8	**0.0**	**0.002**	**0.31**
1	3	2	10	10	10	**0.0**	**0.001**	**0.306**
2	3	2	20	20	20	**0.0**	**0.002**	**0.693**
1	4	2	19	19	19	**0.0**	**0.002**	**0.636**
2	4	2	38	38	38	**0.001**	**0.003**	**0.79**
1	5	2	31	31	31	**0.0**	**0.002**	**1.767**
2	5	2	62	62	62	**0.001**	**0.006**	**1.607**
1	2	3	9	9	10	**0.0**	**0.001**	**0.919**
2	2	3	18	18	20	**0.0**	**0.001**	**0.629**
1	3	3	29	29	64	**0.001**	**0.002**	**0.419**
2	3	3	58	58	128	**0.001**	**0.003**	**1.87**
1	4	3	64	64	235	**0.0**	**0.002**	**0.748**
2	4	3	128	128	470	**0.0**	**0.004**	**3.822**
1	5	3	117	117	631	**0.001**	**0.003**	**0.749**
2	5	3	234	234	1262	**0.002**	**0.006**	**3.344**
1	2	4	16	16	22	**0.0**	**0.002**	**0.578**
2	2	4	32	32	44	**0.0**	**0.002**	**2.141**
1	3	4	58	58	388	**0.0**	**0.002**	**1.267**
2	3	4	116	116	776	**0.0**	**0.004**	**3.008**
1	4	4	136	136	2827	**0.001**	**0.005**	**2.508**
2	4	4	272	272	5654	**0.01**	**0.008**	**2.334**
1	5	4	259	259	12631	**0.002**	**0.01**	**1.963**
2	5	4	518	518	25262	**0.005**	**0.019**	**3.216**
1	2	5	25	25	46	**0.001**	**0.002**	**1.146**
2	2	5	50	50	92	**0.001**	**0.002**	**1.896**
1	3	5	97	97	2332	**0.0**	**0.003**	**2.057**
2	3	5	194	194	4664	**0.001**	**0.004**	**6.493**
1	4	5	235	235	33931	**0.003**	**0.007**	**2.381**
2	4	5	470	470	67862	**0.007**	**0.014**	**3.347**
1	5	5	457	457	252631	**0.025**	**0.029**	**2.547**
2	5	5	914	914	505262	**0.044**	**0.057**	**19.126**

Cuadro 4.2: DevStone: HOmod comparación en red LAN

Como vemos en el gráfico 4.7, la media de tiempo de simulación en la red virtual es mucho mayor que en la red LAN. Cabe destacar que siendo la red LAN doscientas veces más rápida, solo logra duplicar la velocidad de nuestro sistema (5,2 segundos frente a 2,3) por lo que este no aprovecha en su totalidad el ancho de banda de la red. Esto puede ser debido a que Java penaliza cada bloqueo del servidor con un tiempo determinado. Por lo que concluímos que con una red

con un ancho de banda de al menos 20 Mbps sería suficiente para aprovechar la velocidad de los Sockets de Java.

Figura 4.7: Comparación de la simulación distribuida en una red virtual y en una red LAN.

4.4. Conclusiones

Como se puede observar en los resultados de cada uno de los ámbitos, el sistema distribuido no rivaliza con los otros dos. Esto es debido al envío de información constante cada vez que el modelo se modifica, por lo que se producen bloqueos. Estos son debidos a la espera que se produce cuando se envían o reciben datos que necesitan sincronización para su correcto funcionamiento por lo que el tiempo de ejecución es mucho mayor.

También observamos que en redes lentas en las que se pueden ocasionar congestiones, la distribución pierde parte de eficiencia ya que el tiempo de transmisión de datos es elevado respecto a otras redes más rápidas y eficientes. Por otro lado el benchmark DevStone no es totalmente apropiado para el sistema distribuido ya que las transiciones de estado son instantáneas y el sistema no puede aprovechar los envíos del modelo ya que el computo por cada envío es mínimo, por lo que se propone como trabajo futuro adaptar DEVStone para simular modelos distribuidos, con menos cantidad de componentes y más computo por cada uno.

Como vemos en el gráfico 4.7 la infraestructura en la que se ejecuta el sistema distribuido es importante, pero también se aprecia que los Jsockets junto con la máquina virtual de Java, no aprovechan todo el ancho de banda de una red LAN ya que tienen una limitación debido a su protocolo. Esto lo sabemos porque la red Gigabit Ethernet es mucho más rápida y sin embargo solamente logra duplicar el rendimiento respecto a la red virtual, siendo la primera doscientas veces más rápida. Los códigos que se muestran en este trabajo son versiones simplificadas de los originales. Todos los códigos fuente de este proyecto se pueden encontrar en https://github.com/jlrisco/xdevs-lib/tree/master/java/src/xdevs/lib/tfgs/c1516/xdevs_distributed.

Bibliografía

[1] B. P. ZEIGLER, T.G. KIM AND H. PRAEHOFER, *“Theory of Modeling and Simulation"*, 2ª ed. New York: Academic Press, 2000.

[2] H. L. M. VANGHELUWE, *“DEVS as a common denominator for multiformalism hybrid systems modelling"*, Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design, pp. 129-134, 2000.

[3] JOSÉ L. RISCO Y SAURABH MITAL, *“xDEVS r20150903"* : https://docs.google.com/document/d/1_AMjvXbaUxICmLjbnIUzdrgtITLUV8p1VDKvb8OYS-Y/

[4] G. COULOURIS, J. DOLLIMORE, T.KINDBERG., *Sistemas distribuidos. Coneptos y diseño.*, Addison-Wesley, 2005, 4º edición.

[5] M. L. LIU. PEARSON, *Computación distribuida. Fundamentos y Aplicaciones.*, Educación, 2004.

[6] KENNETH L. CALVERT, MICHAEL J. DONAHOO, *TCP/IP Sockets in Java, Practical guide for programmers*, Morgan Kaufmann, second edition.

[7] WIKIPEDIA, *“DEVS"* : https://en.wikipedia.org/wiki/DEVS

[8] WIKIPEDIA, *“Simulation"* : https://en.wikipedia.org/wiki/Simulation

[9] ERNESTO KOFMAN, *Formalismo DEVS - Teoría y Aplicaciones* : http://www.fceia.unr.edu.ar/dsf/devs/apunte_devs.PDF

Agradecimientos

A José Luis Risco Martín tanto por la confianza y paciencia depositada en nosotros como por sus indicaciones cada vez que teníamos problemas.

A nuestros familiares, compañeros y amigos por el apoyo incondicional y por aguantarnos en los momentos críticos.

A nuestro compañero que realizaba otro TFG sobre xDEVS por la simpatía y la ayuda recibida.

Autorización de difusión

Autorización para la difusión del Trabajo Fin de Grado y su depósito en el Repositorio Institucional E-Prints Complutense

Los abajo firmantes, alumno y tutor del Trabajo Fin de Grado (TFG) en Ingeniería de Computadores de la facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el Trabajo Fin de Grado (TFG) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

TÍTULO del TFG: **Diseño e implementación del kernel de xDEVS, versión distribuida**.

Curso académico: 2015/2016

Alumnos:

Guillermo Llorente de la Cita

Luis Lázaro-Carrasco Hernández

Tutor del TFG: **José Luis Risco Martín**, Departamento de Arquitectura de Computadores.

Firma de los alumnos — Firma del tutor