Máster en Redes y Servicios de Comunicación Móviles

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

PROYECTO FIN DE MÁSTER

Implementación de un Simulador de Redes de Sensores Inalámbricas Distribuido Basado en el Algoritmo de *Time Warp*

Autor: Arturo Díaz Almagro

Tutor: Antonio J. Caamaño Fernández

Co-Tutor: Mark R. Wilby

Curso Académico 2010/2011

Universidad Rey Juan Carlos

ACTA DE EVALUACIÓN

Alumno: Arturo Díaz Almagro
Titulación: Máster Oficial en Redes y Servicios de Comunicación Móviles (Res-Móvil).
Título del Proyecto: Implementación de un Simulador de Redes de Sensores Inalámbricas Distribuido Basado en el Algoritmo de *Time Warp.*

¿Es el proyecto resultado de Prácticas en empresas? SÍ / NO

Tutor: Antonio José Caamaño Fernández
Co-tutor: Mark Richard Wilby

TRIBUNAL

Presidente: Francisco Javier Atero Gómez
Vocal: Francisco Javier Ramos López
Secretario: Eduardo Morgado Reyes

CALIFICACIÓN DETALLADA DEL PROYECTO

	Presidente	**Vocal**	**Secretario**
Presentación escrita: (MB-B-R-M-MM) **Presentación oral:** (MB-B-R-M-MM)			
Complejidad técnica: (MB-B-R-M-MM)			
Metodología empleada: (MB-B-R-M-MM)			
Resultados obtenidos: (MB-B-R-M-MM)			
Esfuerzo realizado: (MB-B-R-M-MM)			

CALIFICACIÓN FINAL DEL PROYECTO

	SB / NOT / AP / SS / NP
(nota numérica)	Enmarcar la calificación alcanzada

PROYECTO PROPUESTO PARA MATRÍCULA DE HONOR: SÍ / NO
(sólo si la nota numérica final es igual a 9.5)

Fuenlabrada, 4 de Julio de 2011

El Presidente El Vocal El Secretario

A mi padre

AGRADECIMIENTOS

Desde que comencé con el desarrollo de este Proyecto Fin de Máster han sucedido muchas cosas en mi vida. Tras casi dos años de trabajo interrumpido es normal que esto ocurra. He perdido a mi padre y he ganado a mi hija. Una cosa no compensa la otra pero sí ayuda a afrontar la situación con un ánimo positivo.

Este proyecto está dedicado a mi padre, a su memoria. En los últimos tiempos siempre me regañaba por no rematarlo y yo siempre le decía: ”ya no me queda casi nada para terminarlo”. Ahora ya puedo decirle: ”Papá, ya está terminado”.

La sonrisa la pone el nacimiento de mi hija, Sara. Me ha ayudado, nos ha ayudado, a reponernos ante la adversidad y también ha colaborado, inocentemente, claro, a que este proyecto se retrasara en el tiempo. También va dedicado a ella.

Cómo no, siempre has estado ahí. Dándome ánimos, pidiéndome que no me rindiera, que no lo dejara. Apoyándome en todo lo que se refiere a mi vida profesional, sin cuestionar. Y, por supuesto, también en lo personal. Te quiero Puri, no sabes cuánto. Este proyecto también va dedicado a ti.

A mi familia, por estar todos juntos, por el apoyo que nos brindamos. Que nunca cese.

Mi enorme agradecimiento a David Gutiérrez su amistad, los buenos ratos pasados en el despacho y su soporte con ZeroC ICE. Eres un crack.

Por último, quisiera agradecer a Antonio Caamaño su confianza puesta en mí para llevar a término este trabajo y a Mark Wilby sus charlas sobre computación distribuida y sus limitaciones.

RESUMEN

En este trabajo se plantea el problema de la simulación de Redes de Sensores Inalámbricas (RSN) masivas, es decir, con un elevado número de nodos y una alta densidad espacial. Los simuladores tradicionales incurren en un alto coste computacional y de recursos de memoria para la resolución de problemas en este tipo de redes ya que se trata de ejecuciones secuenciales de las transmisiones de los mensajes. Este proyecto presenta un modelo de simulación distribuido, en varios procesadores, de manera que se intente explotar el paralelismo que presentan las RSN masivas. Este paralelismo viene dado por la formación de regiones de comunicaciones pseudo-independientes con otras zonas de la red.

El modelo escogido para la resolución está basada en simulación por eventos distribuída. Dentro de este tipo existen varios modelos: *modelos conservadores* en los que los cada uno de los procesos no puede avanzar en su ejecución más allá de un tiempo determinado; y los *modelos optimistas* que permiten la ejecución libre de los procesos a riesgo de provocar errores de causalidad en la ejecución de los eventos. El simulador planteado es una solución híbrida en el que se descompone en problema en subregiones que son simuladas de manera conservadora y en el que las interacciones entre las distintas subregiones se hace de manera optimista basándose en el algoritmo de Jefferson de *Time Warp*.

ABSTRACT

This work exposes the problem of simulation of dense Wireless Sensor Networks (WSN), that means, networks with high number of nodes and high spatial density. Conventional simulators incur in too high computational costs and resources consumption to solve this type of problems due to they are afforded in a sequential way. This document presents a distributed simulation model trying to explode the intrinsic parallelism in dense WSN. This parallelism come from the region-forming nature of the problem where each region have pseudo-independent communications.

The model used here to solve the problem is the event driven distributed simulation. Within this type there are several models: *conservative* models in which each process can not be executed uncontrolled upon a given time boundary; and *optimistic* models in which this uncontrolled execution is allowed incurring in potential causality errors. The simulator posed here is a kind of hybrid solution where the problem is divided in several subproblems each simulated in a conservative way and, interactions among regions, simulated in an optimistic way based on the Jefferson's Time Warp algorithm.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPÍTULO 1

ESTADO DEL ARTE EN SIMULADORES DE REDES AD HOC

1.1 Qué son las Redes Ad Hoc

Una red de comunicaciones se define como un conjunto de dispositivos que ofrecen servicios de comunicación a usuarios. En base a esta definición, el conjunto de parámetros considerados para el diseño y operación de la red, así como los niveles de calidad exigidos, pertenecía al ámbito interno de la red; mientras que la actividad de los usuarios se limitaba a un mero acceso bajo ciertas condiciones impuestas por la propia red. Sin embargo, lo usuarios comenzaron a demandar movilidad, diversidad de medios y contenidos y, por último, libertad para el establecimiento de sus propias redes, que atendieran a sus necesidades concretas en tiempo y espacio. Al involucrarse los usuarios más en la infraestructura de las redes se fuerza el desarrollo de las mismas motivando la aparición de nuevos esquemas: redes sin infraestructura fija, compuestas sólo por dispositivos terminales y de bajo coste. Por este motivo, el control de las comunicaciones se desplazó hasta los nodos de la propia red. El origen de este tipo de redes se encuentra en los trabajos de la *Defense Advanced Research Projects Agency* (DARPA) durante los años 70, con el desarrollo de la Red Radio de Paquetes (Packet Radio Network) para comunicaciones entre vehículos en movimiento.

Para facilitar el desarrollo y la interconexión de sistemas de telecomunicación, no sólo entre fabricantes, sino también entre diferentes tecnologías de red, se hizo necesario definir una referencia modélica para la arquitectura de protocolos de dichos sistemas. Durante la década de los 80 la Organización Internacional de Estandarización (ISO) definió un modelo de referencia para la interconexión de Sistemas Abiertos (OSI, Open System Interconnection) [2].

En la figura 1.1 se muestra la conocida torre de protocolos OSI, compuesta por siete niveles con atribuciones diferentes:

- Capa Física; transmisión de símbolos entre sistemas conectados al mismo medio físico.
- Capa de Enlace: se divide en otras dos subcapas funcionales. La capa de Control de Acceso al Medio (MAC, *Medium Access Control*) proporciona mecanismos de control para un acceso ordenado al canal por parte de los usuarios. La capa de Control de Enlace Lógico (LLC, *Logical Link Control*) maneja el intercambio de bloques con control de errores.
- Capa de Red: comunicación entre sistema que no están directamente conectados al mismo medio físico.
- Capa de Transporte: conexión entre usuarios finales y aislamiento de las capas superiores (habitual-

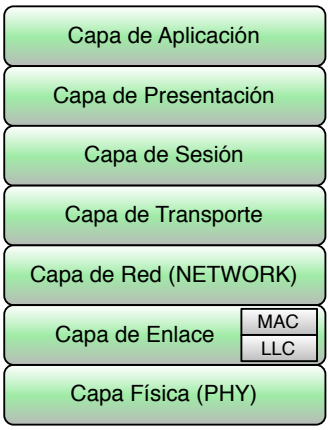


Figura 1.1 : Capas del modelo de referencia OSI

mente ligadas al tipo de servicio) frente a las distintas implementaciones de tecnologías de red.

- Capa de Sesión: organización y estructura de las conexiones entre usuarios (procesos o aplicaciones) finales.
- Capa de Presentación: representación sintáctica homogénea de la información.
- Capa de Aplicación: interpretación común de la semántica de la información.

El concepto de redes sin infraestructura cableada derivó en las conocidas Redes Ad Hoc Inalámbricas, donde el término ”Ad Hoc” establece la característica principal de su comportamiento.

Una red Ad Hoc es un conjunto de nodos que se comunican entre sí y que mantienen una estructura de red multisalto sin el soporte de una estación base o un controlador central [3]. Desde el punto de vista de las aplicaciones, las redes Ad Hoc inalámbricas son útiles en situaciones en las que se requiere de un despliegue rápido de una red local o simplemente de una red que no sea en infraestructura. Ejemplos pueden ser la respuesta a una catástrofe, aplicaciones militares o incluso conferencias.

Las redes de sensores inalámbricas son un subconjunto de las redes Ad Hoc. El cometido de éstas es la digitalización de datos distribuidos en el espacio físico, proporcionando un interfaz entre las magnitudes físicas y los dominios digitales. Los sensores forman redes Ad Hoc para compartir los datos recolectados y transmitirlos a un centro de fusión o procesado. Cada nodo actúa como encaminador de paquetes además de ser receptores de los mismos. La carga adicional que se añade a cada nodo complica el diseño y las prestaciones de los protocolos de encaminamiento [3].

Retomando la narración de la historia de las Redes Ad Hoc Inalámbricas, el estándar 802.11 del *Institute of Electrical and Electronics Engineers* (IEEE) [4] ya contenía este término al contemplar la configuración de operación independiente (Ad Hoc) de las estaciones en la capa MAC, de manera que fuera posible la comunicación entre ellas. La definición de las Redes Ad Hoc Inalámbricas se completa por la *Internet Engineering Task Force* (IETF) en [5] añadiendo la movilidad de los nodos.

El crecimiento de las Redes Ad Hoc Inalámbricas viene como consecuencia de los desarrollos propios realizados dentro del área, derivando en estándares específicos. Por ejemplo,, el propio grupo de trabajo del estándar 802.11 generó una extensión del mismo bajo el nombre de 802.11s que incluye la definición

de las denominadas redes malladas (*mesh networks*). Estas redes malladas se caracterizan por permitir la comunicación a través de topologías multisalto autoconfigurables. La primera versión del estándar (versión D0.01) apareció en marzo de 2006 y, actualmente, se encuentra en su versión D2.00 (septiembre de 2008). Por su parte, dentro del grupo de trabajo del IEEE dedicado a las Redes de Área Personal (PAN, *Personal Area Networks*), también se dedican esfuerzos al desarrollo de las redes malladas, en concreto el estándar 802.15.5. Dentro del mismo grupo, se incluye el estándar 802.15.4 dedicado a las PAN de baja tasa de transmisión. La última versión de este estándar, 802.15.4-2006, fue publicada en septiembre de 2006.

Entre las PAN de baja tasa de transmisión se encuentran las Redes de Sensores Inalámbricas (WSN, *Wireless Sensor Networks*), más extendidas hoy en día [6], [7], [8]. Este estándar recoge las capas Física y MAC, y se encuentra impulsado por la Alianza Zigbee, que cuenta entre sus promotores con potentes empresas como Philips, Siemens, Texas Instruments o Samsung entre otras. La especificación Zigbee recoge aspectos relacionados con la capa de Red y capas superiores para la creación de redes de baja tasa de transmisión y mínima complejidad, coste y consumo de potencia, destinadas primordialmente a aplicaciones de monitorización y control.

Con el surgimiento de las Redes Ad Hoc Inalámbricas apareció la necesidad de incorporar nuevos aspectos relacionados con la autonomía y el dinamismo, los cuales, unido al carácter inalámbrico, supusieron en su momento la aparición de importantes retos que superar para ofrecer los servicios requeridos. En primer lugar, el uso del canal radio frente a los medios cableados tradicionales, presenta los problemas bien conocidos de optimización del ancho de banda disponible, acceso múltiple, control de potencia, capacidad variable del canal y seguridad [9], [10], [11], [12]. En segundo lugar, el carácter dinámico de los nodos, que genera constantes modificaciones en la topología de la red, requiere nuevas técnicas para su mantenimiento y configuración. Finalmente, la ausencia de infraestructuras que soporten las comunicaciones obliga a una operación limitada energéticamente; por ello se debe acudir a nuevas propuestas relativas al direccionamiento de los nodos y al transporte de los datos a través de la red (multisalto) que faciliten el requerido ahorro energético.

Estas limitaciones repercuten en varios aspectos de la comunicación en las Redes Ad Hoc Inalámbricas. Un aspecto especialmente afectado es el encaminamiento, ya que estas redes carecen de la mayor parte de los recursos de los que se dispone para el encaminamiento en las redes tradicionales y que se obtienen de la propia infraestructura. El simple hecho de poder disponer de una red fija o celular simplifica en gran medida el problema del encaminamiento, existiendo un elevado número de técnicas que tratan de resolver el problema de la forma más efectiva posible [13]. Otro aspecto especialmente afectado, aún más si se tiene en cuenta que en él interviene el problema del encaminamiento, es el de las prestaciones ”extremo a extremo”. en las Redes Ad Hoc Inalámbricas, los caminos o rutas seguido por los datos constan de un número no predeterminado de enlaces (inalámbricos, para una mayor complejidad del problema) que pueden variar en cuanto a su número, la distancia de cada enlace, el estado del canal, etcétera. Esta es la principal razón por la que resulta interesante analizar las prestaciones alcanzadas en estas redes al recorrer la ruta completa; es decir, desde el nodo origen hasta el nodo destino, apareciendo el concepto de ”extremo a extremo”. En este tipo de análisis, la simulación de los protocolos o algoritmos de encaminamiento se convierte en fundamental para el proceso de diseño de algoritmos y optimización de prestaciones.

1.2 Introducción a la simulación de Redes Ad Hoc

Como ya se vio en la sección 1.1, el rápido crecimiento de los sistemas de comunicaciones incrementó el tiempo y esfuerzo para el análisis y diseño de los mismos [14]. Surgió, pues, la necesidad de herramientas que faciliten dicha tarea de una manera eficiente en coste y en esfuerzos. Estas herramientas no pueden ser otras que herramientas potentes de análisis y diseño asistido por computador.

En los últimos años se han desarrollado numerosas técnicas asistidas por computador para ayudar al proceso de modelado, diseño y análisis de sistemas de comunicaciones. Estas técnicas pueden dividir en dos categorías [14]:

- Aproximaciones basadas en formulación matemática: la computadora realiza evaluaciones de fórmulas matemáticas complejas.
- Aproximaciones basadas en simulación: la computadora simula determinadas condiciones y estados de los sistemas.

Un motivación importante para el uso de la simulación es que, una simulación correcta, puede arrojar mejores resultados que una implementación de laboratorio de un sistema concreto. En una simulación, las medidas pueden ser tomadas en varios puntos del sistema sin coste adicional. Pueden realizarse estudios paramétricos que arrojen resultados que puedan ayudar a la optimización como, por ejemplo, la variación de los anchos de banda de un filtro para estudiar la variación de la relación señal-a-rudio (SNR, *Signal-to-Noise Ratio*) de un sistema receptor. Más importante aún [15], si cabe, es la posibilidad de realizar estudios del tipo ”qué pasaría si” de una manera económica que si se hiciera con el sistema hardware general.

1.2.1 Capas relevantes en el diseño

Si se enfoca la simulación dentro del problema de las Redes Ad Hoc los parámetros que intervienen son numerosos. Entre ellos están el canal inalámbrico, el acceso al medio o el encaminamiento; es decir, los tres primeros niveles de la torre OSI. Como se vio en la sección 1.1, uno de los objetivos fundamentales, es el diseño de protocolos de encaminamiento eficientes que minimicen el número de saltos hasta alcanzar el centro de control, si existiera. En los últimos años, con el objetivo de esta minimización en mente, se han planteado esquemas de optimización en las que se involucran las tres primeras capas de la torre OSI [3]. Estos niveles son los más relevantes a la hora de afrontar el diseño de nuevos algoritmos de encaminamiento.

Capa Física.

En este caso, la capa física se trata del medio inalámbrico. La propagación de ondas radios a traves de un canal inalámbrico resulta un fenómeno complicado caracterizado por varios efectos, como el multitrayecto (*multipath*) y el ocultamiento (*shadowing*). Estos efectos se traducen en desvanecimientos (fading), de naturaleza aleatoria, de la señal en recepción. Así, para diseñar sistemas de comunicación (transmisores, receptores, modulaciones empleadas...) que permiten intercambiar información en escenarios con desvanecimiento, resulta necesario modelar el comportamiento estadístico del canal de comunicaciones.

Según [16] se pueden presentar dos clasificaciones para el desvanecimiento:

- Desvanecimiento lento o rápido. Un desvanecimiento se define *lento* cuando el tiempo en el que el canal permanece estable (tiempo de coherencia) es mayor que la duración de la transmisión de un símbolo (tiempo de símbolo), y *rápido* en caso contrario.
- Desvanecimiento plano o selectivo en frecuencia. Un desvanecimiento *plano* en frecuencia es el que afecta de forma similar a todas las componentes espectrales de la señal transmitida o, dicho de otro modo, el ancho de banda de la señal es menor que el ancho de banda de coherencia del canal. Estos canales se conocen como canales selectivos en tiempo. Por el contrario, un desvanecimiento *selectivo* en frecuencia es aquel que no afecta por igual a todas las componentes espectrales de la señal. Cuando se unen ambos tipos de desvanecimiento, se dice que el canal es doblemente selectivo o selectivo en tiempo y frecuencia.

También, según [16], los principales parámetros para evaluar las prestaciones de un sistema de comunicaciones a nivel de capa Física son:

- Relación Señal-a-Ruido (SNR, *Signal-to-Noise Ratio*): es la relación entre la potencia de la señal respecto a la potencia de ruido (térmico) a la entrada del receptor. Se trata de la medida de prestaciones más utilizada por ser de las más fácilmente evaluables. En sistemas inalámbricos se define el término *SNR media* como el parámetro SNR que tienen en cuenta la media estadística necesaria con la aparición del desvanecimiento en el canal. Matemáticamente, si la variable aleatoria (v.a.) γ representa la SNR instantánea, incluyendo los efectos del desvanecimiento, las SNR media se calcula como:
$$\bar{\gamma} = \int_0^\infty \gamma \rho(\gamma) d\gamma \tag{1.1}$$
siendo $\rho(\gamma)$ la Función Densidad de Probabilidad (PDF, *Probability Density Function*) de la v.a. γ.
- Probabilidad de Indisponibilidad o Corte del Enlace (*Outage*): se define como la probabilidad de que la SNR instantánea no alcance un nivel umbral de recepción (γ_{th}) que permita que la comunicación transcurra con suficiente calidad.
$$P_{out} = \int_0^{\gamma_{th}} \rho(\gamma) d\gamma \tag{1.2}$$
o Función de Distribución Acumulada (CDF, *Cumulated Distribution Function*) de γ, evaluada en $\gamma = \gamma_{th}$.
- Probabilidad de Error de Bit (BER, *Bit Error Rate*): es la probabilidad de que el valor del bit recibido no coincida con el valor con el que fue emitido. Este parámetros es el indicador de prestaciones, en capa Física, más difícil de calcular puesto que intervienen todos los elementos que caracterizan dicha capa en un sistema de comunicaciones: canal físico y esquemas de modulación/demodulación de la señal en transmisión y recepción. Este indicador se suele presentar en forma de una expresión analítica que determina su dependencia con la SNR, dependencia que no resulta lineal, para unos esquemas de modulación y detección dados.

En cuanto a los tipos de modulación empleados en esta capa, la clasificación fundamental hace referencia al conocimiento o no, por parte del receptor, del desvanecimiento instantáneo introducido por el canal. Así, se distingue entre detección coherente (donde ese conocimiento se consigue mediante señales piloto u otras técnicas de estimación) y detección no coherente (donde se desarrollan técnicas de recepción sin este conocimiento) [17].

Capa de Enlace.

Como se indicó en 1.1, esta capa se divide en dos subcapas: LLC y MAC. Sin embargo, desde el punto de vista de este trabajo sólo se va a considerar la subcapa MAC que determina los nodos que pueden utilizar el medio inalámbrico de forma simultánea y es directamente responsable de la potencia interferente recibida por los nodos de la Red Ad Hoc Inalámbrica.

En relación con la capacidad de la capa MAC para permitir o no ciertas transmisiones, se consideran dos problemas clásicos en redes inalámbricas [18]:

- Nodo Oculto: este problema aparece cuando un nodo cree que el canal de transmisión está libre, pero en realidad no lo está. Es decir, dos nodos transmisores se encuentran dentro del radio de cobertura del nodo receptor pero fuera de sus respectivos radios de cobertura, de manera que no ”se ven” entre sí y no son capaces de coordinar su transmisión. Así se produce una transmisión simultánea que provoca una importante interferencia en el nodo receptor.

- Nodo Expuesto: este problema aparece cuando un nodo cree que el canal de transmisión está ocupado pero, en realidad, lo está ocupando otro nodo que no interferiría en su transmisión. Es decir, un nodo que desea transmitir se encuentra dentro del radio de cobertura de un nodo que ya está transmitiendo pero, cada uno de ellos, se encuentra fuera del radio de cobertura del receptor del otro nodo. Así, se reduce la tasa de transferencia de datos (*throughput*) de la red al retardar de forma innecesaria la transmisión del nodo.

Los protocolos de capa MAC se pueden clasificar en tres grandes clases según traten los problema anteriores [19]:

- Clase 1: prohíben las transmisiones simultáneas dentro del radio de cobertura del nodo transmisor. Estos protocolos no resuelven los problemas del nodo oculto ni del nodo expuesto. Un ejemplo típico es el protocolo CSMA/CA (*Carrier Sense Multiple Access with Collision Avoidance*) sin reserva [4].

- Clase 2: prohíben las transmisiones simultáneas dentro de los radios de cobertura del nodo transmisor y del nodo receptor. Estos protocolos resuelven el problema del nodo oculto pero no del nodo expuesto. Ejemplos típicos son los protocolos CSMA/CA con reserver, MARCH (*Multiple Access with Reduced Handshake*) [20] y S-MAC (*Sensor-MAC*) [21].

- Clase 3: prohíben la transmisión simultánea dentro del radio de cobertura del nodo receptor y hacia nodos que se encuentran dentro del radio de cobertura del nodo transmisor. Estos protocolos resuelven los problemas del nodo oculto y del nodo expuesto pero, para ello, requieren una mayor señalización previa a la transmisión. Ejemplos de esta clase son los protocolos RBCS (*Receiver-Based Channel Selection*) [22] y DBTMA (*Dual Busy Tone Multiple Access*) [23].

Capa de Red.

Esta capa es la responsable del encaminamiento en Rede Ad Hoc Inalámbricas. El objetivo de estos protocolos es el de encontrar una ruta que conecto a los nodos origen y destino.

Un protocolo de encaminamiento debe satisfacer ciertos objetivos [24] como una minimización de costes (por ejemplo, retardo), capacidad multisalto, mantenimiento dinámico de topología y eliminación de bucles. Además, pueden admitir diversos modos de operación [5] (distribuido, bajo demanda, activo y de periodo de sueño). Con estos propósitos, se ha desarrollado un amplio número de protocolos de encaminamiento que pueden agruparse según diversos criterios [25] [26]. Por ejemplo:

- Si el criterio utilizado es la función que desempeña cada nodo de la red en el encaminamiento, se distinguen dos tipos de protocolos: uniforme y no uniforme. En los protocolos de tipo uniforme todos los nodos tienen las mismas características y desempeñan iguales funciones. En los de tipo no uniforme, propios de estructuras jerárquicas, no todos los nodos tienen las mismas funciones.

- Si el criterio de clasificación es el procedimiento para descubrir y mantener la ruta, se distinguen tres tipos de protocolos: activo, reactivo e híbrido. Los de tipo activo construyen tablas de rutas actualizadas. Los reactivos sólo construyen cada ruta en el momento en el que un nodo necesita establecer una comunicación. Por último, los de tipo híbrido suponen una mezcla de los anteriores y se utilizan generalmente en protocolos no uniformes.

De entre las rutas que conectan fuente y destino, no todas son igualmente eficientes. Así, para elegir un protocolo adecuado, surge la necesidad de definir qué se entiende por eficiencia y, en consecuencia, una métrica para medirla. El parámetro de eficiencia más básico es la capacidad de construir un camino entre origen y destino lo más cercano a la línea recta como sea posible. Esta capacidad se denomina Eficiencia de Encaminamiento (o directividad) en [27] y agrupa la contribución de diversos factores (como las características del protocolo de encaminamiento o la topología de la red), ofreciendo una descripción analítica y global del encaminamiento. En [27] se describe el marco teórico que sustenta este parámetro y la metodología para su obtención en una determinada red inalámbrica (dada la topología, el protocolo de encaminamiento, la potencia de trasmisión...) mediante simulaciones.

Para poder diseñar algoritmos de encaminamiento eficientes, los simuladores deben tener en cuenta una modelo del sistema que contemple el funcionamiento en estas tres capas en su conjunto. Sólo si se tiene en cuenta la complejidad de las mismas se podrán diseñar algoritmos de encaminamiento que, si se implementan sobre una Red Ad Hoc Inalámbrica real, obtengan buenos resultados.

No todos los simuladores de redes inalámbricas son válidos para la simulación de las Redes Ad Hoc Inalámbricas, ya que muchos sólo contemplan el modelado del canal inalámbrico (mediante modelos de propagación analíticos o empíricos), es decir, capa Física y, como mucho, la capa de acceso al medio MAC. Por lo tanto, el objetivo de este proyecto es la construcción de un simulador con el objetivo final de poder ser utilizado en el diseño de protocolos de encaminamiento. Los dos requisitos fundamentales que se van a exigir son el cálculo de las pérdidas por propagación, es decir, la simulación del medio físico, y el cálculo de la interferencia, es decir, capa MAC unida a la capa Física.

En la siguiente sección 1.3 se enumeran los simuladores más utilizados para realizar un estudio de capacidades y poder extraer las capacidades de cada uno de ellos y unificarlas en el simulador que se va a construir.

1.2.2 Análisis y Diseño a través de Capas

En [28] se proporciona una definición de un diseño multicapa o a través de capas (*cross-layer*) en telecomunicaciones: *"Protocol design by the violation of a reference layered communication architecture is cross-layer design with respect to the particular layered architecture"*. Según esta definición, un diseño a través de capas incumple las estrictas interconexiones entre capas adyacentes, propias de las tradicionales arquitecturas por capas. Hoy día, esta definición se ha visto ampliada y también se consideran diseños a través de capas aquellos que mantienen la tradicional estructura por capas pero amplían las posibilidades de interacción entre ellas. Tanto arquitecturas, como protocolos y análisis de prestaciones pueden plantearse desde un punto de vista multicapa.

Esta perspectiva a través de capas resulta especialmente indicada para Redes Ad Hoc Inalámbricas y WSN debido a los siguientes factores [3]:

1. Aspectos multicapa: en estas redes se busca conseguir diversas prestaciones que resultan transversales en el tradicional diseño en capas. La estricta separación en capas no permite suficiente interacción como para tomar decisiones conjuntas que optimicen estas prestaciones. Un ejemplo de estos aspectos multicapa es la escasez de recursos, como la energía, de los nodos. La necesidad de extender el tiempo de vida de las baterías de los nodos hace necesaria una eficiente colaboración entre dichos nodos. Un enfoque multicapa, en el que las variables relacionadas con la energía de los nodos intervengan en diversas capas, puede dar lugar a un uso más eficiente de los recursos energéticos.

2. Estado distribuido: el estado de la red se encuentra distribuido en sus nodos (en lugar de existir estaciones base como en las redes tradicionales). Cada nodo tiene un cierto conocimiento (incompleto) de la red y debe tomar decisiones en función de esta información. Debido a esto, resultan útiles los algoritmos multicapa distribuidos que puedan ser utilizados por los nodos de la red según su información local. Así, en estas redes los nodos colaboran para alcanzar un objetivo global de la red, pero lo hacen tomando decisiones locales.

3. Movilidad de los nodos: afecta a factores de la capa Física (influye en el nivel de potencia interferente determinado por la cercanía de los nodos vecinos), de la capa de Enlace (influye en el ordenamiento de las transmisiones) y de la capa de Red (al variar los nodos vecinos). Por tanto, resulta necesario un análisis multicapa para la gestión de recursos en entornos móviles.

4. Propiedades de los enlaces inalámbricos: más propensos a la variación de interferencias y a la aparición de errores causados por el canal. De esta forma, puede resultar de interés la construcción de protocolos de capas superiores que tengan en cuenta estas propiedades. Por ejemplo, un protocolo de encaminamiento puede buscar una nueva ruta si detecta que el canal en un enlace está sufriendo una pérdida excesiva de calidad.

5. Nuevas modalidades de comunicación: la difusión (*broadcast*) natural que se produce en el canal inalámbrico, puede aprovecharse para el diseño de nuevos esquemas de comunicación. Por ejemplo, los nodos pueden aprovechar las transmisiones de nodos próximos, no directamente dirigidas hacia ellos, para evaluar el estado del canal respecto a los nodos transmisores o para colaborar en esa comunicación.

6. Decisiones de diseño específicas de cada aplicación: existen diferentes criterios de diseño (maximizar el tiempo de vida de los nodos, minimizar el retardo de transmisión, minimizar el error...) y todos ellos son inherentemente multicapa.

En la actualidad, las estrategias de diseño multicapa se aplican en distintos escenarios de Redes Ad Hoc Inalámbricas, como:

- Redes Ad Hoc con Acceso Múltiple por División en el Tiempo (TDMA, *Time Division Multiple Access*): Son redes con separación de canales TDMA y con una serie de asunciones comunes: sincronización (entre los relojes internos de los nodos), nodos estacionarios, comunicaciones iniciadas por el nodo origen (y no continuadas en el tiempo), con limitaciones en potencia (la potencia de transmisión es un parámetro a optimizar) y antenas isótropas.

  Otra asunción de la mayoría de desarrollos [29], es la utilización de un canal de control dedicado, a través del cual se realizan las reservas y el ordenamiento de los periodos de transmisión de los nodos.

  En algunas ocasiones, los desarrollos [30] también asumen que los nodos disponen de información geográfica (región donde se encuentran localizados) como ayuda para el encaminamiento.

- Redes Ad Hoc UWB (*Ultra-Wideband*): Estas redes utilizan una técnica de transmisión de espectro ensanchado, por lo que la calidad de la señal recibida depende de la presencia de transmisores activos en las proximidades del receptor. Los desarrollos en este tipo de redes [31], [32], buscan asignar los parámetros de transmisión óptimos para los nuevos enlaces preservando los requisitos de calidad de los enlaces ya activos.

- Redes Ad Hoc Multimedia: Los requisitos de alta tasa de transmisión y bajo retardo de las aplicaciones multimedia son un complejo objetivo en redes Ad Hoc móviles. En [33] se propone una interacción multicapa entre el encaminamiento (capa de Red) y las capas superiores para aumentar la disponibilidad de datos en aplicaciones multimedia.

- Redes Ad Hoc generales: También existen desarrollos en redes generales en las que no se realizan asunciones en cuanto a la sincronización temporal entre los nodos, la tecnología de capa Física o la aplicación objetivo de la red. Por ejemplo, en [34] se considera el problema de maximizar la utilización del ancho de banda en Redes Inalámbricas Multisalto (MHWN, *Multihop Wireless Network*).

Además de los desarrollos anteriores para Redes Ad Hoc Inalámbricas en sentido amplio, también existen desarrollos específicos para WSN. Las diferencias con los anteriores son tres asunciones: un objetivo común para el conjunto de la red; la existencia de flujos de datos que se dirigen hacia uno o pocos destinos; y, en la mayoría de los casos, nodos estacionarios. De esta forma, se distinguen dos escenarios según el tipo de acceso al medio de las WSN:

- WSN con TDMA: Son redes que cumplen las mismas asunciones ya citadas para las Redes Ad Hoc con TDMA. En el caso de las WSN hay que tener en cuenta un mayor coste en la sincronización de los nodos (sobre todo en redes densas o en redes dinámicas). En este escenario se han realizado desarrollos como [35], [36], [37], [38].

- WSN sin TDMA: Por ejemplo, redes de monitorización en las que los nodos sensores envían su información de forma infrecuente y asíncrona. Desarrollos como los presentados en [39], [31] y [40], son aptos para este tipo de WSN.

Relacionado con el creciente interés en el diseño de sistemas de comunicaciones utilizando una estrategia multicapa, han surgido dos ámbitos de investigación y desarrollo en Redes Ad Hoc Inalámbricas: la Comunicación Cooperativa y la Codificación de Red.

1.2.3 Comunicación Cooperativa

Se trata de una de las áreas de investigación de mayor crecimiento, clave para el uso eficiente del espectro en las comunicaciones futuras [41]. La Comunicación Cooperativa se caracteriza porque los nodos de la red comparten sus recursos[1] con otros nodos vecinos, con el objetivo de mejorar las prestaciones de la red a nivel global. Como consecuencia de la cooperación entre nodos, pasamos de tener un enlace directo (un canal) a un conjunto de enlaces (una red de nodos) entre origen y destino. Debido a esto, las prestaciones antes consideradas en el enlace directo, ahora deben analizarse de extremo a extremo. Las redes malladas suponen un gran campo de aplicación para este tipo de comunicaciones.

Cronológicamente, la primera forma de cooperación entre nodos, y también la más sencilla, es la ruta multisalto que permite la comunicación entre dos nodos cuando el enlace directo punto a punto entre ellos no es viable. La ruta multisalto es una cadena de enlaces punto a punto, desde el nodo origen (S) de la transmisión hasta el destino final (D), y donde cada nodo intermedio retransmite la información recibida. En la figura 1.2(a) se muestra el enlace directo entre S y D, mientras que en la figura 1.2 (b) se muestra la ruta multisalto a través de un nodo cooperador (R).

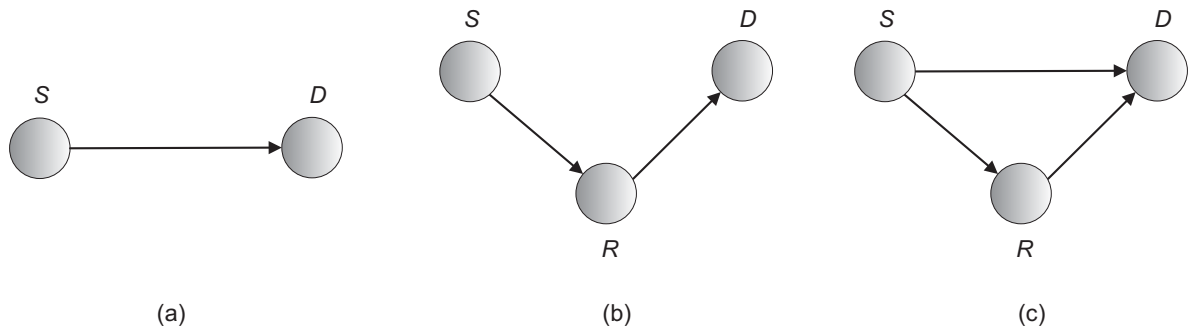


Figura 1.2 : Canal directo (a), en dos saltos (b) y canal de retransmisión (c).

En realidad, para que exista cooperación, sólo es necesario que la red se componga de más de dos nodos. Así, una red particular de tres nodos, denominada como ”canal de retransmisión” (*”relay channel”*) y que se representa en la figura 1.2 (c), se considerada la red fundamental en Comunicación Cooperativa. Esta red ha sido ampliamente estudiada en la literatura, especialmente en el ámbito de Teoría de la Comunicación, desde su presentación [42], [43], [44], y dio origen a la línea de investigación en Comunicación Cooperativa que sigue en auge en nuestros días [45], [46], [47], [48], [49], [50], [51], [52].

[1] Recursos a compartir son, por ejemplo, la potencia de transmisión o la capacidad de cómputo de los nodos.

1.2.4 Codificación de Red

En las redes de comunicación tradicionales, los paquetes de datos de información permanecen inalterados a lo largo de la ruta entre origen y destino (excepto cambios debidos a errores producidos en las retransmisiones). Sin embargo, en los últimos años y con objeto de mejorar las prestaciones en la red, ha surgido un nuevo tipo de comunicaciones, conocida como Codificación de Red, en la que unos paquetes de datos se combinan con otros en los nodos intermedios que componen la ruta. Debido a esta característica, la Codificación de Red puede considerarse una Comunicación Cooperativa a nivel de capa de Red o una codificación a nivel de paquete.

Según [53], la Codificación de Red aporta las siguientes ventajas:

- Aumento de la capacidad de la red: la Codificación de Red permite reducir considerablemente el tiempo de transmisión, especialmente en comunicaciones de difusión (*multicast*) [54], [55], pero también en comunicaciones punto a punto [56], [57].
- Aumento de la robustez en las comunicaciones: al igual que la codificación tradicional, la codificación a nivel de paquete también permite disminuir los errores en la comunicación [58].

Además, partiendo de las capacidades anteriores, se pueden mejorar otras prestaciones que resultan claves en WSN como, por ejemplo, el ahorro de energía en los nodos [59].

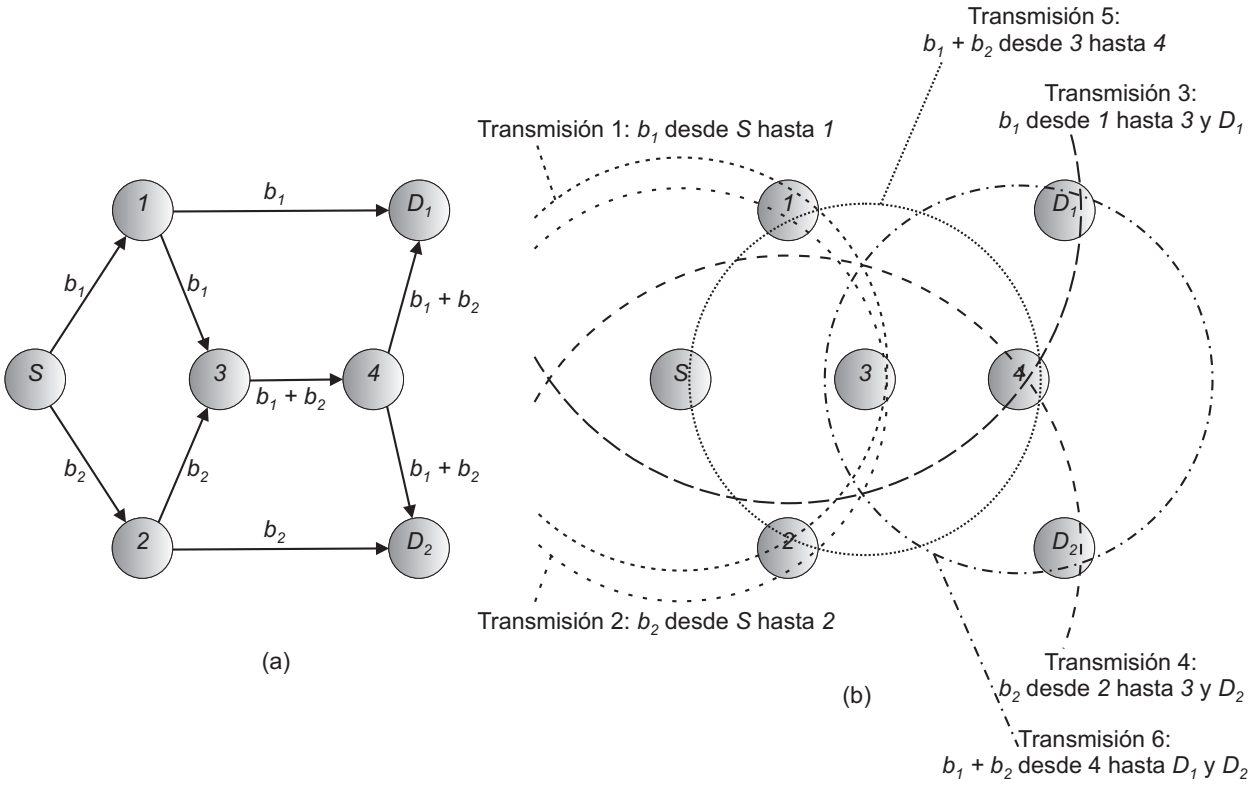


Figura 1.3 : Ejemplo canónico de Codificación de Red [1] para red cableada (a) y para red inalámbrica (b). Denotamos como $b_1 + b_2$ a la suma binaria de los bits b_1 y b_2.

Debido al muy reciente planteamiento de este tipo de comunicaciones, que apenas lleva una década en desarrollo desde su inicio en [1], en esta introducción explicamos su utilidad sobre un ejemplo canónico

que fue presentado en dicho trabajo inicial. La figura 1.3(a) representa la red de trabajo: una red cableada compuesta por siete nodos y por enlaces unidireccionales, de capacidad unitaria, libres de errores y que no interfieren entre ellos. El objetivo de dicha red es la transmisión de datos desde el nodo origen (S) hasta los dos nodos destino (D_1 y D_2). Trabajando con bits (en lugar de paquetes, para una mayor claridad en la explicación) y representando con "$+$" la suma binaria de dos bits, la figura 1.3(a) muestra cómo se puede transmitir dos bits (b_1 y b_2) de forma simultánea a D_1 y D_2, utilizando una Codificación de Red: suma de los bits en el nodo 3 y decodificación de la información en los nodos destino (D_1 recupera b_2 a partir de b_1 y $b_1 + b_2$; D_2 recupera b_1 a partir de b_2 y $b_1 + b_2$).

Gran parte de las contribuciones que siguieron al trabajo inicial, como [60], [61], se centraron en el ámbito de las comunicaciones guiadas (no inalámbricas) para alcanzar la máxima capacidad de difusión.

En el caso particular de las redes inalámbricas, este tipo de redes tiene una serie de características que se deben tener en cuenta para la aplicación de la Codificación de Red, pero que dan lugar a nuevas oportunidades [62]. Para comprender las diferencias y similitudes al considerar redes inalámbricas, presentamos la figura 1.3(b), donde se puede observar la misma topología de la figura 1.3(a) pero con enlaces inalámbricos y antenas isótropas en cada nodo. Al tratarse de una red inalámbrica, no es posible la transmisión y recepción simultánea en un nodo; ni que un nodo sea capaz de recibir dos paquetes al mismo tiempo. Debido a esto, se hace necesario un ordenamiento de las transmisiones como el indicado en la figura 1.3(b).

Obsérvese que la Codificación de Red inalámbrica resulta muy dependiente de factores como el radio de cobertura o el posicionamiento físico de los nodos. Aún así, tiene potencial suficiente para mejorar las prestaciones de las redes inalámbricas en términos de aumento de la capacidad de difusión o disminución de la BER extremo a extremo [41] [63], [64], [65], [66], [67], [68].

1.3 Simuladores de Redes Ad Hoc: Ventajas e inconvenientes

En esta sección se enumeran y exponen las características fundamentales de los simuladores de redes de comunicaciones existentes en la actualidad.

1.3.1 Network Simulator, *ns-2*

network simulator (ns-2) es un simulador, de *código abierto*, de redes de comunicaciones creado dentro del *Proyecto* VINT (*Virtual InterNetwork Testbed*) en el que colaboran varias instituciones [69]. Esta herramienta provee las capacidades necesarias para poder añadir nuevas funcionalidades que se adapten a la simulación de nuevos sistemas de comunicación. Inicialmente, *ns-2* se concibió para la simulación de redes cableadas, pero con el avance de la tecnología, han aparecido nuevos módulos que permiten la simulación de nuevos protocolos y de sistemas inalámbricos.

ns-2 cubre un gran número de aplicaciones, protocolos y tipos de elementos de red y modelos de tráfico [70]. El simulador está basado en dos lenguajes de programación: el corazón del sistema que está escrito en C++, y el intérprete de OTcl que es el formato de definición de los escenarios de simulación. El usuario define la topología de la red a simular utilizando las librerías proporcionadas en OTcl y *ns-2* establece una correspondencia con los objetos definidos en C++. Los resultados de la simulación se almacenan en

formato de trazas en archivos de texto que deben ser procesados a posteriori para su interpretación.

ns-2 es un simulador basado en eventos discretos, en el que el avance del tiempo depende del tiempo de ocurrencia de los eventos que son ejecutados por un programador de eventos. Un evento es un objeto en la jerarquía C++ definido por un identificador único, un tiempo de ejecución y una referencia al objeto que debe interpretar el evento. El programador de eventos mantiene una estructura de datos ordenada de los eventos y los ejecuta uno a uno invocando el manejador de eventos [70].

Una de las extensiones fundamentales de *ns-2* es la capacidad para simular redes inalámbricas. El modelo inalámbrico implementado está basado en nodos móviles y otras características que permiten la simulación de Redes Ad Hoc multisalto, WLANs, etcétera. La diferencia de los nodos móviles respecto a los nodos fijos es la capacidad de cambiar su posición y que no están unidos mediante enlaces a otros nodos.

Existe una implementación específica para el protocolo IEEE 802.15.4 para *ns-2*. Esta implementación tiene en cuenta aspectos como tasas de transferencia o consumos energéticos pero no va más allá. El enfoque del simulador *ns-2*, en general, está orientado al campo de la Telemática más que al de Teoría de Señal y, aspectos fundamentales de esta última área, no son tenidos en cuenta. Desde el punto de vista del diseño y optimización de protocolos de encaminamiento, en el área de Teoría de Señal, debe tenerse en cuenta un factor muy importante que es la **interferencia**. *ns-2* no proporciona mecanismos que permitan la evaluación de este parámetro de una manera sencilla. Adicionalmente, otra carencia importante, es la escalabilidad de las simulaciones. En el Departamento de Teoría de la Señal y Comunicaciones se tiene una amplia experiencia en el uso de *ns-2* en simulación de Redes Ad Hoc Inalámbricas. De esta experiencia se extrae la conclusión de que Network Simulator no es un buen candidato para el estudio de Redes Ad Hoc Inalámbricas Densas.

1.3.2 JIST/SWANS, *Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator*

JiST es un simulador de altas prestaciones basado en eventos discretos que se ejecuta sobre una máquina virtual de Java. Se trata de un prototipo que nace de un proyecto más ambiciosos de construcción de simuladores basados en eventos denominado *simulación basada en máquinas virtuales*, que unifica los sistemas tradicionales y los diseños de simuladores basados en lenguaje.

Por otro lado, SWANS, es un simulador de redes inalámbricas escalable, construido en base a la plataforma JiST. Se organiza de manera independiente de los componentes software que lo integran. Sus capacidades son similares a *ns-2* o GloMoSim, pero es posible simular redes con un mayor número de nodos. SWANS potencia el diseño de JiST para alcanzar los objetivos de máximo rendimiento, ahorro de memoria y ejecución de aplicaciones de red Java sobre simulaciones de redes de comunicaciones. Además, SWANS, implementa una estructura de datos para un cálculo eficiente de los modelos de propagación de señal en entornos inalámbricos. Algunos componentes implementan diferentes tipos de aplicaciones: protocolos de encaminamiento y acceso al medio, conectividad de red; transmisión, recepción radio y modelado del ruido; modelos de propagación de señal y desvanecimiento; modelos de movilidad, etcétera.

SWANS, gracias a JiST y su modelo, tiene un alto grado de paralelización ya que la comunicación entre entidades es muy eficiente. No se incurre en costes de serialización, copia o cambio de contexto entre entidades co-localizadas ya que los objetos Java se pasan por referencia. Los paquetes de red simulados son una cadena de objetos anidados que emulan las cabeceras que añadiría una pila de protocolos de

red. Además, no se replican los paquetes, sino que dos entidades, la emisora y receptora, contienen la referencia en memoria del dicho paquete, ahorrando memoria de manera efectiva.

1.3.3 OMNeT++

OMNeT++ más que un simulador propiamente dicho es una librería o plataforma basada en C++ para construir simuladores de redes tanto cableadas como inalámbricas. Extensiones específicas para los diferentes tipos de redes, como Redes Ad Hoc Inalámbricas, se desarrollan como módulos adicionales.

La simulación que implementa OMNeT++ está basada en eventos discretos. Estos eventos son ordenados por una marca temporal que indica el instante en que se recibe el evento. El evento con el menor tiempo es el primero en ser ejecutado. Ante mensajes con el mismo tiempo, la ejecución se realiza por prioridad de ejecución. Es decir, los eventos, además de marca temporal, tienen prioridad.

Los escenarios de simulación se describen utilizando el lengua NED, *NEtwork Description*. Este lenguaje permite al usuario definir módulos que pueden unirse formando módulos complejos. Estos módulos complejos se pueden etiquetar como *redes*, es decir, modelos de simulación autodefinidos. Los canales son otro tipo de componentes que pueden ser, también, creados como módulos complejos.

OMNeT++ ofrece la posibilidad de realizar simulaciones de manera distribuida. La aproximación que implementa OMNeT++ para la simulación distribuida es una aproximación conservadora, ver capítulo 3, utilizando el algoritmo de *mensajes nulos* (capítulo 3).

Este simulador, en su versión distribuida, presenta la aproximación más semejante a la propuesta en este trabajo con la diferencia de que la filosofía seguida es optimista en lugar de conservadora.

1.3.4 Castalia

Castalia es un simulador específico de Redes de Sensores Inalámbricas (WSN) y de Redes de Área Corporal (BAN) y, en general, de redes de bajo consumo energético. Está basado en OMNeT++ y su objetivo es el diseño y optimización de algoritmos de encaminamiento utilizando modelos reales de canales inalámbricos. Las principales características de Castalia son:

- Modelos de canal avanzados basados en datos empíricos.
  - Se define el mapa de pérdidas de propagación y no únicamente las conexiones entre nodos.
  - Modelos complejos para la variación temporal de las pérdidas de propagación.
  - Soporte para movilidad de los nodos.
  - La interferencia se maneja como potencia de señal recibida y no como una característica separada.
- Modelo radio avanzado basado en dispositivos radio reales de bajo consumo.
  - Probabilidad de recepción basada en la SINR, tamaño de paquete, tipo de modulación. Se soportan varios tipos de modulación y se ofrece la posibilidad de definir nuevas modulaciones basándose en la curva SNR-BER.

  - Potencia de transmisión por nodo.
  - Detección de portadora flexible.
  - Estados con diferente patrón de consumo energético y retardos de conmutado entre ellos.
- Modelos de detección extendidos.
  - Modelado de capa Física flexible.
  - Ruido de recepción, bias y consumo de potencia
- Desviación del reloj de los nodos, consumo de potencia de CPU.
- Disponibles protocolos MAC y de encaminamiento.
- Con capacidad de adaptación y expansión. El usuario puede implementar sus algoritmos e importarlos en Castalia.

1.3.5 GloMoSim, *Global Mobile Information Systems Simulation Library*

GloMoSim es un simulador de redes cableadas e inalámbricas escalable. Está basado en Parsec [71], lo que le dota de buenas capacidades de simulación basada en eventos paralelizable. GloMoSim construye todo una torre OSI de protocolos como funcionalidad básica, por lo que se pueden realizar simulaciones a cualquier nivel de la torre.

El hecho de basar la construcción de GloMoSim en Parsec, proporciona la posibilidad de paralelizar/distribuir los cálculos. Cada nodo de la red puede definirse como una entidad Parsec independiente, es decir, un proceso autónomo. Uno de los objetivos de GloMoSim es la simulación de redes con número elevado de nodos, pero la construcción anterior no es válida ya que los requerimientos de memoria se disparan. Para resolver dicho problema, se introducen particiones en la red, de manera que cada partición es una entidad independiente compuesta de múltiples nodos. Cada partición está asociada a una región geográfica de la red y dichas particiones son definidas por el usuario.

La funcionalidad de GloMoSim está muy restringida en favor de su versión comercial: *Qualnet* [72]. Qualnet explota las capacidades de los procesadores multi-núcleo por lo que es posible simular redes de miles de nodos.

GloMoSim es una de las mejores opciones para realizar simulaciones masivas de Redes Ad Hoc Inalámbricas ya que, por concepción, contempla la paralelización/distribución de la computación. El problema fundamental es que (a) la versión gratuita es muy limitada en funcionalidad (de hecho no permite computación distribuida) y (b) la versión completa es comercial con elevado coste.

1.3.6 TOSSIM, *TinyOS Simulator*

TOSSIM es una herramienta desarrollada para simular dispositivos TinyOS. TOSSIM escala a simular miles de nodos y ejecuta directamente el código de TinyOS de manera que se puede probar directamente el código que se ejecuta sobre los dispositivos.

TOSSIM simula la torre de protocolos a nivel de bit lo que permite trabajar con protocolos y aplicaciones a bajo nivel.

El principal inconveniente de este simulador es que está específicamente desarrollado para dispositivos basados en TinyOS, los cuales, son minoría. El lenguaje de programación en el que se basa para la programación de la lógica de las motas es nes-C, una variante de C específica para pequeños dispositivos. El aprendizaje de este lenguaje, junto con las librerías que proporciona TinyOS no es inmediato a partir del conocimiento del lenguaje C. Se trata, por lo tanto, de un simulador específico para este tipo de dispositivos perdiendo el carácter general de los otros simuladores presentados.

1.3.7 SAHNE, *Simulation Environment for Ad Hoc Networks*

SAHNE es un simulador de redes inalámbricas y de interacción de nodos en una Red Ad Hoc. Proporciona un interfaz gráfico basado en LEDA (*Library of Efficient Datatypes and Algorithms*).

Originalmente, SAHNE, se diseño con el objetivo de simular algoritmos de control de topología con redes de comunicación direccionales, aunque pronto se incluyo la comunicación omnidireccional. La simulación de la capa física puede realizarse mediante dos modelos: radiofrecuencia (RF) o infrarrojos (IR). En el modelo de RF, la señal se propaga de manera omnidireccional de acuerdo con las leyes de propagación en espacio libre pero con un exponente de pérdidas de propagación variable ($2 \leq \alpha \leq 4$). El modelo IR, un mismo nodo puede tener varios transceptores cada uno orientado en una dirección diferente. El modelo de propagación IR es el propuesto por Kahn y Barry en 1997. En ambos modelos, IR y RF, la potencia de transmisión de cada nodo puede ser diferente. La potencia recibida se determina en el nodo receptor. Además, se tienen en consideración las posibles interferencias de señales transmitidas en el mismo instante temporal. Este hecho es fundamental para poder realizar un diseño de protocolos de encaminamiento a través de capas considerando las tres primeras capas de la torre de protocolos OSI.

SAHNE presenta funciones para el cálculo de la calidad de los algoritmos de encaminamiento basándose en medidas de congestión, retardo y consumo de energía definidas por Meyer auf der Heide et al. [73]. Estas medidas son indicadores genéricos de las prestaciones de los algoritmos de encaminamiento. Se define el máximo número de caminos mediante la carga de un enlace de manera que si se suma la carga de todos y cada uno de los enlaces interferentes se puede obtener el grado de congestión de un enlace determinado. La congestión de la red se mide como la congestión máxima de todos los enlaces en dicha red. La energía es la potencia consumida para el funcionamiento de la red y la transmisión de paquetes. El camino más largo de una ruta se denomina *dilatación* y proporciona una medida del tiempo máximo de encaminamiento. SAHNE calcula caminos óptimos en cuanto a número de saltos y consumo energético, determina los enlaces interferentes basándose en las relaciones señal-a-ruido (SNR) dado un modelo de propagación. Con estos cálculos se realizan las estadísticas de congestión, retardo y energía.

El simulador soporta movilidad de nodos y proporciona algunos modelos de movilidad aleatorios. Además, el patrón de movilidad puede ser definido por el usuario mediante un archivo de escenario en el que las trayectorias son definidas por puntos predefinidos (*waypoints*) y su velocidad.

1.3.8 SEAMCAT, *Spectrum Engineering Advanced Monte Carlo Analysis Tool*

SEAMCAT es una herramienta software de simulación basada en el Método de Monte Carlo. Este simulador ha sido desarrollado dentro del marco de la Conferencia Europea de Correos y Telecomunicaciones (CEPT, *European Conference of Postal and Telecommunication*), en concreto por la Oficina de Comunicaciones

Europea (ECO, *European Communications Office*).

Este simulador está orientado, fundamentalmente, al modelado estadístico de diferentes escenarios de interferencia radio. Los resultados arrojados permiten la optimización de la compartición del espectro radioeléctrico mediante estudios de compatibilidad de bandas de frecuencia adyacentes. El modelo estadístico implementado en SEAMCAT está publicado en en ERC Report 68 [74].

SEAMCAT permite el análisis de escenarios complejos de compatibilidad ofreciendo importantes características:

- Cuantificación de los niveles de interferencia. El nivel de interferencia entre dos sistemas se expresa en términos de probabilidad de capacidad de recepción de un sistema frente a la interferencia del otro.
- Consideración de las distribuciones espaciales y/o temporales de las señales recibidas. Este parámetro sirve de ayuda a la hora de realizar una planificación de frecuencias óptima.
- Puede simularse cualquier tipo de escenario independientemente del tipo de sistema radio interferido e interferente.

Dado que el objetivo principal de SEAMCAT es el cálculo de interferencia, no se trata de un simulador útil en el ámbito de las Redes Ad Hoc Inalámbricas. Como se vio anteriormente, el diseño de las Redes Ad Hoc se enfoca desde la perspectiva multicapa. SEAMCAT nos proporciona valiosa información a nivel de capa Física únicamente. Es por ello por lo que se hace interesante, al menos, conocer los principios que rigen la simulación en SEAMCAT para poder ser aplicados en el diseño de simuladores de Redes Ad Hoc Inalámbricas multicapa.

1.4 Objetivos del proyecto

Una vez realizado un estudio de los simuladores existentes y concluido que no son aptos para la simulación de Redes de Sensores Inalámbricas densas, en este trabajo, se propone el desarrollo de una arquitectura que permita una computación distribuida sobre múltiples procesadores.

Esta arquitectura puede ser utilizada como complemento a alguno de los simuladores anteriores, de manera que se puedan aunar la potencia y versatilidad que ofrecen los mismos con la posibilidad de poder aumentar las prestaciones mediante cálculo distribuido. Este trabajo de integración conllevaría la elección del simulador más apropiado y de conocer su arquitectura, lo que nos llevaría un tiempo demasiado largo. Como alternativa se implementará un simulador sencillo, operando sobre la arquitectura distribuida, que nos permita evaluar las prestaciones y bondades del entorno de computación distribuido.

Todo el código desarrollado se pondrá a disposición de la comunidad científica para su uso y mejora a través de la licencia GPLv3 (*GNU General Public License*). El código fuente se encuentra disponible en el disco compacto anexo a este documento y también en los siguientes enlaces de la web:

- ParADisE - Parallel Architecture for Distributed Events: http://sourceforge.net/projects/warpparadise/
- PWiSeSim - ParADisE Wireless Sensor Simulator: http://sourceforge.net/projects/pwisesim/

CAPÍTULO 2

EL PROBLEMA DE LA SIMULACIÓN DISTRIBUIDA

2.1 Introducción a la simulación distribuida

En la actualidad, la simulación se convierte en una herramienta esencial a la hora de realizar el diseño de un sistema complejo. Ahorra tiempo y recursos de manera que la implementación cuasi-definitiva ya ha sido depurada en varias iteraciones previas mediante simulación [75].

Las capacidades de computación existentes hoy en día, dotan a la simulación, de una extraordinaria potencia para representar de manera, más o menos fiel, la realidad del comportamiento de un sistema. A pesar de ello, existen determinados tipos de simulaciones en las cuales la carga computacional es lo suficientemente grande para que el tiempo de simulación se haga excesivamente largo.

En el capítulo 1 se realizó una revisión de los simuladores de redes de sensores que más se utilizan en la actualidad. Se puede afirmar sin temor a equivocarse que ninguno de ellos es adecuado para la simulación a Gran Escala de Redes Densas (más de 1000 nodos). Si bien es cierto que alguno de ellos puede llegar a superar la cifra anterior para simulaciones puntuales, la obtención de resultados estadísticamente significativos es un problema de difícil solución. Para muestra, sirva el siguiente ejemplo: La simulación en ns-2 de una red triangular de 547 nodos, en los que se crean caminos entre 24.000 pares de nodos fuente-destino, con transmisión secuencial (es decir, no concurrente), durante 2 segundos de tiempo de red cada uno (necesario para la estabilización del camino encontrado), tarda más de 150 horas en un servidor biprocesador G5 de 64-bits con 4 Gb de memoria RAM y genera una traza (que deberá ser post-procesada) con un tamaño de 40 Gb. Estas mismas magnitudes de red en el simulador JIST/SWANS de la Universidad de Cornell, diseñado y optimizado para simular redes densas, reduce el tiempo de simulación en un 5 % a costa de casi triplicar la huella de memoria utilizada. Teniendo en cuenta que, para una red tan modesta como la anterior, podemos considerar que se empiezan a obtener resultados estadísticamente significativos y con un margen de confianza aceptable (mayor de un 95 %) con los resultados de 8 simulaciones como las anteriores, creemos que aún existe mucho margen para la mejora en este aspecto de la investigación.

El trabajo que nos ocupa es, por lo tanto, la mejora de las condiciones de simulación de las redes ad-hoc densas de manera que, ni el tiempo de computación se dispare, ni los recursos utilizados sean abusivos. Para ello, la propuesta es utilizar el paradigma de *computación distribuida* sobre múltiples computadoras. Para llevar a cabo esta distribución, el problema se fracciona en múltiples subproblemas y se ejecutan en diferentes procesadores.

En general, en una computación distribuida existen múltiples *agentes* [76] que colaboran para la conse-

cución de una tarea determinada. Estos agentes, son elementos tanto *hardware* como *software* y son los que permiten la ejecución de tareas. Desde el punto de vista del *hardware*, un agente es un procesador; desde el punto de vista del *software*, un agente es un proceso o *thread* como comúnmente se le denomina en los entornos de producción de código. Conceptualmente, un proceso ejecuta sus tareas dentro de un procesador, ver figura 2.1. En este punto, un procesador puede ser considerado como un elemento lógico en lugar de un elemento físico. En lo sucesivo, y como detalle de implementación, consideraremos al procesador como un elemento lógico pero que está directamente ligado a un elemento físico. Es decir, si una máquina tiene dos procesadores o CPUs, se ligarán dos procesadores lógicos a dicha máquina.

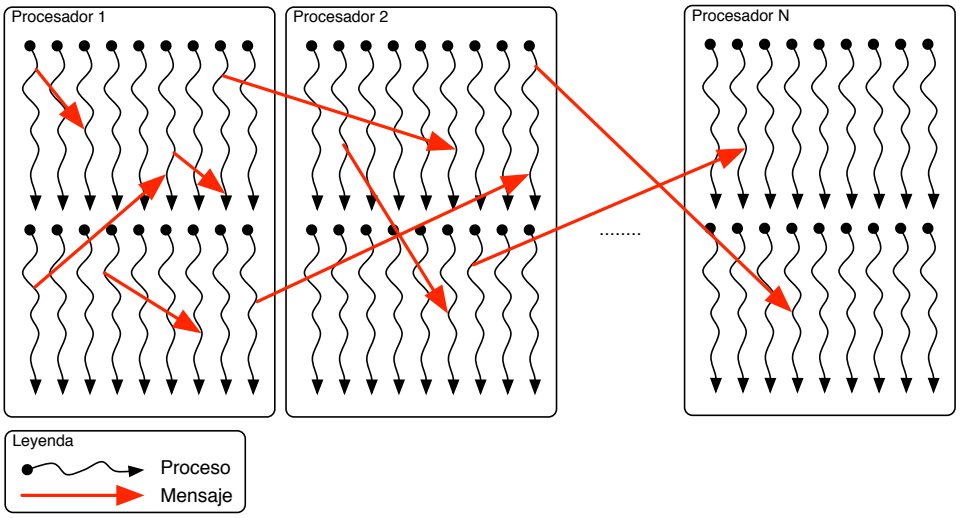


Figura 2.1 : Arquitectura de Procesos y Procesadores

Para poder implementar una simulación mediante el uso del paradigma de la computación distribuída, el problema debe ser **separable** o **cuasi-separable**. La descomposición del problema debe ser tal que los subproblemas deben poder ejecutarse de manera paralela al resto. Si el problema es completamente separable no existirán interacciones entre los diferentes subproblemas. Por el contrario, si el problema es cuasi-separable (por lo general, la gran inmensa mayoría), existen mínimas interacciones entre los diferentes subproblemas, influyendo unos en los resultados finales de los otros. En concreto, en una red de sensores existen interacciones de unos sensores con otros por lo que la separación en subproblemas no es inmediata.

Dependiendo de la naturaleza del problema, la simulación se puede concebir como un proceso basado en la ocurrencia de eventos o por un proceso continuo. En el caso de una simulación *continua*, las tareas se realizan de manera continua en el tiempo. No existen intervalos temporales durante el tiempo de simulación en los que no se realice ningún cálculo. Un ejemplo sencillo es la simulación de la codificación, transmisión, recepción y decodificación una trama de datos a través de un canal con multitrayecto. Por otro lado, en el caso de la simulación basada en eventos, las tareas se ejecutan cuando se ha producido un evento. Es decir, es éste el que provoca la ejecución de la tarea. Un ejemplo de este tipo de simulación es el nodo que ha recibido la trama anterior, que contiene un mensaje que provoca la lectura y el envío de otro mensaje a otro nodo. Es decir, dicho nodo se encuentra *en reposo* escuchando los mensajes que van destinados hacia él y únicamente entra en ejecución cuando recibe un mensaje, un evento.

2.2 Problemas del procesamiento distribuido

A pesar de que el procesamiento distribuido puede mejorar las prestaciones en simulaciones masivas presenta, como se verá, problemas que no siempre son resolubles.

El primer problema que se presenta cuando se plantea una simulación distribuida es la separabilidad. Lo habitual es que las separación en subproblemas no sea trivial y, en concreto, en las redes de sensores inalámbricas es problema de la separabilidad no es inmediato. En el funcionamiento normal de una red, los sensores envían mensajes a otros sensores bien para que el mensaje sea enrutado hacia un punto de fusión, bien porque el funcionamiento de la red exige intercambio de información entre los sensores. Esta interacción hace que los nodos sean *dependientes* y, por lo tanto, la simulación no puede realizarse sin tener en cuanta a los otros nodos.

En este punto cabe distinguir entre computación paralela y distribuida [76][75]. En la computación paralela, las unidades de procesamiento están situadas cerca unas de las otras con lo que las comunicaciones entre ellas, si las hubiera, no supondrían un gran perjuicio en el tiempo final de la simulación. En estos casos, las limitaciones entre las interacciones vendrán determinadas por el ancho de banda de los buses de datos que interconectan unas unidades de proceso con otras. En los sistemas distribuidos, las unidades de procesamiento se encuentran alejadas las unas de la otras. La comunicación entre ellas se realiza, habitualmente, mediante conexiones TCP/IP que conllevan una sobrecarga en el tiempo de procesamiento al realizar en encapsulado/desencapsulado de los datos. Nos aparece el primer problema intrínseco al procesado distribuido, que es la necesidad de comunicaciones entre los procesos sitos en diferentes puntos. Al tener que hacer uso de una red de comunicaciones, la simulación está vinculada estrechamente al estado de dicho canal de comunicaciones.

El problema más significativo cuando se habla de un procesamiento distribuido donde los diferentes procesos deben comunicarse entre sí es la **sincronización**. En un sistema distribuido no todos los procesos tienen la misma carga computacional, por lo que unos pueden avanzar más rapidamente que otros generando mensajes que deben ser transmitidos a otros procesos, posiblemente, más lentos. Por ejemplo, en una red de sensores con cientos de sensores se puede poner en marcha un algoritmo en el que existen nodos que transmiten poca información y otros que transmiten en cada iteración. Los primeros realizan la medida de la evolución de una determinada magnitud y cada 5 minutos envían un mensaje con el valor mínimo y máximo medidos en ese periodo de tiempo. Por otro lado, hay otros nodos tiene que transmitir información de la magnitud sensada cada vez que esta es medida cada segundo, generando así un paquete por segundo. En la simulación de este tipo de red, el tiempo de simulación de los nodos que transmiten un paquete cada cinco minutos avanza muy rápidamente mientras que el otro tipo de nodos lo hace más lento. En el caso de que un nodo de tipo 2 envíe un mensaje a un nodo de tipo 1, con una alta probabilidad, dicho mensaje llegará en un instante de simulación mucho más avanzado que el tiempo del mensaje recibido y por lo tanto el resultado final de la simulación será inconsistente. Independientemente del tipo de algoritmo utilizado (2.4), se pueden producir estos incumplimientos del principio de causalidad. Si no se establecen unos principios de sincronización los resultados de la simulación no serán los correctos.

En el caso que nos ocupa, el objetivo es la simulación del comportamiento de redes de sensores inalámbricas densas. En esta aproximación, simular tal cantidad de elementos en un único procesador supondría un tiempo muy elevado, como ya se vio en el ejemplo de la sección 2.1, y la simulación no sería de provecho. Asumiendo este hecho, se puede proceder con un particionamiento del problema. La red completa de

sensores se fracciona *aleatoriamente* formando múltiples particiones disjuntas. Cada subconjunto podrá simularse de manera independiente quedando la totalidad de las comunicaciones entre nodos dentro de la misma máquina y disminuyendo parcialmente el tráfico en la red. El problema aparece en los bordes de dichos subconjuntos. Las comunicaciones entre ellos, ver figura 2.2, sí serán numerosas por pertenecer a los vecindarios y por lo tanto se incrementará el tráfico en la red y con ello el retardo en la recepción de los mensaje.

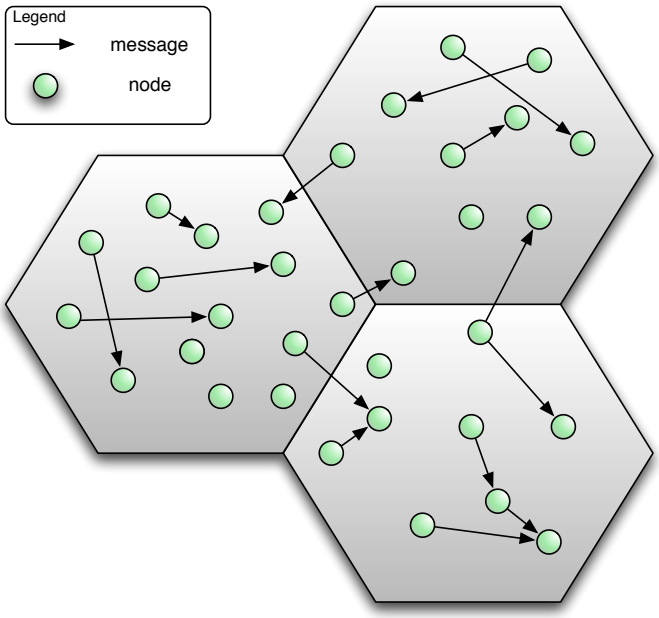


Figura 2.2 : Intercambio de mensajes en una red particionada

La asignación de recursos se lleva a cabo mediante la concesión de un procesador por cada una de las particiones y un proceso por cada nodo. Cuantos más procesadores se dispongan mayor podrá ser el número de particiones a realizar. En esta situación se pueden explotar las capacidades de los terminales de cómputo basados en GPUs, en los que se disponen de numerosos procesadores por tarjeta[1]. El aumento del número de particiones conlleva un mayor aumento de mensajes inter-procesador, lo cual implica que si los procesadores están en diferentes máquinas se tendrá un retardo por la transmisión. Por el contrario, si están en la misma máquina ese retardo no aparecerá .

La definición de *sincronía* y *asincronía* [75] puede verse desde el punto de vista de paso mensajes como el comportamiento de un proceso cuando envía un mensaje a otro. En los sistemas síncronos, un proceso envía un mensaje a otro y espera a obtener un resultado antes de continuar con su ejecución. En los sistemas asíncronos, el proceso envía el mensaje y continua su ejecución sin tener en cuenta el resultado del mismo.

La sincronía entre procesos se erige como un problema fundamental para poder llevar a cabo correctamente una ejecución distribuida. Cuando se habla de algoritmos de procesado distribuido síncronos, se habla de un reloj global para todos los procesos involucrados. Este reloj marca el avance del procesado para todos los procesos por igual, lo que permite una sincronización de todos ellos basado en este valor. Cómo se comparte el valor de este reloj es un problema que se debe tratar aparte y que está resuelto

[1]Este trabajo se planteará como una mejora de rendimiento en el capítulo de trabajos futuros.

[77][78] mediante los algoritmos de sincronización de relojes en red (Network Time Protocol [79]).

En el caso de algoritmos asíncronos, cada proceso se ejecuta con su propio reloj. No existe una referencia común a todos ellos. Además, la transmisión de mensajes tiene retardos no controlados a diferencia de los síncronos[2]. Cuando un proceso se ejecuta regido por su reloj local, éste, si no interacciona con otros procesos terminará su ejecución y quedará bloqueado a la espera de que el resto termine. Por el contrario, si interacciona con otros la simulación puede terminar en un estado inconsistente si se han producido violaciones del principio de causalidad.

En una simulación en la que existen comunicaciones entre los distintos procesos, los mensajes tendrán un efecto en el estado de ejecución del proceso, por lo que la ordenación de los eventos tiene importancia. En el ejemplo de la figura 2.3 se observan varios procesos, algunos de los cuales no interaccionan con el resto (Proceso 1 y Proceso 2) y los demás que sí lo hacen. En dicho ejemplo se ha llevado al extremo el hecho de que cada proceso se ejecuta en procesadores diferentes ubicados en lugares diferentes. Si los procesos se sincronizan de manera que no pueden ejecutar un evento hasta que no reciben el correspondiente mensaje se produce un bloqueo de los mismos en espera de ellos. Por ello, los eventos con fondo verdoso podrían ejecutarse pero no pueden hacerlo hasta que no reciben el mensaje. Este bloqueo provoca una penalización en el tiempo de simulación que se puede observar en dicha figura como rectángulos con fondo grisáceo. En este tipo de algoritmo, incluso los procesos Proceso 1 y Proceso 2, a pesar de no intercambiar mensajes, van a estar sincronizados y van a ser bloqueados. Este tipo de algoritmos, como se verá más adelante se denominan algoritmos conservadores.

Otra fórmula para realizar una simulación consiste en relajar el chequeo en cada iteración del principio de causalidad. En este caso, los procesos se ejecutan de manera independiente sin esperar a los demás. En el ejemplo de la figura 2.4 se muestra cómo funcionaría un algoritmo optimista.

- los procesos Proceso 1 y Proceso 2 no tienen interacción con el resto por lo que puede ejecutarse sin ningún problema hasta su finalización.
- los otros procesos sí interaccionan entre si, por lo que aparecen problemas de cumplimiento del principio de causalidad. Los eventos con fondo verdoso son aquellos que son susceptibles de violar dicho principio.
- el Proceso 4 ejecuta un evento de este tipo antes de recibir el mensaje correspondiente y está ejecutando el siguiente evento cuando lo recibe. En este punto, ya ha enviado un mensaje al Proceso N como consecuencia de haber ejecutado dicho evento, provocando la ejecución de un nuevo evento en el Proceso N.
- el mismo problema ocurre entre el Proceso N y el Proceso 3. Cuando el Proceso 4 recibe el mensaje comprueba que su tiempo de ejecución está muy por delante del tiempo del mensaje recibido, por lo que ha violado la causalidad. Esto provoca que debe rehacer los cálculos empezando en el instante de recepción del mensaje (flechas rojas).
- la violación se propaga hacia los procesos Proceso N y Proceso 3. Aquí el tiempo de penalización tiene naturaleza diferente, que es el tiempo que se pierde rehaciendo los cálculos cuando se viola la causalidad del sistema.

[2]En los algoritmos síncronos los retardos de transmisión no son controlables pero al tener una referencia común que dicta los tiempos de ejecución, mientras no llegan todos los mensajes a sus correspondientes destinos no se produce la siguiente iteración.

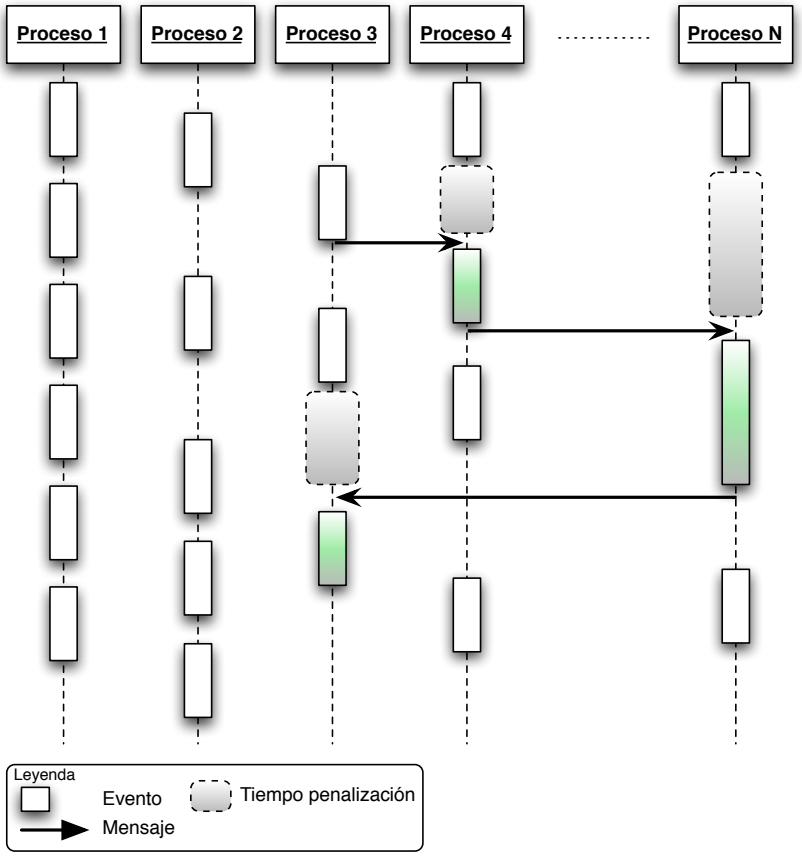


Figura 2.3 : Penalización de un algoritmo conservador.

La ordenación de tiempos se erige obligatoria para la consecución de resultados satisfactorios y causales. La selección del siguiente evento a ejecutar está directamente relacionada con la forma en que se ordenan los eventos en las distintas colas de los distintos procesos. En la literatura existen infinidad de algoritmos para tal fin. El primero en analizar este problema a finales de la década de los 70 fue Leslie Lamport [80], generando toda una corriente posterior cuyas últimas contribuciones se extienden hasta finales de la primera década del siglo XXI.

2.3 Simulación basada en eventos

Después de analizar el comportamiento de un sistema y proponer una solución mediante simulación basada en eventos, no es trivial encontrar dicha solución. Ya no a la hora de implementarla, sino a la hora de que los resultados obtenidos sean los correctos.

En una simulación secuencial (no distribuida) típica de eventos, se suelen utilizar dos estructuras de datos [81]: las *variables de estado* y la *cola de eventos* pendientes de su ejecución. Cada evento posee una marca de tiempo que indica cuándo debe ser ejecutado, es decir, un parámetro de ordenación. La ejecución de un evento supone la potencial variación de las variables de estado del proceso que ejecuta el mismo. En cada momento, en cada iteración, el evento ejecutado es aquel cuya marca de tiempo es la menor. Si se seleccionase otro evento cualquiera E_{t_n} para la siguiente ejecución, en lugar del de menor marca

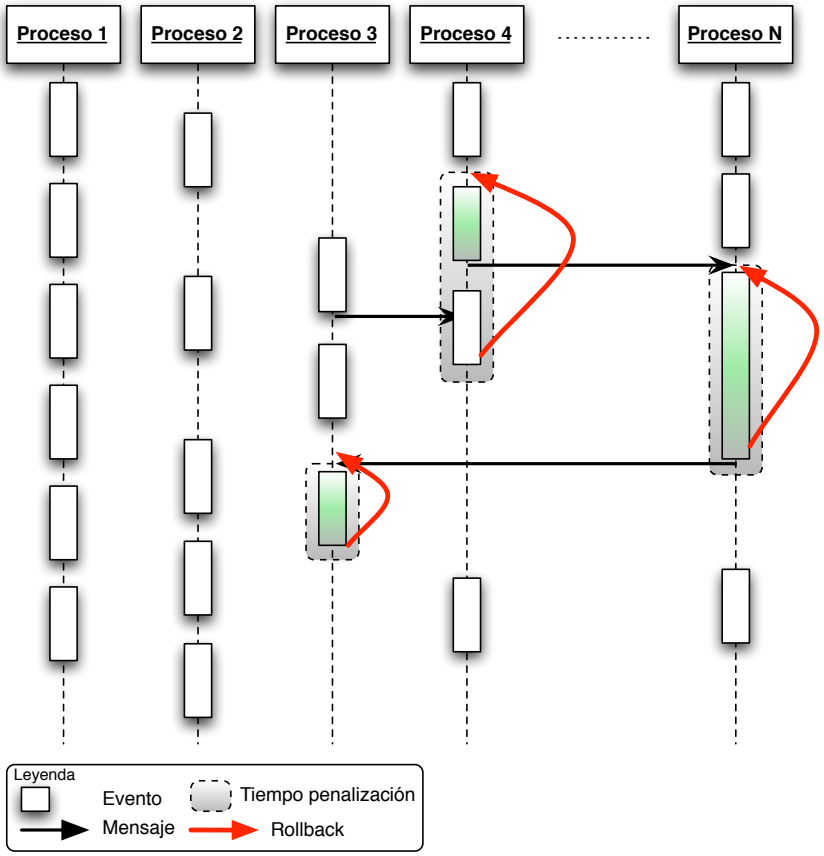


Figura 2.4 : Penalización de un algoritmo optimista.

temporal, $E_{t_{(n-m)}}$, las variables de estado tras la ejecución del evento $E_{t_{(n-m)}}$ podrían ser muy diferentes a los valores que tuvieren tras la ejecución causal. De hecho, los valores que tomen dichas variables de estado influyen en los valores que tomarán en la próxima iteración. En este problema no se puede aplicar la propiedad de conmutatividad, por lo que el resultado final de la simulación sería no válido y no acorde con los datos iniciales. Se estarían produciendo **errores de causalidad** ya que ejecuciones *del futuro* estarían afectando a resultados *en el pasado*.

Obviamente, no es lo habitual que en un sistema con una única cola de eventos, éstos sean ejecutados de manera arbitraria. Pero este problema aparece de inmediato cuando son varios los procesos con una cola de eventos a ejecutar y es necesario determinar cuál de ellos es el más adecuado para ser ejecutado en la siguiente iteración. Aún se puede añadir un grado más de complejidad si no sólo tenemos varios procesos a ejecutar en un sistema, sino que además tenemos varios sistemas con las mismas características que, además, realizan paso de mensajes entre ellos. Es decir, interaccionan entre sí. En este caso, es habitual recibir un evento de otro proceso con una marca temporal muy anterior al evento que se está ejecutando en la actualidad.

Para poder asegurar que el principio de causalidad se ha respetado en los resultados finales de la simulación, es necesario poner los medios necesarios desde el primer momento. En este caso, las colas de eventos deben estar ordenadas por sus marcas de tiempo para así evitar el problema de la causalidad dentro de un mismo proceso. De la misma forma, cuando a un proceso llega un mensaje de otro proceso,

éste debe ser insertado en la cola ordenadamente. La ejecución posterior de los eventos estará regida por un programador de eventos o **scheduler**. Dicho algoritmo es el responsable de decidir, de entre todos los procesos que se ejecutan en un sistema, cuál es el evento a ejecutar que llevará al sistema al cumplimiento del principio de causalidad en esa iteración. Cuando, además, tenemos múltiples sistemas, la situación se complica de manera sustancial.

Según la literatura [82],[75], existen dos mécanismos para la definición del algoritmo de selección del siguiente evento: los *mecanismos conservadores* y los *mecanismos optimistas*. Ambos, como veremos a continuación, poseen sus ventajas y sus incovenientes, las cuales nos servirán de ayuda a la hora de decidir qué tipo de implementación vamos a elegir para el problema que nos ocupa.

2.4 Simulación optimista frente a simulación conservadora

El principio que rige una simulación **conservadora** es el de conservar inviolable [82][81] el principio de causalidad. Por lo tanto, la principal tarea de un algoritmo de carácter conservador es el de la selección *segura* del siguiente evento a ejecutar. Por selección *segura*, del siguiente evento a ejecutar, se entiende como la elección de aquel evento que evita incumplimientos del principio de causalidad en cada iteración.

Los primeros algoritmos conservadores aparecieron al final de la década de los años 70. Éstos aplicaban una serie de restricciones al problema que garantizaban la causalidad: las marcas temporales no podían decrecer y el canal aseguraba la entrega FIFO (*first-in-first-out*). De esta manera, el siguiente evento a ejecutar era aquel que tenía la marca temporal menor de entre todos los procesos. Con estas condiciones se asegura la causalidad completa del sistema, pero favorece la posible aparición bloqueos mútuos o *deadlocks*. Esta situación es común cuando se tiene pocos eventos pendientes de ser ejecutados y muchas más colas de eventos que eventos. Por ejemplo, cuando se tienen varias colas de eventos de varios procesos se tenderá a ejecutar aquella cuyo reloj sea el menor. Si se tienen varios ciclos en varios procesos en los que no existen eventos a ejecutar, se puede dar el caso que otros eventos estén a la espera de recibir un evento de otro proceso y ambos queden bloqueados.

Para evitar los posibles *deadlocks* en la simulación se pueden enviar mensajes *vacíos* o *null messages* [82]. Estos mensajes poseen una marca temporal que indica que el proceso originario de dicho mensaje no enviará en el futuro mensajes con marca temporal menor que la del mensaje vacío. El objetivo final es notificar al resto de procesos que es seguro ejecutar el siguiente evento en la cola. El inconveniente que presenta es la difusión de estos mensajes vacíos. Si se tiene una simulación con un gran número de procesos, la cantidad de mensajes vacíos intercambiados se dispara consumiendo recursos tanto de memoria como de procesado y transmisión.

Una forma de evitar la cantidad de mensajes nulos consiste en aumentar el paso temporal de ejecución. Es decir, en lugar de incrementar el tiempo de unidad en unidad, se incrementa al siguiente valor en el que existe un evento a ejecutar. En [83] se plantea un algoritmo que detecta las situaciones de *deadlock* y las resuelve teniendo en cuenta que los eventos con la menor marca temporal son siempre seguros de ejecutar. La ventaja que presenta dicho algoritmo es que posibilita la recuperación del sistema ante un bloqueo permitiendo la continuidad de la simulación. Del mismo modo, es capaz de identificar falsos bloqueos evitando el trabajo de procesado que implica deshacer un bloqueo.

Como caso opuesto a la rigidez de los métodos conservadores, se presentan los métodos de simulación

optimista. En este caso, el principio de causalidad **no** se comprueba explícitamente, y paso a paso, a lo largo de la simulación, pero sí que debe cumplirse al final de la misma para garantizar que los resultados obtenidos son los correctos.

Estrictamente hablando, los métodos optimistas no previenen los errores de causalidad, sino que se recuperan de ellos cuando se producen. Dichos métodos no determinan cuándo es seguro o no ejecutar el siguiente evento en la cola, lo ejecutan y mantienen su proceso de ejecución hasta que la simulación termina o llega un evento que produce el error. En este caso, el método se debe recuperar el sistema del fallo producido. Los métodos optimistas explotan al máximo el paralelismo en situaciones en las que los errores de causalidad *podrían ocurrir* [82]. El método optimista más conocido es el de *Time Warp* [84], donde se expone el algoritmo de *Tiempo Virtual* que es sinónimo de tiempo de simulación (y no tiempo real). La característica fundamental del algoritmo es que la recuperación del error se realiza mediante lo que se denomina *rollback* o marcha atrás. Se deshacen las ejecuciones realizadas hasta el tiempo del evento recibido y se recupera el estado a ese instante. Este algoritmo se expondrá en la sección 3.3 más en detalle.

Existen múltiples variantes del algoritmo de *Tiempo Virtual* en función de cómo se realiza el proceso de marcha atrás y su minimización. Se han propuesto aproximaciones *perezosas* [85] en las que la marcha atrás sólo se realiza una vez ejecutado el evento fuera de tiempo y comparado las variables de estado en el instante posterior de simulación al evento anticausal. Si los estados difieren, es necesario repetir la ejecución de todos los eventos posteriores, por el contrario, si son idénticos, se puede continuar con la ejecución desde el momento en el que se recibió el evento ya que no afecta al resultado final de la simulación. Otro método consiste en la utilización de *ventanas temporales* [86]. Éste establece una ventana temporal de ejecución y sólo los eventos cuya marca está dentro del intervalo son candidatos a ser ejecutables en la próxima iteración. Este algoritmo evita que, en el caso de error de causalidad, cálculos erróneos se propaguen mucho en el tiempo de simulación. En la literatura se plantean otros muchos métodos diferentes que no son más que pequeñas variaciones de estos últimos [87][88][89][90][91].

Los métodos optimistas poseen dos ventajas respecto a los métodos conservadores. La primera es que explotan en mayor grado el paralelismo en las simulaciones, como se dijo anteriormente. Y segundo, los métodos conservativos dependen en mayor medida de la información específica del problema a simular para determinar cuáles son los eventos más adecuados para ser ejecutados en la siguiente iteración. Esto deriva en que el desarrollo de código en las aplicaciones donde se utiliza una simulación optimista sea más sencillo.

Un concepto fundamental en la simulación optimista es que el tiempo es relativo o *virtual*. El tiempo utilizado en la simulación no es el tiempo real como estrictamente lo conocemos, sino que es el tiempo de simulación. En este método, el tiempo avanza irregularmente en función del tiempo del siguiente evento a ejecutar. Por ejemplo, el reloj local del proceso tiene el valor 569 milisegundos porque el siguiente evento a ejecutar posee esa marca de tiempo. En los métodos conservadores, el siguiente tiempo de simulación sería 570 y seguiría aumentando de unidad en unidad hasta que se alcance el tiempo del siguiente evento a ejecutar en 574. Por el contrario, con el método optimista, después del tiempo 569 el reloj local indicaría 574.

A pesar de que, aparentemente, los métodos optimistas presentan buenas características para la computación distribuida, presentan también algunos incovenientes. El primer problema identificado aparece cuando el algoritmo de simulación realiza operaciones de Entrada/Salida (I/O), es decir, por ejemplo, realiza la

escritura de un histórico en un archivo. Por su naturaleza, éstas no encajan en el concepto de marcha atrás, por lo que no puede deshacerse su ejecución. El consumo de memoria se ve incrementado en cada iteración por el hecho de tener que almacenar los estados anteriores de ejecución ante una potencial marcha atrás. Este problema puede solventarse parcialmente con la inclusión de un reloj global que controle la simulación. El valor de este reloj indica que eventos con marca temporal por debajo de su valor nunca ocurrirán, por lo que esos estados pueden ser despreciados y, por lo tanto, la memoria liberada. Pero el problema que más daño puede causar en una simulación optimista es una cascada de *rollbacks*. Esto puede producirse cuando alguno de los procesos avance mucho más rápido que el resto en tiempo de ejecución afectando a otros más lentos con los eventos que les envía. Aparte de necesitar más memoria para almacenar los estados anteriores, si al proceso rápido le llegase un evento fuera de tiempo (*straggler*) tras haber enviado gran número de eventos a otros procesos, provocaría una cascada de *rollbacks* en el resto llegando, en el peor de los casos, al punto de partida inicial de la simulación. Una forma de limitación de este problema consiste en tener un reloj global de la simulación [84]. El tiempo límite inferior al que se podría llegar con una cascada de *rollbacks* sería el valor de dicho reloj global que siempre será menor o igual a los relojes locales de cada proceso.

Un problema de simulación que se caracteriza por su alto grado de paralelismo es el de las Redes de Sensores Inalámbricas (WSN). En la actualidad se están desarrollando numerosos algoritmos de enrutamiento de carácter jerárquico en el que se intenta minimizar el número de transmisiones mediante optimización y diseño en varias capas [3][92]. En el diseño se tiene en cuenta el funcionamiento de las capas MAC y física (PHY) para optimizar el funcionamiento de la capa de red (NETWORK) [36][66]. Al tratarse cada sensor de una entidad autónoma y al tener un gran número de sensores en una red, conceptualmente, la forma más apropiada de realizar la simulación es mediante un algoritmo optimista que explote el grado de paralelismo intrínseco del problema. Además, la paralelización va a permitir la simulación de redes de sensores muy densa que, con una única máquina, convendría una gran cantidad de tiempo de computación. Por lo tanto, el método elegido para la implementación del simulador de redes de sensores es un algoritmo optimista, en concreto el algoritmo de *Tiempo Virtual* [84].

CAPÍTULO 3

DESCRIPCIÓN DE LA ARQUITECTURA DE PROCESAMIENTO DISTRIBUIDO

3.1 Propuesta híbrida de arquitectura distribuida

En el capítulo anterior se vio los problemas fundamentales de la computación distribuida y las soluciones que se han abordado a lo largo de la historia reciente en el campo de la computación. Se estableció la distinción entre algoritmos conservadores y algoritmos optimistas describiendo las ventajas e inconvenientes de ambos.

En este capítulo se plantea una arquitectura que hibride ambos tipos de algoritmos en un único sistema. Este sistema híbrido utilizará una aproximación optimista cuando se trata de comunicaciones inter-procesadores y una aproximación conservadora cuando son comunicaciones intra-procesador.

La justificación para ello es que dentro de un mismo procesador sólo se puede ejecutar un proceso en cada instante de tiempo, por lo que debe ser seleccionado cuál es el siguiente a ejecutarse en función de una serie de reglas que respeten el principio de causalidad. Es en esta situación en la que se emplean algoritmos conservadores de ordenación. Los procesadores ejecutan sus procesos en paralelo en función de la cola de eventos actual de cada uno de manera que avanzan de manera optimista. Sólo cuando un mensaje procedente de un procesador llega a otro procesador se pueden producir violaciones de la causalidad, en cuyo caso sería necesario la ejecución de una marcha atrás.

El sistema dispondrá de un programador (*scheduler*) por cada procesador que será el que definirá cómo se ordenan y ejecutan los eventos de cada uno de los procesos contenidos en el mismo. En función de la implementación del programador, se podrán tener algoritmos con diferentes prestaciones. En la literatura abundan nuevos planteamientos de este programador que, en alguna medida, mejoran las prestaciones.

En las próximas secciones se exponen los algoritmos fundamentales que explican ambas formas de afrontar un problema distribuido. Por un lado, la aproximación conservadora de Lamport [80] y por otro la definición de Tiempo Virtual de Jefferson [84].

3.2 Ordenación de eventos: propuesta de Lamport

En [80] se define la relación *ocurre antes* para la ordenación parcial de eventos en un sistema distribuido.

Habitualmente se considera que el evento a ocurrió antes que el evento b si ocurrió en un tiempo previo:

$$a \texttt{ ocurrió antes que } b \Leftrightarrow T_a < T_b \tag{3.1}$$

Esta definición es válida si se considera en términos de las teorías físicas del tiempo. Sin embargo, si se desea ajustarse correctamente a la especificación, ésta debe ser formulada en término de los eventos observables en el sistema. Por lo tanto, si el sistema se formula en términos de tiempo físico, éste debe contener relojes reales que, por otra parte, presentan el difícil problema de la sincronización. Por ello es necesario definir la relación *ocurrió antes* en otros términos diferentes.

Se define el sistema como un conjunto de procesos cada uno de ellos con una secuencia de eventos a ejecutar. En este caso, se define la relación *ocurrió antes* ($\rightarrow$) como lo siguiente [80]:

Definición 3.1. *La relación $\rightarrow$ sobre el conjunto de eventos de un sistema es la menor relación que satisface las siguientes condiciones:*

1. *Si a y b son eventos del mismo proceso y a ocurre antes que b, entonces $a \rightarrow b$.*
2. *Si a es el evento que provoca el envío de un mensaje por un proceso y b es el evento resultado de la recepción de dicho mensaje en el proceso destino, entonces $a \rightarrow b$*
3. *Si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$.*
4. *Dos eventos diferentes a y b se dicen concurrentes si $a \nrightarrow b$ y $b \nrightarrow a$.*

Una definición alternativa, menos formal, de la relación sería la siguiente: $a \rightarrow b$ si existe alguna posibilidad de que el evento a pueda afectar de manera causal al evento b.

Para establecer una ordenación de los eventos a través de la relación definida, es necesario introducir el concepto de *reloj lógico*. En los términos requeridos no son más que una manera de asignar un valor numérico a un evento que representará el instante de ocurrencia del mismo. La *condición de reloj* establece [80] que para cualquier eventos a y b:

$$a \rightarrow b \Rightarrow C(a) < C(b) \tag{3.2}$$

siendo $C(e)$ el instante de ocurrencia del evento e. Con ambas condiciones se establece un método de ordenación de los eventos del sistema.

El criterio de ordenación de los eventos es el tiempo de ocurrencia de los mismos. Para la ordenación de los mismos se define la siguiente relación:

Definición 3.2. *Se puede decir que $a \Rightarrow b$ si y sólo si se cumple alguna de las siguientes condiciones:*

1. $C_i(a) < C_j(b)$
2. $C_i(a) = C_j(b)$ *y* $P_i \prec P_j$

donde $C_i(a)$ es el valor del reloj del evento a para el proceso i; $C_j(b)$ el valor de reloj del evento b para el proceso j; y P_i y P_j son los procesos i y j. La relación $\prec$ indica la precedencia del operando de la izquierda sobre el de la derecha. Esta relación depende del sistema de relojes y no es único. Es decir, existen múltiples ordenaciones parciales según la elección del reloj que satifacen la definición 3.1.

En un sistema distribuido, es necesario un proceso específico que realize la programación de los eventos (*scheduler*). Éste es el que debe decidir qué evento debe ser ejecutado, es decir, es el proceso que debe

aplicar las condiciones de ordenamiento anteriores. La simple ejecución del algoritmo de ordenación no funcionará a menos que se realicen algunas asunciones. Para probar este extremo, imaginemos tres procesos. P_0 es el algoritmo de *scheduling*, P_1 envía una petición a P_0 y posteriormente le envía un mensaje a P_2. Después de recibir dicho mensaje, P_2 le envía una petición a P_0. Es posible que esta petición llegue primero a P_0 que la enviada por P_1 lo cual violaría las condiciones antes expuestas. Para solucionar este problema, el sistema debe cumplir dos reglas adicionales:

R1 - Cada proceso P_i incrementa C_i entre dos eventos consecutivos.

R2 - (a) Si un evento a es el resultado del envío de un mensaje m por el proceso P_i, entonces m tiene una marca temporal igual a $T_m = C_i(a)$. (b) Después de recibir el mensaje m, el proceso P_j ajusta C_j a un valor mayor o igual a su valor actual **simpre mayor que** T_m.

Estas reglas permiten ordenar los eventos totalmente de manera única [80]. Adicionalmente, son necesarias otra serie de asunciones:

- Para cualquiera dos procesos P_i, P_j los mensajes enviado de P_i a P_j se reciben en el mismo orden en que se envían.
- Todo mensaje enviado es siempre recibido.
- Cualquier proceso puede enviar directamente mensajes a otro proceso.

A pesar de tomarlas como asunciones, esto puede ser implementado simplemente con la introducción de números de mensaje y de un protocolo de asentimiento ante la recepción de un mensaje (ACK). Inicialmente, las colas de cada uno de los procesos contienen el mensaje $[T_0 : P_0]_{req}$ para hacer la petición de un recurso compartido, siendo P_0 el proceso inicial que se ejecuta y T_0 menor que el tiempo del reloj de cualquiera de los otros procesos.

El algoritmo de *scheduling* de procesos se define con los siguientes pasos:

1. Para solicitar el recurso, P_i envía el mensaje $[T_m : P_i]_{req}$ a cada uno de los otros procesos encolándose en las correspondientes colas. T_m es la marca de tiempo del mensaje.
2. Cuando el proceso P_j recibe el anterior mensaje lo ubica en su cola y envia un asentimiento (ACK) con marca temporal a P_i.
3. Para liberar el recurso, P_i elimina $[T_m : P_i]_{rel}$ de su cola y envía el correspondiente mensaje de liberación del recurso a todos los otros procesos.
4. Cuando P_j recibe el mensaje $[T_m : P_i]_{rel}$, elimina el correspondiente a la reserva del recurso de su cola.
5. A P_i se le asigna el recurso cuando se satisface: (i) Existe un $[T_m : P_i]_{req}$ en la cola que está ordenado con la relación $\Rightarrow$. (ii) P_i ha recibido un mensaje de todos y cada uno de los otros procesos con una marca temporal posterior a T_m. Estas condiciones son comprobadas localmente en cada proceso.

En este algoritmo, no existe ningún elemento central de sincronía, sino que cada proceso transcurre de manera independiente. A pesar de esta independencia, cada proceso necesita conocer el estado de los

demás para poder ejecutarse con seguridad, es decir, queda a la espera de los otros en el caso de que no sea posible su ejecución.

En este caso, la sincronización se especifica en términos de un máquina de estados consistente en un conjunto $\mathbb{C}$ de comandos, un conjunto $\mathbb{S}$ de stados y una función:

$$\mathbf{e} : \mathbb{C} \times \mathbb{S} \rightarrow \mathbb{S} \tag{3.3}$$

donde $\mathbb{C}$ son los comandos de petición y liberación de recurso, $\mathbb{S}$ la cola de comandos de espera para la petición del recurso.

Una vez explicado el algoritmo, se puede ver que presenta varios problemas. El primero es que cada proceso debe informar al resto de sus intenciones. Si en el sistema hay un número elevado de procesos esto supone una carga considerable de mensajes a transmitir por la red. El segundo, como consecuencia de los anterior, el fallo de un proceso puede provocar que el resto quede en espera conduciendo al sistema a un estado de bloqueo general. Por último, el hecho de que un proceso espera a otros restringe las capacidades de paralelismo del sistema distribuido reduciendo la posible ganancia de computación.

3.3 Definición de *Tiempo Virtual*: propuesta de Jefferson

El algoritmo de *Time Warp* [84] se definió a mediados de la década de los 80 por David R. Jefferson. Puede considerarse el primer algoritmo optimista de la Historia de la computación distribuida. En el mismo se plantea una simulación distribuida en la que cada proceso realiza sus acciones sin tener en cuenta el tiempo de los demás procesos. Si en un momento determinado, a ese proceso, le llega un evento con una marca de tiempo anterior a la del actual evento en ejecución, el proceso entra en un error de causalidad que debe subsanarse mediante un proceso de marcha atrás o *rollback*. En esta sección describiremos teóricamente el funcionamiento de dicho algoritmo de manera bastante somera. Un mayor detalle se tendrá en cuenta en el capítulo 3 donde se describirá la implementación en Java del algoritmo.

Es condición necesaria y suficiente que los mensajes sean procesados en orden temporal para que el algoritmo funcione correctamente. Tampoco es deseable [84] que el tiempo virtual de cada proceso progrese con la misma tasa que la del tiempo real, ya que esto, *secuencializaría* la ejecución del sistema y se obtendría ningún beneficio de la paralelización. Dado el funcionamiento de los sistemas distribuidos, no importa cuál es el siguiente evento a ejecutar ya que siempre puede llegar un evento, procedente de otro proceso, con una marca temporal menor. Discernir y resolver este problema es lo que define el algoritmo de *Time Warp*.

Este algoritmo se compone de dos parte fundamentales: un mecanismo de control local que se asegura que los eventos son recibidos y ejecutados en el orden correcto y un mecanismo de control global responsable de tareas más generales como el control de flujo, la detección de fin de ejecución o el sistema I/O.

El **mecanismo de control local** se basa en que cada proceso tiene su propio *reloj virtual local* que avanza a cada evento terminado de ejecutar y nunca mientras un evento se está ejecutando. Por este motivo, la gran mayoría de los procesos estarán desincronizados cada uno con su propio valor. Cada vez que se envía un evento a otro proceso, su reloj local que incluye en el evento como el tiempo de envío. El tiempo de recepción es el del reloj local del proceso que recibe dicho evento. Los eventos recibidos se insertan en la cola local de recepción del proceso ordenado por orden de tiempo de recepción.

Idealmente, un proceso ejecuta sus eventos en orden creciente de tiempo virtual de recepción. Esta ejecución se produce mientras nos se reciba un evento un tiempo virtual de recepción *pasado*, hecho que se puede producir si los procesos tienen diferentes tasas de incremento del tiempo virtual local. Esto rompe la premisa de causalidad y tiene que ser subsanado. La manera de hacerlo es, que el proceso receptor, debe dar marcha atrás en sus ejecuciones hacia un estado de tiempo virtual anterior al del evento recibido, restaurando todas las variables de estado al valor en ese instante temporal previo, y continuando la ejecución desde ese punto teniendo en cuenta el nuevo evento. Cuando el proceso termina de ejecutar todos los eventos encolados su reloj virtual toma el valor de $+\infty$ indicando que esa simulación ha finalizado. A pesar de haber *terminado* [84], el proceso no se cierra ya que pueden llegar eventos al mismo fuera de tiempo, lo cual nos devolvería el proceso a un estado de *no terminado*.

Una vez que un proceso se ve obligado a dar marcha atrás, necesita de la ejecución de una serie de acciones, no sólo locales, para restaurar su estado al estado anterior al mensaje recibido. El proceso de *rollback* requiere de los siguientes elementos para su consecución:

- *coordenada del proceso*. Son la coordenada espacial (identificación del proceso) y la coordenada temporal (reloj local virtual del proceso).
- *estado del proceso*. El conjunto de variables que definen el estado en un instante tempora, virtual, de un proceso.
- *cola de estados*. La cola donde se almacenan, ordenadas temporalmente, los estados del proceso. Si es necesario un *rollback*, podemos restaurar el estado del proceso en el instante deseado extrayendo las variables de la cola. El número de estados almacenado dependerá del *reloj virtual global* que se analizará próximamente.
- *cola de entrada*. La cola en la que se almacenan los eventos, ordenados por tiempo virtual de recepción, a ejecutar y recibidos de otros procesos. A medida que se ejecutan, éstos, no son borrados por si es necesario realizar una marcha atrás.
- *cola de salida*. En ella se almacenan los mensajes enviados a otros procesos, ordenados por tiempo virtual de envío. En realidad no son exáctamente los mensajes enviados sino los *anti-mensajes* o *anti-eventos*, que son exáctamente igual que los eventos pero que indican que es necesario eliminar el correspondiente evento de la cola. En el caso de realizarse un *rollback*, estos mensajes son reenviados a los procesos que recibieron previamente el evento. El número de anti-eventos almacenados dependerá, una vez más, del *reloj virtual global*.

El mecanismo de actuación de un anti-evento es el siguiente: si un proceso recibe un anti-evento comprueba si en la cola de entrada existe su correspondiente evento, en cuyo caso lo elimina de la cola. De esta forma se evita que dicho evento sea ejecutado evitando la producción nuevos *rollback* hacia otros procesos. En el caso de que el evento no se encuentre en la cola porque el reloj virtual del proceso es superior al del tiempo del anti-evento, esto es, ha sido ejecutado, el proceso debe iniciar también su mecanismo de *rollback* hasta el instante virtual previo a la ejecución del evento, eliminarlo de la cola y continuar la ejecución sin dicho evento. Por último, en el caso extremo de que un proceso reciba antes el anti-evento que el evento, si éste no se ha ejecutado, se eliminaría de la cola, y si llega el momento de ejecutar el anti-evento, la acción a tomar debería ser ninguna y almacenar dicho suceso para cuando llegue el evento eliminarlo directamente.

En este proceso, como es fácil inferir, no existe la posibilidad de *deadlock* o bloqueo, ya que en el peor de los casos se tendrá un efecto dominó que llevará al sistema al instante inicial de la simulación.

En la práctica, datos experimentales [84] sugieren que el número de procesos de marcha atrás que suceden en un sistema no es elevado, por lo que computacionalmente no resulta muy costoso dicha aproximación. Se asume que la mayoría de los programas obedecen el *principio de localidad temporal* que dice que la mayoría de los mensajes llegan en un tiempo virtual futuro a su destino mientras que los que llegan en el pasado lo hacen en un pasado *reciente* evitando consumir grandes recursos de memoria para almacenar las variables de estado y los correspondientes anti-eventos.

El **mecanismo de control global** resuelve algunos problemas que presenta el algoritmo de *Time Warp*. Estos problemas son el control del progreso global de la ejecución, la detección de la finalización de la ejecución, errores I/O cuando se realiza un *rollback* o la insuficiencia de memoria para almacenar las variables de estado.

El concepto central [84] para el control versa en torno al *reloj virtual global (GVT)*. Este valor representa la instantánea en un instante determinado de la ejecución del sistema. Por definición, GVT, es en un instante determinado el mínimo entre todos los tiempos virtuales locales de cada proceso en el sistema, y el tiempo virtual de envío de aquellos mensajes que han sido enviados pero no procesados.

$$gvt(t) = \min_i\{lvt_i(t), svt_{ij}(t)\} \tag{3.4}$$

De la ecuación 3.4 se extráen varias conclusiones. GVT nunca decrece a pesar de que, independientemente, los relojes locales pueden dar marcha atrás a causa de posibles *rollbacks*. GVT es el valor ínfimo que cualquier reloj virtual local puede tomar, es decir, el valor temporal máximo al cual un proceso puede retroceder. Todo esto implica que GVT es el *reloj virtual del sistema completo* y mide el progreso de la ejecución del sistema en su totalidad.

Al valor de GVT se denomina como *horizonte móvil* por el cual ningún proceso puede dar marcha atrás más allá de dicho valor, y es segura su ejecución. Como se ha visto, la definición está proporcionada en función de capturas del estado del sistema en un instante de tiempo determinado, lo cual no es operativo. Una definición más operativa, e implementable, es la siguiente:

$$gvt \leq \min_i\{lvt_i(t), svt_{ij}^{nack}(t), svt_{ij}^{nproc}\} \tag{3.5}$$

donde $lvt_i(t)$ es el valor del reloj virtual local del proceso i; svt_{ij}^{nack} el tiempo virtual de envío de un mensaje del proceso i al proceso j que todavía no ha sido confirmado (ACKED); svt_{ij}^{nproc} el tiempo virtual de envío de un mensaje del proceso i al proceso j encolado pero no procesado. Esta forma de calcular el valor de GVT nos da un valor desfasado con respecto al valor real, ya que mientras se calcula, la simulación continua ejecutándose y el valor real puede cambiar. Este efecto es una consecuencia de que el sistema no esté sincronizado.

El cálculo del valor de GVT debe realizarse de manera asidua, ya que permite el ahorro de memoria al tener que almacenar menos variables de estado. Como contrapartida, el hecho de calcular el valor, requiere tiempo de procesador y transmisión de mensajes a través de la red, por lo que ralentiza las comunicaciones del resto de procesos. Debe existir un compromiso entre ambos.

3.4 Herramientas para la programación de sistemas distribuidos

Para la implementación de la arquitectura que se está exponiendo es necesario realizarla con herramientas que faciliten el procesamiento distribuido. Para ello, se hace un estudio previo de las herramientas disponibles y se evalúa cuál es la que mejor se adapta a nuestro problema.

3.4.1 Tipos de herramientas

Existe muchas herramientas que habilitan la programación de sistemas distribuidos. Unas son librerías ya programadas pero que dejan poca flexibilidad a la hora de afrontar determinados problemas y otras son middlewares basados en algún(os) lenguajes de programación o incluso el propio lenguaje lo lleva implícito en su construcción.

A continuación se enumeran aquellas más conocidas y utilizadas explicando sus características principales:

- **Java Remote Method Invocation (RMI).** Es un mecanismo implícito del lenguaje de programación Java que habilita la comunicación de servidores en aplicaciones distribuidas. Es el equivalente orientado a objectos a las llamadas a procedimiento remotas (RPC). RMI permite el paso de objetos por referencia y presenta un recolector de basura distribuido. El modelo que propone es de cuatro capas partiendo desde la capa de aplicación donde se implementan los servidores, pasando por las capas de *stub-skeleton* y de referencia remota, hasta llegar a la capa de transporte que realiza las conexiones oportunas. El problema principal que presenta está en la interoperabilidad entre servidores ya que todos deben estar basados en Java.
- **Common Object Request Broker Architecture (CORBA).** Es un estándar que define una plataforma para el desarrollo de aplicaciones distribuidas bajo la filosofía de orientación a objetos remota. CORBA define las librerías, protocolo de comunicaciones y la infraestructura necesaria para permitir la interoperabilidad entre servidores escritos en diferentes lenguajes de programación. Se define un lenguaje de definición de interfaces remotos IDL para especificar los servicios disponibles en los diferentes servidores. Define, también, mecanismos de seguridad y transaccionales para asegurar la integridad de las comunicaciones. De la experiencia se extrae que operar con CORBA es complicado ya que existen innumerables implementaciones del estándar cuya forma de operar difiere unas de otras.
- **Distributed Component Object Model (DCOM).** Es una tecnología propietaria de Microsoft para el desarrollo de aplicaciones distribuidas. Éste extiende el modelo COM y ha sido abandonada en favor de .NET. No es multiplataforma sino que únicamente es ejecutable sobre sistemas MS Windows.
- **Simple Object Access Protocol (SOAP).** Se trata de un protocolo estándar que define el intercambio de información entre procesos utilizando el estándar de datos XML. Es el protoco más utilizado para la implementación de servicios Web. El uso de XML hace que el protocolo se pesado y cargue en exceso los servidores.
- **.NET.** Es una plataforma propietaria de Microsoft que se ejecuta sobre máquinas con el sistema operativo MS Windows. Presenta una librería de recursos muy extensa y está basada en una máquina virtual para la ejecución del código. Soporta múltiples lenguajes de programación C#, C/C++, Java.

La anterior enumeración nos muestra, a grandes rasgos, las características fundamentales de las principales herramientas programáticas para computación distribuida. Dichas herramientas presentan inconvenientes que las excluyen para su utilización como base de la implementación que se desea desarrollar. El objetivo buscado es una herramienta multiplataforma y multilenguaje, y cuyo desarrollo con ella sea simple, es decir, con una curva de aprendizaje pequeña. El *middleware* ZeroC ICE presenta todas esas características y, por lo tanto, es el seleccionado como herramienta base de la implementación. Gracias a la utilización de ZeroC ICE como herramienta, en un futuro, se podrá explotar la potencia de la supercomputación híbrida [93], basada en tarjetas gráficas, para la realización de simulaciones. La integración en el sistema se realizará de una manera transparente mediante la implementación del código específico para la arquitectura de la GPU pero manteniéndose los interfaces, y por lo tanto, el corazón de la arquitectura distribuída. ZeroC ICE presenta otras utilidades como el IceGrid [94] que permite la gestión de los nodos de computación para distribuir la simulación en diversas máquinas. A continuación se hace una introducción del *middleware* profundizando en sus características y propiedades.

3.4.2 ZeroC ICE

Internet Communications Engine (ICE) es la plataforma (middleware) creada por la empresa ZeroC. Se trata de una plataforma de programación orientada a objetos basada en llamadas a procedimientos remotos, *grid computing* y con funcionalidad de publicación/subscripción.

Es multiplataforma y multilenguaje, soportando C++, Java, lenguajes .NET, Objective-C, Python, PHP y Ruby. Una de las mayores ventajas aportadas por la plataforma es ser transparente para los cortafuegos (firewalls) de manera que se puede crear aplicaciones distribuidas a lo largo de toda la Internet.

Su implementación y diseño está basado en CORBA, de hecho varios desarrolladores de CORBA colaboran en el desarrollo de ICE, pero es mucho más simple que CORBA. Dicha simplicidad favorece la robustez y la estabilidad de los sistemas implementados.

Su arquitectura se basa en aplicaciones cliente/servidor que publican una serie de objetos cuyos métodos pueden ser invocados de manera remota. Realmente, los clientes/servidores no son específicamente eso, sino roles tomados por determinadas partes de la aplicación, es decir, realizan además otras tareas [94]. Los objetos dentro de la plataforma se denominan objetos ICE que se caracterizan por lo siguiente:

- es una entidad local o remota que responde a peticiones de un cliente.
- puede ser instanciado en un servidor o en múltiples pero seguir siendo un único objeto.
- un objeto ICE presenta uno o varios interfaces de operación con diferentes parámetros de entrada o de salida.
- un objeto ICE presenta un interfaz principal que lo distingue, pero puede presentar otros alternativos que se conocen como *facetas* (*facets*).
- cada objeto ICE presenta un identificador único que lo representa en todo el sistema.

Un objeto ICE puede invocar métodos de otro objeto ICE a través de su **proxy**. El proxy se obtiene a través de las herramientas que proporciona el middleware y se opera con él como si se operase directamente con el objeto remoto, restringido eso sí, a las operaciones publicadas por el interfaz.

La publicación de las interfaces se realiza a través del Lenguaje de Especificación de ICE (SLICE). Es el equivalente al lenguaje IDL en CORBA, pero la sintaxis es diferente. Se definen varios tipos posibles que se pueden usar en la definición. Los más importantes o más usados son:

- `struct` define estructuras de datos al estilo del lenguaje de programación C. No define métodos o funciones.
- `class` define estructuras de datos y métodos. Estos métodos se implementan y sólo se pueden ser invocados desde la máquina local.
- `interface` define una serie de métodos que son invocados desde la máquina local o desde otra máquina remota.

El protocolo que define ICE está basado en un protocolo RPC que usa o TCP/IP o UDP/IP como capas de transporte y de red. Opcionalmente, se habilita la posibilidad de utilizar SSL como transporte de manera que las comunicaciones entre el cliente y el servidor estan encriptadas y son seguras.

3.5 Descripción de la arquitectura distribuida

La arquitectura implementada se basa en código Java y en la librería para comunicaciones ZeroC ICE. A pesar de que se podría implementar en cualquiera de los lenguajes disponibles se elige Java por ser multiplataforma y uno de los lenguajes más utilizados, así como por la inmensa capacidad de recursos que presenta en la red.

La filosofía que guía dicha implementación está basada en el algoritmo de *Time Warp* presentado en 3.3 y [84]. Los procesos se ejecutarán independientemente unos de otros hasta su finalización siempre y cuando no reciban un evento que viole la causalidad y se vean obligados a dar marcha atrás. La comunicación entre los procesos se realiza mediante las capacidades que ofrece ZeroC ICE.

En futuras evoluciones de la arquitectura se planteará el uso de unidades de procesamiento gráfico (GPU) para potenciar aún más la simulación. En el caso de nVIDIA CUDA, la programación se realiza en C/C++ por lo que el uso de ZeroC ICE haría transparente la implementación de los procesadores y procesos para los otros componentes del sistema. Incluso podrían coexistir en un mismo sistema implementaciones utilizando GPUs e implementaciones Java como la presentada aquí.

El sistema se compone de un elemento central donde se configura el resto de componentes, un elemento para registrar los mensajes de información y los elementos de proceso distribuidos en diferentes máquinas.

3.5.1 Procesos y procesadores

Estos elementos son las unidades principales del sistema ya que son los encargados de realizar la tareas de computación en sí misma. En la figura 2.1 se mostró cual es la relación entre ambos de manera gráfica. Un procesador es una unidad independiente de ejecución que opera en paralelo con otros procesadores ya que posee autonomía. Los procesos son unidades lógicas que ejecutan una tarea determinada utilizando los recursos de un procesador.

En la implementación realizada el procesador es también un elemento lógico pero que se vincula directamente con un procesador físico. De esta manera, es éste el que se encarga de organizar la ejecución de los diferentes procesos relacionados con el sistema y no el propio sistema operativo (SO). Los procesadores se componen de los siguientes elementos:

- La lista de procesos que debe ejecutar.
- Un programador de ejecución de los procesos.
- Funcionalidad para poner en marcha, para y pausar el procesador.
- Referencia al sistema de control central.

La implementación se basa en un hilo de ejecución independiente de la posibles llamadas remotas que puedan realizarse a un procesador. Dichas llamadas remotas se definen en un archivo SLICE que el intérprete de ICE es capaz de procesar para generar el código necesario para establecer las comunicaciones remotas. Las llamadas remotas disponibles son las siguientes:

- `start()`, `pause()` y `stop()` para el control remoto de la ejecución del procesador y, por ende, de los procesos.
- `createLogicalProcess()`, `removeLogicalProcess()`, `getLogicalProcess()` y `getLogicalProcessList()` para el control del ciclo de vida de los procesos.
- `getLVT()` para obtener el menor valor de reloj de todos los procesos. Este método es invocado por el sistema de control para estimar la evolución de la simulación global.

Estas llamadas pueden ser invocadas independientemente de que el procesador esté realizando su trabajo, por lo que dinámicamente se podrían añadir o eliminar nuevos procesos en el mismo.

En su hilo de ejecución central la tarea que lleva a cabo es muy simple. Se invoca el elemento programador para que seleccione el evento más oportuno para ser ejecutado de entre todos los eventos de las colas de eventos de todos los procesos de la lista. Una vez obtenido el evento idóneo se invoca el método `execute()` del proceso para que realice la acción que es indicada en el evento seleccionado. Este compendio de acciones se ejecuta por cada evento restante en las colas de eventos de los procesos. Cuando un procesador ha finalizado todos los procesos continúa ejecutándose para el caso de que se pueda recibir un evento no causal y haya que reejecutar los procesos.

Los procesos son las unidades de ejecución de las tareas, es decir, poseen la lógica de la simulación en sí. Se componen de los siguientes elementos:

- La cola de eventos a ejecutar.
- La cola de eventos ejecutados en el caso de que sea necesario realizar una marcha atrás.
- Funcionalidad para poder recibir eventos de otros procesos (*callback functions*).
- Referencia al sistema de control central.

En este caso, a diferencia del procesador, la implementación no está basada en hilos de ejecución independiente, ya que éste es gestionado por el procesador y debe ejecutarse única y exclusivamente cuando éste lo indica. Posee métodos disponibles para la invocación remota que se utilizan para la gestión de la cola de eventos. Estos métodos se definen en el archivo SLICE:

- `receiveEvent()` y `initEvents()` para la gestión de la cola de los eventos. Uno para insertar un evento en la cola procedente de otro proceso y el segundo para inicializar una cola con eventos generados por otro componente.
- `getLVT()` y `setLVT()` para la gestión del reloj local del proceso.
- `freeze` paraliza un evento para poder ser reubicado en otro procesador.
- `execute()` para iniciar la ejecución de la acción indicada por el evento.

A pesar de que `execute()` es un método remoto, éste es invocado por el procesador en local. Como se verá más adelante, los eventos llevan información de la acción que se debe ejecutar. De esta manera, la arquitectura se generaliza lo suficiente como para poder realizar todo tipo de tareas.

Para que dos procesos puedan comunicarse entre sí deben tener una referencia al otro e invocar el método `receiveEvent()`. Para obtener la referencia del proceso remoto se hace uso del elemento de control central que, actuando como si de un servidor de nombres se tratara, mantiene la información de todos los procesos y procesadores del sistema.

3.5.2 Sistema de control centralizado

El sistema de control es el elemento central de la arquitectura. La plataforma está diseñada como un proveedor de servicios. En el mismo se organiza la gestión de los diferentes nodos que ejecutan alguna tarea dentro del sistema de manera que actúa como un *servidor de nombres* para poder localizar a otros componentes. Cuando un cliente desea utilizar las capacidades del sistema realiza una petición al proveedor del servicio y éste le devuelve un servicio de computación. Es, entonces, cuando el cliente interactúa con el servicio únicamente para realizar las labores de computación deseadas.

La forma de operar del elemento de control es recibir los registros de los diferentes elementos de red, es decir, de las diferentes máquinas implicadas en el sistema. Se introduce pues el concepto de *máquina remota* o *host*. Una máquina remota se compone de múltiples procesadores que, habitualmente, estarán unívocamente ligados al número de CPU, en relación $1:1$, de las que se compone. Dichas máquinas ejecutan un pequeño programa servidor que permite asignar procesadores mediante las invocaciones remotas `createProcessor()`, `removeProcessor()` y `getProcessorList()` para crear, eliminar y listar los procesadores contenidos en dicha máquina remota. Para ser coherentes con el número de CPU del sistema remoto, el programa servidor identifica el número y crea tantos procesadores lógicos como CPU físicas tiene el sistema. De esta manera no se sobrecargan las CPU con más de un procesador lógico.

Cuando un servidor se arranca en una máquina remota, éste se registra contra el proveedor de servicios de manera que pueda ser accesible para la realización de computaciones. Los métodos que define el proveedor son los siguientes:

- `register()`, `unregister()` y `registeredHosts()` para registrar, des-registrar y listar las máquinas remotas disponibles.
- `requestService()` devuelve una referencia a un servicio de computación. Este servicio es el encargado de interactuar y de gestionar la computación sobre los recursos asignados por el proveedor del servicio.

Una vez que el proveedor del servicio ha creado un servicio de computación, se desvincula de las operaciones que se realizarán y sabrá que los recursos asignados no pueden ser utilizados en una nueva petición hasta que sean liberados por el correspondiente servicio.

- `getHosts()` devuelve la lista de referencias a los servidores remotos asignados para realizar la computación.
- `getProcessor()` devuelve la referencia al procesador lógico dado su identificador único para poder obtener más información a partir de dicha referencia.
- `getLogicalProcess()` para obtener la referencia a un proceso dado su identificador único y posibilitar el envío de mensajes. Esta llamada
- `start()`, `pause()` y `stop()` invocan las homólogas correspondientes a cada uno de los procesadores lógicos asignados para la computación. Estas llamadas sirven para controlar globalmente la computación.

Aparte de las funciones remotas proporcionadas, el servicio de computación implementa un hilo de ejecución continuo que se encarga de la actualización del reloj global de la simulación en función de los relojes locales de todos y cada uno de los procesos que integran el sistema. Cómo se realiza el cálculo se verá en la sección 3.5.6.

Adicionalmente se ha implementado una pequeña aplicación gráfica para la gestión de procesadores y procesos dentro de las máquinas remotas. Dicha aplicación es independiente del sistema de control y se comunica con él utilizando el *middleware* implementado con ICE para obtener la información necesaria con la que poder operar.

3.5.3 Arquitectura basada en eventos y paso de mensajes

Como se indicó en secciones anteriores, la arquitectura está basada en la ocurrencia de eventos que disparan la ejecución de acciones. La implementación se ha realizado de manera que los eventos sean genéricos y que las acciones que se ejecutan sean definidas independientemente de la arquitectura.

Un evento es una estructura (`struct`) SLICE que es transformada en una clase Java por el compilador. Dicha estructura contiene diversos campos que dan forma al evento que son:

- `transmitter` y `receiver` que son los identificadores del proceso originante del evento y del proceso receptor del mismo.
- `txTimeStamp`, `rxTimeStamp` y `exTimeStamp` son los tiempos de reloj de transmisión (indicado por el proceso que genera el evento), recepción y ejecución (indicados por el proceso receptor).

- `type` indica si se trata de un evento o un anti-evento.
- `className` y `classURL` indican la acción que se debe ejecutar.
- `data` son los datos de entrada de la acción.

Los eventos llevan información explícita de la acción que se debe ejecutar. En esta implementación basada en Java, la información de la acción a ejecutar se indica con el nombre de la clase a instanciar -`className`- y la URL -`classURL`- de la ubicación donde descargarla.

El proceso destinatario instancia la clase utilizando un cargador de clases estándar de Java (`ClassLoader`). La clase instanciada debe heredar de una clase genérica que define un método específico para ejecutar la acción. El interfaz `Action` define el método `execute()` que es el invocado por el proceso. La clase debe sobreescribir dicho método introduciendo la lógica necesaria para la ejecución de la acción. Una generalización aún mayor e independiente de la implementación sería realizar directamente una llamada a sistema para invocar un programa que bien pudiera estar escrito en Java C/C++, Python, etc.

La ejecución de la acción puede implicar que se realice un cálculo en el proceso y/o la generación de un nuevo evento que debe ser transmitido a otro proceso a través de la red. Por ello, la implementación de la acción debe ser consciente del middleware ICE establecido para la comunicación inter-proceso cuyos valores pueden ser pasados como parámetro en la invocación de la clase específica.

3.5.4 Programación de eventos: scheduling

Los procesadores integrados dentro del sistema distribuido diseñado se sirven de un programador para establecer un orden en la ejecución de los eventos correspondientes a los procesos que se ejecutan en el mismo. Este programador decide, en función de determinados algoritmos, qué evento es el siguiente en ser ejecutado. Debido a que sólo un proceso puede ejecutarse en cada instante dentro de un procesador.

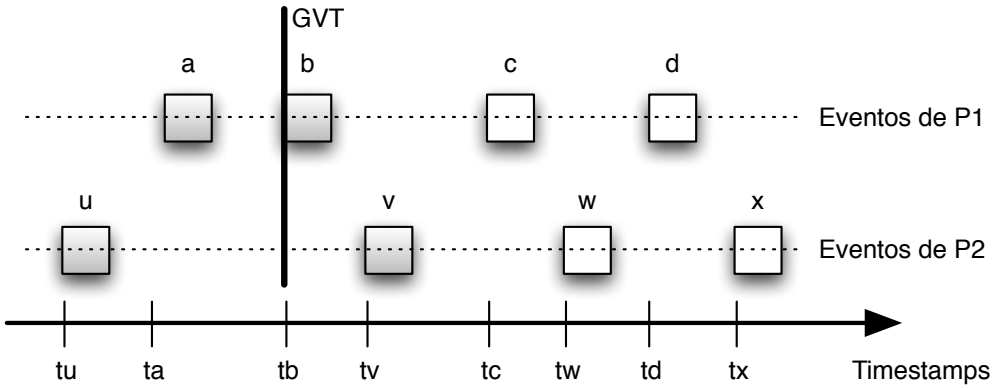


Figura 3.1 : Ordenación de eventos

La figura 3.1 muestra dos colas de procesos cada una con cuatro eventos ocurrentes en diferentes instantes de tiempo. Los eventos sombreados son aquellos que ya han sido ejecutados y los de fondo blanco aquellos que aún están por ejecutar. Podría darse el caso que algún algoritmo pudiera el evento w an-

tes que el evento c o incluso que se programase el evento c antes que el evento v. Esa determinación vendrá marcada por cada autor y por la ganancia de procesado que quiera obtener.

Como arquitectura híbrida, se explicó que dentro de un mismo procesador, la ordenación de eventos seguía un algoritmo conservador (figura 3.1). En la literatura existen numerosos algoritmos definidos para dicha ordenación basados en diferentes principios. Por ejemplo en [95] expone una ordenación de eventos basado en la menor probabilidad de ser posteriormente vuelto a atrás. En [96] se basa en el estado previo de los procesos. [90], [89] está basados en un intervalo de incertidumbre en el cual el evento puede ser ejecutado sin ser necesariamente el tiempo en el que se indica que deben ser ejectuados. [91] expone un filtrado de los procesos que deben ser dados marcha atrás, es decir, no todos los procesos que sufren una anti-causalidad son reejecutados. Incluso [84] expone un algoritmo muy sencillo en el que el siguiente evento a ejecutar es el de menor LVT.

Para que la arquitectura no se pierda en este tipo de detalles, la implementación que se propone es la de [84] seleccionando el evento con menor LVT. A efectos de implementación y planteamiento en la arquitectura, se define un interfaz genérico denominado `Scheduler` que define los siguientes métodos:

- `addProcess()` y `removeProcess()` para la gestión de los procesos sobre los que debe operar.
- `next()` para indicar, de entre todos, qué evento es el siguiente en ser ejecutado.

Este interfaz no está definido como un componente al que se pueda invocar de manera remota ya que cada procesador debe tener una instancia propia del mismo con su propia lista de procesos. La particularización para la primera versión de la arquitectura, implementa como ya hemos dicho la propuesta de Jefferson de seleccionar el evento con menor valor de LVT (`LVTScheduler`). En la figura 3.1, según este programador, el siguiente evento a ejecutar sería el evento c.

3.5.5 Proceso de *rollback* y anti-eventos

Como se vió en la sección 3.3, en los algoritmos optimistas la violación del principio de causalidad supone tener que dar marcha atrás en la simulación hasta el instante previo al evento que no se simuló y que hay que simular, lo que se denominó con el término inglés *rollback*.

En la arquitectura implementada, el sistema detetecta una violación cuando un proceso recibe un evento fuera de tiempo. Es decir, cuando se invoca remotamente por otro proceso el método `receiveEvent` el proceso que ejecuta la llamada realiza las siguientes acciones:

1. Se comprueba si se trata de un evento o de un anti-evento observando el tipo del mismo. Si se trata de un evento:
   a) se comprueba si el posible anti-evento pudiera haber sido recepcionado previamente al evento, en cuyo caso se elimina de la cola de eventos.
   b) si no existe el anti-evento, se comprueba el tiempo de recepción del evento y se compara con el actual reloj de simulación del proceso.
      i. Si es mayor se inserta en la cola de eventos ordenadamente y se sale del método,

ii. si, por el contrario, es menor se lanza el proceso de marcha atrás. Se reestablece el valor del reloj local al del evento previo al tiempo del evento recibido; se recorre la lista de eventos ejecutados -que contiene los anti-eventos correspondientes a los eventos ejecutados- y se envían a los procesos destinatarios; se reestablecen las variables de estado del proceso en el instante de reloj previamente calculado.

2. Si se trata de una anti-evento se busca en la cola de eventos el correspondiente evento y si existe se eliminan ambos de la cola y de memoria, se *neutralizan*. Si el evento todavía no se ha recibido, se inserta el anti-evento en la cola a la espera de que sea recepcionado y neutralizado. Si el evento correspondiente ya ha sido ejecutado -se encuentra en la cola de eventos ejecutados- se lanza un proceso de marcha atrás hasta el instante previo al evento, se elimina y se contínua la ejecución sin ese evento.

Como se observa, es necesario mantener en memoria todos aquellos eventos ejecutados -sus anti-eventos- para, en el caso de una marcha atrás, sean enviados a sus destinatarios. El envío de anti-eventos puede generar nuevos procesos de marcha atrás en los otros procesos provocando una reacción en cadena que, en el peor de los casos, llevaría a la simulación a su estado inicial. Esta cascada se puede cortar estableciendo un límite inferior de los relojes por el cual ya no sería necesario ni almacenar los anti-eventos y estados anteriores ni regresar la simulación a los estados iniciales.

3.5.6 Control local y global de la simulación

El control local y global de la simulación se refiere a los tiempos de ejecución de cada uno de los procesos (reloj local) y el tiempo global compendio de todos los locales.

Como se expuso en la sección 3.3 se define el reloj local virtual (LVT) como el valor de reloj local a un proceso. Este valor no transcurre de continuo sino que toma valores discretos con incrementos no uniformes marcados por el valor de recepción del último evento. En la implementación de la arquitectura, cada proceso actualiza su valor local lvt cuando se termina de ejecutar un evento, tomando el valor de recepción que indica el evento.

El valor local puede avanzar o retroceder en un proceso en función de si es necesario realizar un proceso de marcha atrás. Dado que esto puede suceder, la simulación, en su totalidad, podría tener oscilaciones en el tiempo global de la simulación. Para evitarlo, el reloj global virtual (GVT) se calcula teniendo en cuenta todos los parámetros explicados en 3.3 con lo que es imposible que, en su totalidad, GVT oscile.

Gracias a lo anterior, se puede optimizar el uso de memoria de cada uno de los procesos. Dado que GVT representa el límite inferior de la simulación y que **siempre** crece, no es necesario guardar los estados de simulación y los eventos ejecutados con LVT menor que GVT ya que nunca se producirá un proceso de marcha atrás a un instante anterior.

La implementación del cálculo de GVT es de las partes más complejas de la arquitectura y, al igual que con el programador de eventos, existen numerosas propuestas en la literatura. La más sencilla, nuevamente, es la propuesta en [84] y se realiza en el sistema de control central. La forma de plasmar el algoritmo en la arquitectura es el siguiente:

1. se consultan los valores lvt_i de todos los procesos del sistema y se almacenan en un vector.

2. cada proceso, cuando envía un evento a otro mediante la invocación del método `receiveEvent()` se informa al sistema de control central del tiempo de recepción de ese evento invocando un método remoto del mismo. Ese valor es insertado en la cola con el resto de valores de relojes locales consultado previamente.

3. se calcula el mínimo de todos los valores y se actualiza el valor de GVT.

Este método no es aproximado, ni siquiera ninguno de los que se proponen en la literatura, ya que los eventos siguen ejecutándose en paralelo al mismo tiempo que se realiza este cálculo, pero representa una buena aproximación. El cálculo se ejecuta periódicamente cada s segundos. Cuanto menor sea s mejor se reflejará el estado actual del sistema pero también mayor será la carga computacional de los procesos que deben ser consultados.

3.5.7 Detalles de la implementación

En las secciones anteriores se expuso de manera descriptiva el funcionamiento de cada una de las partes involucradas en el sistema. En la sección que nos ocupa se expondrá mediante diagramas de bloques y de flujo el funcionamiento del sistema en su conjunto.

En la figura 3.2 se expone de manera gráfica el flujo de acciones del sistema en su conjunto. Los procesos reciben mensajes de otros procesos que se ejecutan bien en el mismo procesador o bien en distinto procesador. Estos mensajes se almacenan de manera ordenada y *concurrente* dentro de la cola de eventos de entrada.

Cada procesador tiene un *scheduler* que tiene acceso a las colas de eventos de entrada de todos los procesos ejecutándose en dicho procesador y su cometido es la de seleccionar el siguiente evento más idóneo a ejecutar. La idoneidad o no viene determinada por un algoritmo 3.5.4 que determina el siguiente evento. Cuando el evento es seleccionado, el procesador invoca el método `execute()` del proceso para que ejecute la acción que indica el evento seleccionado. Al ejecutar la acción, se produce, habitualmente, una modificación de las variables de estado. Estas variables se almacenan de manera que se tiene una evolución temporal de las mismas para poder realizar los procesos de vuelta atrás. Además, el evento ejecutado se transforma en su anti-evento y se almacena en otra cola de eventos ejecutados por el mismo motivo que se almacenan las variables de estado. Cuando se produce una situación de marcha atrás se envían todos los eventos de esta cola (anti-eventos) a los correspondientes procesos.

Entrando más en detalle en las dos partes fundamentales del sistema, procesos y procesadores, a continuación se presentan los diagramas de flujo de cada uno de ellos. En la figura 3.3 se muestra el diagrama de flujo de las acciones realizadas por un procesador.

El procesador es un proceso lógico que se ejecuta en un único procesador. Éste es el encargado de analizar las colas de eventos de cada uno de los procesos. Por lo tanto, su ejecución es un bucle gobernado por las acciones de `start()`, `stop()` y `pause()`, por lo que, en cada iteración, comprueba el valor de dicho valores booleanos. Si no se encuentra ni parado ni pausado, hace uso del objeto interno `Scheduler` para, de entre las colas de eventos de los procesos que se ejecutan bajo su tutela, seleccionar el siguiente mediante la instrucción `next()`. Una vez identificado el evento y el proceso correspondiente a ese evento, invoca el método `execute()` de dicho proceso y continua con una nueva iteración (figura 3.3).

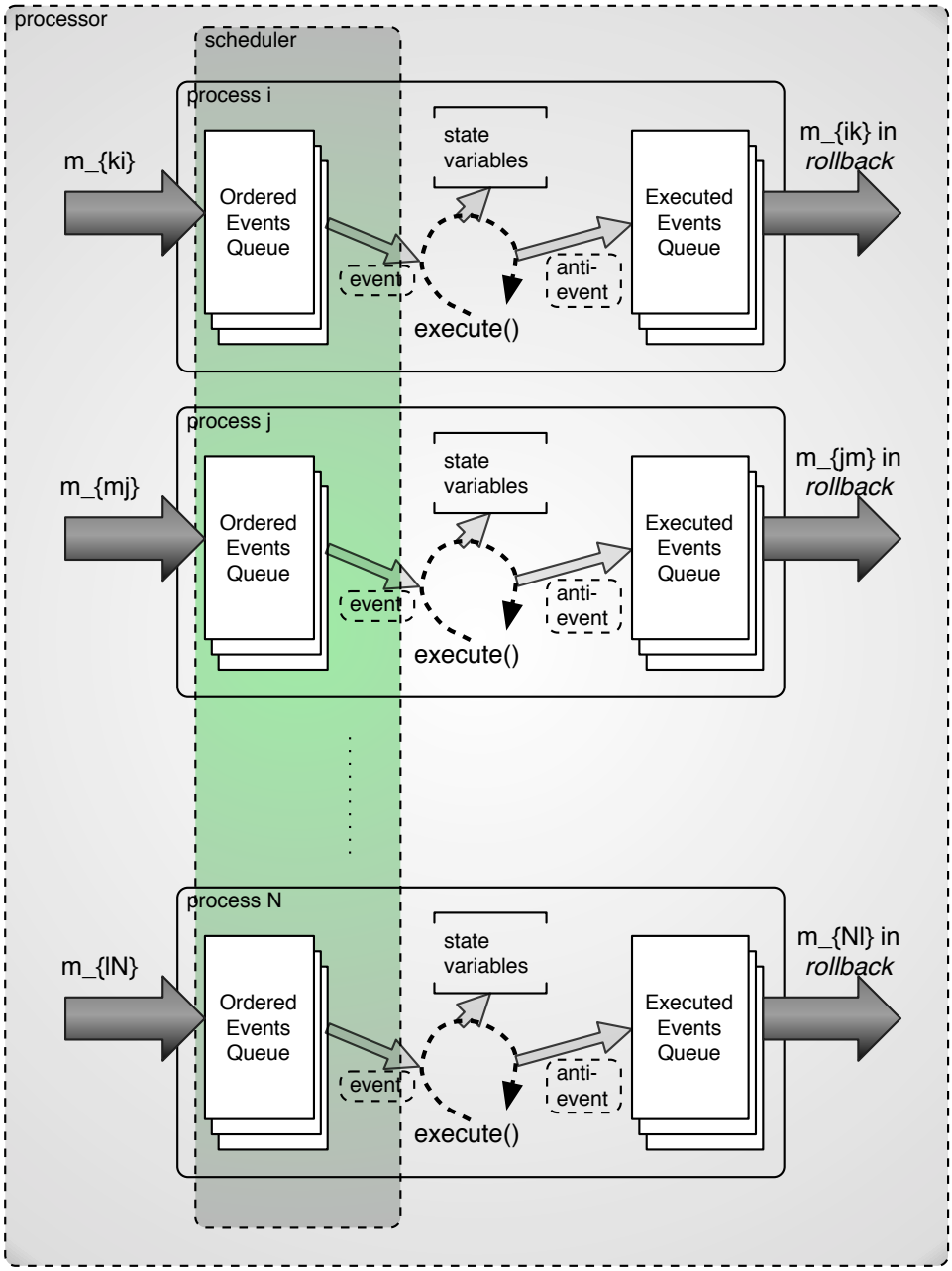


Figura 3.2 : Funcionamiento del sistema distribuido

Los procesos, como ya se ha comentado previamente, se ejecutan a instancias de un procesador. Es, el procesador, el elemento que gobierna la ejecución de un proceso y cuándo debe ejecutarse. En la figura 3.4 se muestra el diagrama de flujo de las acciones que son ejecutadas por un proceso. Entrando en detalle sobre la figura, en un proceso se ejecutan dos acciones en paralelo, una es la recepción de nuevos eventos provenientes de otros procesos (parte izquierda de la figura), y otra es la invocación, por parte del procesador, para ejecutar un evento (parte derecha de la figura).

La acción de recepción de eventos está implementada como un servidor basado en ZeroC ICE. Es decir, se ejecuta de manera autónoma paralelo al hilo principal de ejecución del proceso. Simplemente escucha

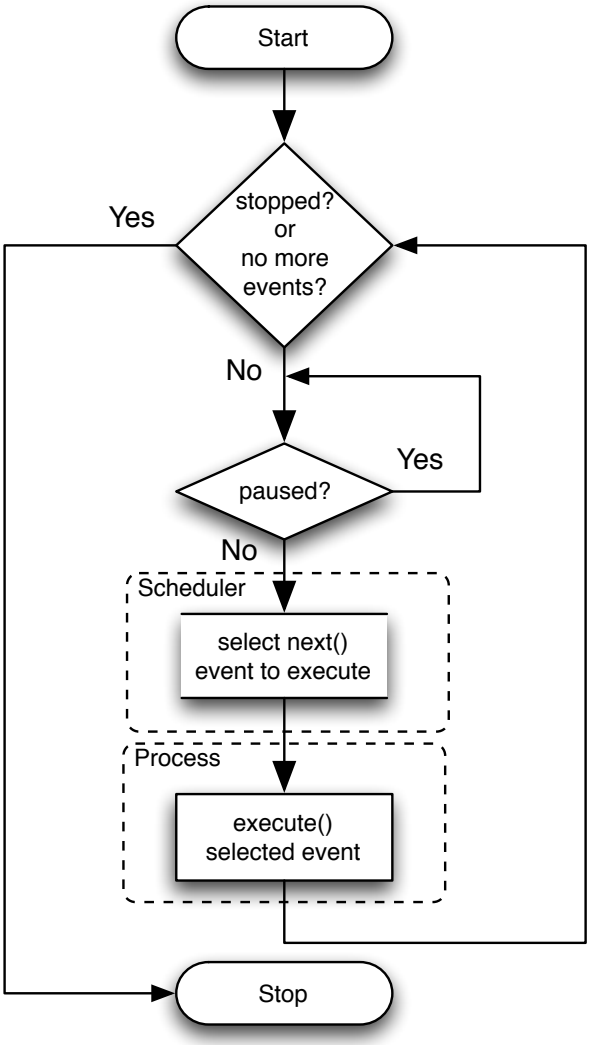


Figura 3.3 : Diagrama de flujo de las acciones de un procesador

invocaciones al método `receiveEvent()`. Lo primero que se comprueba, en el nuevo evento recibido, es si se trata de un evento normal o de un anti-evento. Si es un evento normal se comprueba si se ha violado el principio de causalidad, en cuyo caso se ejecuta la acción de vuelta atrás hasta el instante temporal previo al nuevo evento recibido, y si no se inserta el evento en la cola. Si, por el contrario, se trata de un anti-evento se comprueba si su evento correspondiente ha sido ejecutado en cuyo caso se procede con la acción de vuelta atrás, y si no, se elimina el evento de la cola.

Por otro lado, la acción de ejecución de eventos está implementada dentro del hilo de ejecución principal del proceso. No es una invocación que se pueda realizar de manera remota, como en el caso de recepción de eventos, sino que debe ser ejecutada por el procesador. Cuando se invoca el método `execute()`, el proceso lee de la cola de eventos el primer evento y ejecuta la acción definida por el objeto `IEvent`. Una vez ejecutada, inserta el correspondiente anti-evento en la cola de eventos ejecutados. De la misma manera se almacena el estado de las variables anterior al actual y se actualiza el valor del reloj local de ejecución. En este caso no se trata de un hilo recurrente, sino que es una serie de acciones secuencial

con principio y fin y que se ejecuta cada vez que se procede a consumir un evento.

Figura 3.4 : Diagrama de flujo de las acciones de un proceso

CAPÍTULO 4

IMPLEMENTACIÓN DE UN SIMULADOR DE REDES DE SENSORES DISTRIBUIDO

4.1 Introducción a la simulación de Redes de Sensores

Las redes de sensores inalámbricas no se comportan como las redes inalámbricas o alámbricas tradicionales, fundamentalmente por su carácter de funcionamiento ajeno a la infraestructura. Éstas realizan labores de medición de determinadas variables, en algunos casos realizan algún procesado sobre esa información, y la transmiten a otros nodos para ser propagada a un punto de fusión de datos.

Como en muchos otros campos profesionales, la simulación de los sistemas se erige como fundamental antes de realizar la puesta en escena de los mismos. Esto permite el ahorro de costes que se incurren cuando es necesario depurar un sistema existente. Las redes de sensores, habitualmente, se caracterizan por altas densidades espaciales de ocupación por la naturaleza de las magnitudes a medir, lo cual implica un alto número de comunicaciones. Otro elemento diferenciador de las redes de sensores es su organización. Mientras que la gran mayoría de redes inalámbricas desarrolladas hasta el momento son utilizadas en modo infraestructura, éstas suelen caracterizarse por su asociación *ad-hoc*.

Elevadas densidades de nodos sensores genera niveles de tráfico de comunicaciones inter-nodo muy alto. Este hecho provoca que la interferencia de otros nodos esté altamente correlada con la información que transmite el nodo de interés. Además, dependiendo de la naturaleza de la magnitud medida, este número de comunicaciones puede dispararse. Por lo tanto, sensores muy próximos y transmisiones muy elevadas va a conducir a graves problemas de colisiones de paquetes. Esto, dependiendo del protocolo implementado, llevará a una elevada tasa de pérdida de paquetes. A esto, cabe unirle el efecto de la interferencia. Aunque, individualmente, la potencia recibida de un nodo lejano no sea suficiente para corromper la recepción de un mensaje, si existen muchos nodos *lejanos*, la suma de toda la potencia debida a interferencia sí puede llevar a que la recepción se vea perturbada.

Tenemos, pues, que las Redes de Sensores Inalámbricas presentan una serie de particularidades que hacen que su planteamiento sea diferente al de las redes en infraestructura tradicionales. En este capitulo abordaremos la manera de definir una estructura de simulación coherente con esta naturaleza.

4.2 Propuesta de un simulador de Redes de Sensores distribuido

El incremento en el tiempo de simulación cuando se realizan diseños que involucren Redes de Sensores Inalámbricas densas es prohibitivo en la mayoría de los casos. La utilización del cálculo distribuido puede

ser una solución para poder fraccionar el problema y ejecutar simulaciones pseudo-independientes[1] de determinados fragmentos de la red.

Se va a implementar un simulador de Redes de Sensores Inalámbricas basándose en la arquitectura descrita en el capítulo 3 de modo que se exploten las capacidades de computación paralela en múltiples procesadores. Para ello, cada nodo de la red se va a ejecutar en un proceso lógico independiente, por lo que, de manera efectiva, la paralelización se hará por grupos de nodos (tantos como número de procesadores). El simulador deberá, por tanto, implementar determinados aspectos de la arquitectura distribuida, además de definir el comportamiento propio de un nodo sensor en una red inalámbrica que implemente la torre de protocolos correspondiente.

4.2.1 Planteamiento del problema

El problema fundamental que se debe tener en mente a la hora de implementar el simulador es la simulación y resolución de las *colisiones* de paquetes, en nuestro caso, y de la *interferencia*, en futuros trabajos. Además, es importante, también, la implementación de la capa MAC ya que ayudará a tratar el problema de las colisiones en determinados momentos.

El cálculo de dichas colisiones debe ser realizado por cada elemento de la red, de manera que monitorice el envío de paquetes de su vecindario. Es por tanto, fundamental, que cada nodo conozca qué otros elementos lo rodean y lo interfieren para *interrogarles* cada vez que se dispongan a realizar una transmisión. Para detectar una colisión es necesario tener en cuenta varios parámetros:

- instante del inicio de la transmisión del nodo
- tiempo de transmisión del paquete (duración)

con esto, se consulta a todos aquellos nodos, que pueden interferir en la transmisión, cuyo instante de inicio de transmisión o instante de fin de transmisión está dentro del intervalo definido por los dos parámetros anteriores. La figura 4.1 muestra varios ejemplos de colisión. Las franjas verdosas indican las zonas en las que se producen colisiones.

Para que la ejecución de la simulación pueda ser distribuida sobre la arquitectura presentada en el capítulo 3, la implementación de la lógica del nodo de red debe seguir unas reglas definidas por dicha infraestructura. Gracias a esta construcción es fácil implementar diferentes lógicas de simulación e intercambiarlas transparentemente. Dicho interfaz es denominado `INode` y define las funciones que serán invocadas desde el proceso lógico ejecutado en un procesador de la arquitectura distribuida:

- `saveStatus()` indica cómo y qué variables se deben almacenar en memoria ante un posible fallo de causalidad y por lo tanto, la ejecución de una marcha atrás.
- `rollback()` indica cómo se debe recuperar el estado previo al error de causalidad.
- `execute()` define las acciones a realizar por el sensor. Esto es, define la lógica de la red de sensores.

[1] Existe la posibilidad de fraccionar el problema en subproblemas disjuntos en las que partes de la red no interaccionan con otras pero, en general, siempre existirá un intercambio de mensajes entre bloques de nodos considerados independientes.

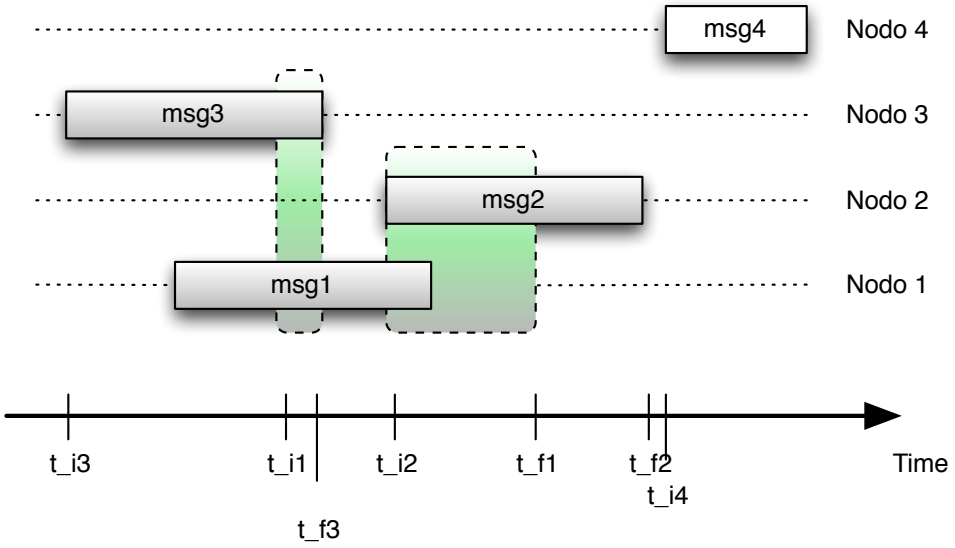


Figura 4.1 : Situaciones de colisión y no colisión

4.2.2 Detalles de la implementación

En el capítulo 3, se explicó que los procesos lógicos ejecutan los eventos almacenados en su cola de eventos. Estos eventos definen la acción a ejecutar mediante el interfaz `INode`, por lo que sólo se tiene que descargar el código de comportamiento del nodo, inicializarlo y ejecutarlo. Además, existen otros dos elementos específicos del algoritmo de Jefferson [84] que son el *rollback* y el almacenamiento de los estados de los procesos en cada instante. Esta implementación debe ser proporcionada, también, específicamente ya que depende de las características del problema. Por tanto, en nuestro simulador, cada componente susceptible de ser ejecutado de manera distribuida, debe proporcionar una implementación de estas funcionalidades de manera que el sistema quede perfectamente definido y cerrado.

Para definir una estructura a modo de nodo de red, además de los métodos requeridos por el interfaz `INode`, se define también otra funcionalidad más específica del problema en el interfaz `ISensor`. Éste hereda del anterior los métodos mencionados previamente y añade los específicos del comportamiento de un nodo de una Red de Sensores Inalámbrica:

- `setTxPower()`, `getTxPower()`: se utilizan para configurar la potencia de transmisión del sensor y, como consecuencia, determina el vecindario del nodo.
- `setSensitivity()`, `getSensitivity()`: se usan para configurar la sensibilidad en recepción del nodo. Influye en el alcance del vecindario.
- `getLocation()`: devuelve la información de posición geográfica del sensor.
- `doNETProtocol()`, `doMACProtocol()` y `doPHYProtocol()`: definen el comportamiento del sensor en cada capa.
- `updateNeighbours()`, `getNeighbours()`: configura el vecindario del nodo. Esta información es fundamental para el proceso de detección de colisiones.

- `addRoute()`, `removeRoute()` y `getRoutes()`: mantiene la información de las tablas de rutas del nodo.

Como complemento a la simulación se definen otros elementos auxiliares que permiten la identificación de los elementos de la red:

- `Packet` es el objeto que representa el mensaje que debe ser enviado de un nodo de la red a otro. Los paquetes se encapsulan en eventos de la arquitectura distribuida.
- `Route`, como se mencionó antes, representa la resolución del siguiente salto para llegar desde un nodo de la red a otro.

Para facilitar las labores de simulación se ha desarrollado un entorno gráfico (figura 4.2) que nos muestra la topología de la red, la información editable de los nodos, el progreso de la simulación y los mensajes que va generando y estado de la plataforma ParADisE, es decir, los recursos que estamos utilizando.

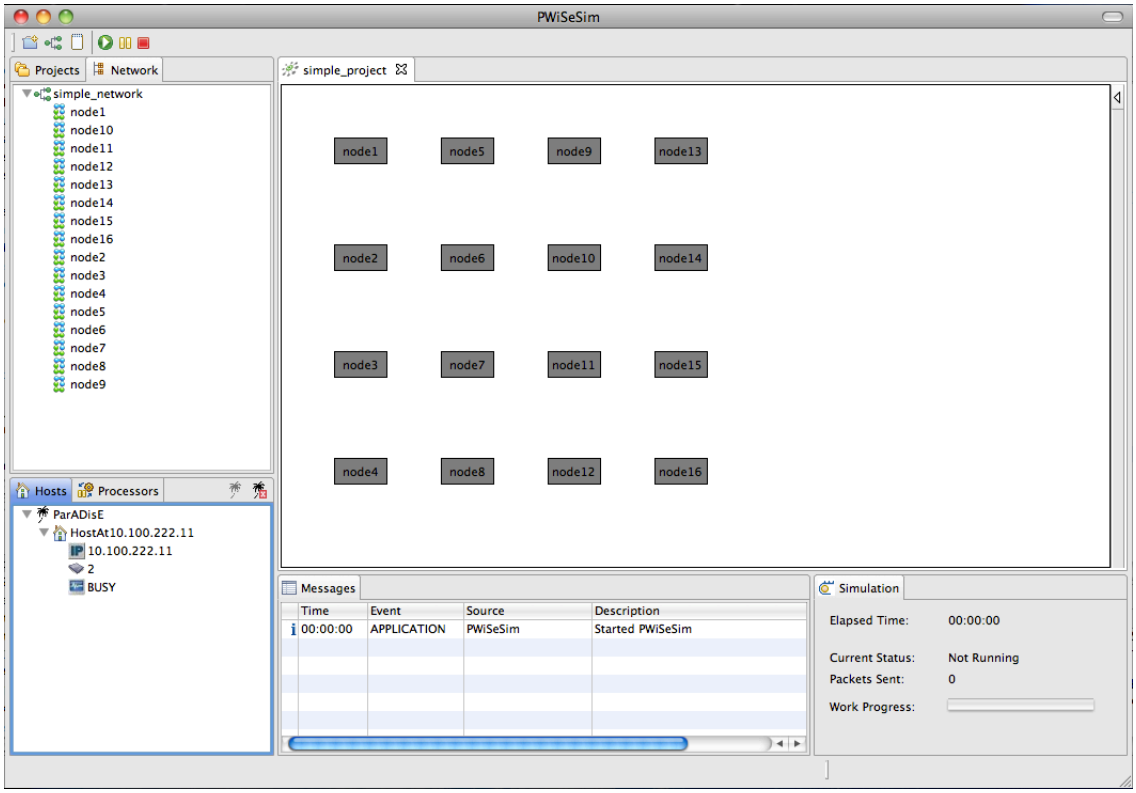


Figura 4.2 : Entorno gráfico del simulador

Flujo de trabajo de la simulación

El simulador se implementa como un cliente de la arquitectura distribuida, `IClient`. Éste descubre, mediante configuración, el elemento proveedor del servicio de ParADisE y se conecta para, posteriormente, requerir servicios de cómputo. En concreto, este cliente descubre el `SimulationServcieProvider` y solicita un `requestService()` donde se le asignan los servidores remotos con capacidad de proceso

acorde a la complejidad del problema. A partir de ahí, el cliente interactua con el `SimulationService` proporcionado ya que es este elemento el que gestionará el uso de los recursos asignados en función del problema.

El usuario define la topología de una red de nodos mediante un archivo de texto en el que se definen las coordenadas de cada sensor, la potencia transmitida y su sensibilidad. Estos parámetros podrán modificarse con la herramienta gráfica a posteriori, pero es necesario proporcionar una definición previa del problema.

Cada nodo puede ser editado individualmente de manera que se puede:

- modificar sus coordenadas, potencia transmitida y sensibilidad del receptor.
- definir la cola de eventos del nodo. Se pueden definir los destinos de los paquetes desde un nodo en particular o generarlos de manera aleatoria con destinos aleatorios.

Este método no es viable en la simulación de redes de nodos masivas, pero sí nos facilita el trabajo a la hora de hacer pruebas de concepto y generar casos particularizados en el que los flujos de paquetes deben seguir un patrón determinado.

Con todos los datos definidos, se puede lanzar la simulación. Es en este momento cuando se disecciona el problema y se asignan los nodos a los procesadores disponibles con todos los datos relacionados a cada nodo. Es decir, con sus colas de eventos y otra información necesaria para la ejecución. En este punto cabría proponer un problema de optimización en función de cómo se distribuyen los paquetes a simular dentro de la red y sus destinos pero esto queda fuera del alcance de este proyecto y será considerado en el apartado de trabajos futuros. Esta división en subproblemas es notificada al `SimulationService` que distribuirá la carga de trabajo en los recursos asignados. Para que las acciones que realizan los nodos puedan ser ejecutadas en los servidores remotos, aparte de las colas de eventos, es necesario que los procesos conozcan el código que deben ejecutar. Por ello, también debe notificarse dicho código fuente acorde con el interfaz establecido, como ya se comentó, en `INode`.

Describiéndolo de otra manera, es en el cliente donde se define el problema y cómo resolverlo: definición de la red y sus mensajes, definición del código fuente de la lógica de la simulación, etc; y en la arquitectura distribuida donde se ejecuta en paralelo para incrementar las prestaciones en cuanto a tiempo de simulación.

Una vez finalizada la simulación, el `SimulationService` notifica al cliente el final de la misma para que el usuario sea informado. Los resultados de la simulación son una serie de líneas informativas sobre las acciones que se realizan dentro de la simulación, es decir, transmisiones de paquetes, reenvíos, colisiones, tiempos, etcétera, así como si se han producido violaciones en la causalidad y, en consecuencia, situaciones de *rollback*. Dichas líneas deben ser procesadas a posteriori para el análisis del protocolo en encaminamiento.

Descripción de la capa de Red (NET)

Uno de los objetivos de construir un simulador de Redes de Sensores Inalámbricas distribuido es la de permitir una optimización de protocolos de encaminamiento a través de capas y el análisis de determinados

parámetros influyentes en la comunicación. Por ello, se define un interfaz común a todos los esquemas de encaminamiento de manera que pueda ser integrado fácilmente en el simulador.

Este interfaz, que se presentó anteriormente y que debe ser implementado para definir la lógica del algoritmo, define los siguientes métodos:

- `doNETProtocol()` define el comportamiento de la capa de red de la red a simular, es decir, cómo deben encaminarse los paquetes para alcanzar su correspondiente destino. En la implementación debe distinguirse entre transmisión y recepción.
- `doMACProtocol()` define el comportamiento de la capa de acceso al medio. En la implementación debe distinguirse entre transmisión y recepción. En el caso de que se implemente detección de colisiones en transmisión debe proporcionarse la lógica para el reenvío del paquete.
- `doPHYProtocol()` define el comportamiento de la capa física. En la implementación debe distinguirse entre transmisión y recepción. Debe proporcionarse la lógica para la detección de colisiones en recepción.

Estos métodos son particulares de la implementación del simulador pero, a su vez, debe ser implementado el interfaz que permitirá la ejecución de la simulación dentro de la arquitectura distribuida, es decir, los métodos de `rollback()` y `saveStatus()`, que dependen directamente de la implementación del simulador pero que son utilizados por ParADisE.

Como se ha indicado, la lógica del protocolo de encaminamiento debe proporcionarse en el método `doNETProtocol()`. Aquí se puede o describir su funcionamiento mediante codificación o utilizar alguna tabla de encaminamiento previamente indicada en el archivo de definición de la red a simular. En los casos de simulación que propondremos, cada nodo tiene almacenado en memoria su tabla de encaminamiento. Esta tabla se ha definido mediante un archivo de configuración que se lee también en el proceso de carga de la topología de la red.

El protocolo de encaminamiento implementado para las pruebas en este proyecto está fijado a priori. Este hecho nos simplifica la implementación del comportamiento de la red de sensores, puesto que el objetivo de este proyecto no es el diseño o análisis de un protocolo de encaminamiento concreto, sino la construcción de una herramienta que permita simular múltiples protocolos y su medida de prestaciones. El algoritmo diseñado, y plasmado en la tabla de rutas, es el de menor número de saltos hacia el nodo destino. En la tabla 4.1 se muestran los siguientes saltos para cada nodo en dicho protocolo. En la tabla, las columnas representan el nodo origen de la comunicación y las filas el nodo destinatario del mensaje.

Descripción de la capa de Acceso (MAC)

La lógica en la capa de acceso al medio debe implementarse en el método `doMACProtocol()`. Lo habitual en esta capa es la resolución de colisiones en transmisión, es decir, la implementación de un algoritmo/protocolo que evite las colisiones con otros paquetes.

En la implementación que nos ocupa en los ejemplos presentados, el método de acceso al medio implementado es un CSMA/CA (*Carrier Sense Multiple Access/Collision Avoidance*). Gracias a la infraestructura ParADisE se puede interrogar a los vecinos por sus eventos procesados en un determinado instante. Ana-

	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	N16
N1	X	1	2	3	1	1	2	3	5	5	6	7	9	10	10	11
N2	2	X	2	3	2	2	2	3	5	6	6	7	9	10	10	11
N3	2	3	X	3	2	3	3	3	6	6	7	7	9	10	11	11
N4	2	3	4	X	2	3	4	4	6	7	7	8	10	10	11	12
N5	5	5	2	3	X	5	6	7	5	5	6	7	9	10	10	11
N6	6	6	6	3	6	X	6	7	6	6	6	7	9	10	10	11
N7	6	7	7	7	6	7	X	7	6	7	7	7	10	10	11	11
N8	6	7	8	8	6	7	8	X	6	7	8	8	10	11	11	12
N9	5	5	6	3	9	9	6	7	X	9	10	11	9	9	10	11
N10	5	6	6	7	10	10	10	7	10	X	10	11	10	10	10	11
N11	6	6	7	7	10	11	11	11	10	11	X	11	10	11	11	11
N12	6	7	7	8	10	11	12	12	10	11	12	X	10	11	12	12
N13	5	5	6	7	9	9	10	11	13	13	10	11	X	13	14	15
N14	5	6	6	7	9	10	10	11	14	14	14	11	14	X	14	15
N15	6	7	7	7	10	11	11	11	14	15	15	15	14	15	X	15
N16	6	7	7	8	10	11	11	12	14	16	16	16	14	15	16	X

Tabla 4.1 : Protocolo de encaminamiento implementado para las pruebas de concepto

lizamos si nuestro paquete a enviar colisiona en tiempo con los eventos de los vecinos y, si es así, se retarda la transmisión del paquete un tiempo aleatorio y se vuelve a introducir en la cola de eventos.

En cuanto a la parte de recepción se refiere, la capa MAC sería la responsable de realizar las comprobaciones oportunas de errores de bit (CRC, *Cyclic Redundancy Check*). En el caso implementado para este proyecto se ha asumido un canal que no introduce perturbaciones sobre los mensajes transmitidos. De esta manera, logramos simplificar la lógica de la simulación y nos evitamos la implementación de los algoritmos de CRC que implican un coste computacional grande dependiendo del algoritmo utilizado.

Descripción de la capa Física (PHY)

Por último, quedaría por implementar la funcionalidad de la capa física. Para ello hay que dotar de contenido lógico al método `doPHYProtocol()`. Al tratarse de un simulador de redes inalámbricas, el canal es aéreo y, por lo tanto, dado a todo tipo de perturbaciones propias de un entorno inalámbrico.

En los ejemplos presentados, la colisión es el factor fundamental que nos influirá para tener una correcta recepción. Para ello, se interroga a los nodos vecinos si tienen eventos encolados en un intervalo temporal entorno al tiempo de recepción del mensaje. Si existen paquetes dentro de ese intervalo existirá colisión. Otro factor que, para tener simulaciones más realistas, puede ser considerado es la interferencia. Los niveles de potencia provenientes de nodos más allá de los vecinos, aunque pequeña, en suma, puede tomar un valor elevado y corromper la recepción del paquete. Este factor se planteará como trabajos futuros pero queda fuera del objetivo de este trabajo.

En cuanto a la transmisión se refiere, es en este método donde se debería implementar el comportamiento del canal incluyendo la tipología del canal y cómo afecta a la transmisión. En el caso que nos ocupa, una vez más con afán de simplificar el simulador, se ha optado por un canal transparente que no afecta a las transmisiones más que en una atenuación de la señal.

4.3 Cálculo de prestaciones del nuevo simulador

Finalmente, con toda la infraestructura de computación y diseñado e implementado el simulador, resta realizar algún ejemplo de simulación para comprobar:

- la fidelidad del simulador. Es decir, comprobar que las simulaciones dan lugar a resultados correctos.
- las prestaciones del simulador. Es necesario comprobar si la simulación distribuida supone ventajas sobre una simulación secuencial.
- viabilidad de implementaciones más complejas o integración con otros simuladores. Si los resultados obtenidos están a favor del procesado distribuido, ¿será viable integrar la arquitectura distribuida sobre otros simuladores como OMNET++?

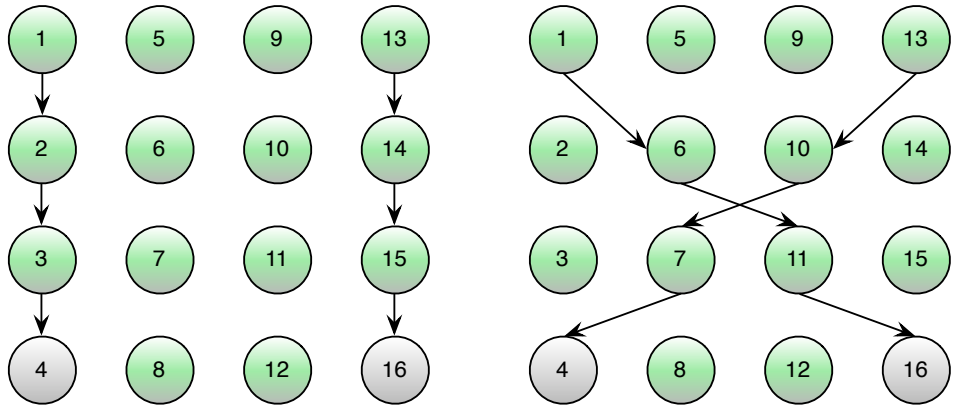
Para comprobar los puntos anteriores se propone la simulación de una red sencilla como la que se muestra en la figura 4.3. En ella se representan los dos casos de simulación: en el primero se implementará dos comunicaciones paralelas en las que los nodos involucrados en un camino no interfieren en los del otro camino; en la segunda existirá un cruce entre los dos caminos de comunicación.

4.3.1 Condiciones de simulación

Para llevar a cabo la simulación, se van a fijar una serie de condiciones y asunciones que facilitarán el trabajo.

Como se indicó, en la figura 4.3 se muestran los dos tipos de comunicación que se van a implementar. Para medir la escalabilidad[2] del sistema se realizarán agrupaciones de nodos por número de procesadores involucrados. Se partirá de realizar la simulación en un único procesador hasta seis procesadores. Para llevar a cabo las pruebas, se ha empleado un servidor con 2 procesadores, cada uno con cuatro núcleos, y 24 GB de memoria RAM.

En cada ejemplo hay dos nodos sumideros, uno para cada camino de la comunicación, al que el resto de nodos envían sus mensajes, estos nodos son el *nodo 4* y el *nodo 16*. En la figura 4.3a, se muestra cómo será la comunicación paralela y en 4.3b la comunicación cruzada. Lo que se pretende ilustrar con estos ejemplos es que en el caso de las comunicaciones disjuntas, al dividir el problema y ejecutarlo en paralelo, sí podemos obtener una clara mejora en las prestaciones de la simulación debido a que los errores de causalidad van a ser pocos; por el contrario, en el caso de comunicaciones cruzadas, las violaciones de causalidad pueden ser más numerosas obligando a realizar muchas acciones de marcha atrás que degenerarían, incluso más allá, las prestaciones de la simulación secuencial.



(a) Comunicaciones disjuntas **(b)** Comunicaciones cruzadas

Figura 4.3 : Ejemplos de simulación propuestos

Es por tanto, un factor importante cómo se realiza la división del problema en función del flujo de comunicaciones. Este problema podría plantearse como un problema de optimización previo a la simulación, que puede proporcionar información adicional a la hora de diseñar u optimizar algoritmos de encaminamiento.

La red está configurada en potencia para que los nodos no vean más allá de su primer vecino. Es decir, estamos restringiendo el vecindario al primer salto. En esta configuración, el *nodo 10*, por ejemplo, recibiría directamente señal de los nodos *5, 6, 7, 9, 11, 13, 14* y *15*. En cuanto a la generación de paquetes, sólo algunos nodos son productores de información, el resto son simples retransmisores.

Para medir las prestaciones del simulador en cada caso se realizarán agrupaciones de nodos en función

[2]En este caso, la escalabilidad debería medirse aumentando el número de nodos pero, debido a la restricción introducida sobre las tablas de encaminamiento estáticas, haría muy laborioso este proceso. Es por ello, por lo que se ha decidido simular la red de 16 nodos pero aumentando el número de procesadores involucrados en la simulación.

del número de procesadores disponibles. Con el equipo disponible podemos hacer hasta 8 divisiones, pero se comprobó que a partir de 7 procesadores las simulaciones entran en graves procesos de *rollback* haciendo que las ejecuciones se eternicen y no sea rentable continuar con las simulaciones. Por ello se han presentado resultados hasta 6 procesadores (agrupaciones).

4.3.2 Resultados de las simulaciones

Tras la ejecución de todas la realizaciones planteadas para cada uno de los casos y obtener los tiempos empleados para la simulación en cada caso se obtienen los resultados presentados en la figura 4.4. El trazo rojo representa la simulación de la red con comunicaciones paralelas y el azul el de las comunicaciones cruzadas. Cabe comentar que únicamente se han tomado datos hasta seis procesadores ya que a partir de 7 procesadores las simulaciones tomaban un tiempo indefinido. No sería arriesgado en este caso afirmar que la simulación es estable hasta 6 procesadores y que la asíntota de la curva se encuentra en 7 procesadores.

De la gráfica presentada se puede interpretar que el aumento en el número de procesadores no tiene por qué ser beneficioso en términos de prestaciones. De hecho, más bien, todo lo contrario. Se degradan respecto a utilizar un único procesador. Se observa un mínimo cuando se tienen tres procesadores ejecutando acciones que interactúan entre sí. A partir de ese valor, ya con 4 procesadores, el tiempo de simulación comienza de nuevo a aumentar.

Si se analizan los estadísticos de las distintas realizaciones se tiene que los tiempos de simulación son bastante estables y únicamente difieren por la cantidad de acciones de marcha atrás que deben realizar. La dispersión de los tiempos aumenta, también, con el aumento en el número de procesadores. Este hecho es bastante razonable ya que, si existe un deterioro de prestaciones, también existirá una mayor dispersión en los tiempos debido, una vez más, a las violaciones de causalidad. Al haber un número mayor de grados de libertad, aumenta la probabilidad de que exista un error de causalidad y por lo tanto que haya un rango más amplio de tiempos de simulación total para un mismo caso. Este dato queda muy bien reflejado con la representación del diagrama de cajas.

En cuanto a las prestaciones obtenidas si se tienen comunicaciones paralelas o cruzadas no hay grandes diferencias. Como era de esperar, para el caso de un procesador, los resultados obtenidos para ambos casos son muy similares. De hecho los intervalos de confianza de los diagramas de caja solapan. Comienza a haber ligeras diferencias a partir de 2 procesadores. Mientras que en el primer caso las comunicaciones no interfieren, por ejecutarse cada una en un procesador, en el segundo caso sí lo hacen apareciendo situaciones de violación de causalidad. Es en 3 agrupaciones donde se obtiene el mínimo tiempo de simulación en ambos ejemplos y, para mayor número de agrupaciones, el tiempo de simulación aumenta de manera exponencial.

Este resultado no deja de ser sorprendente. Intuitivamente se puede pensar que cuantos más procesadores dispongamos en un cálculo mayor será la velocidad con que éste se realizará. Sin embargo esa es la situación ideal. La realidad es otra, y es que existe un límite en el que las computaciones en paralelo dejan de ser efectivas. Este límite viene determinado -más bien es un defecto de diseño- por la arquitectura definida por John von Neumann en 1945 [101], donde se diseña la estructura que, aún a día de hoy, poseen la mayor parte de las computadoras: unidad central de proceso, almacenamiento y hardware de comunicaciones entre ambos. El problema es que el elemento de comunicaciones entre la memoria y la unidad

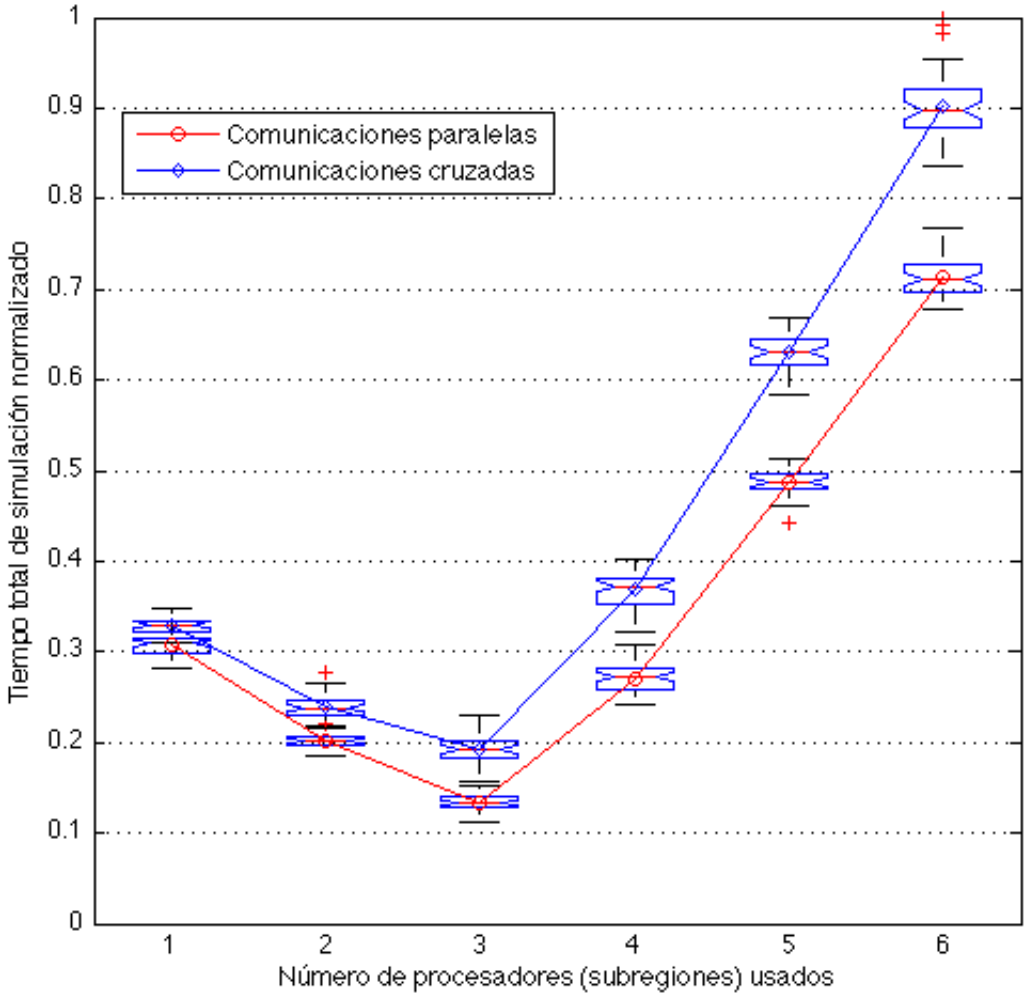


Figura 4.4 : Tiempo total de simulación para distinto número de agrupaciones. Se representan los valores obtenidos para los dos ejemplos de simulación propuestos

de proceso es lenta y no es capaz de transferir toda la capacidad de la memoria hacia el procesador, se convierte en un *cuello de botella* [97]. Este problema es aún más grave cuando existen múltiples unidades de procesamiento ya que no existen múltiples unidades de memoria, es compartida. Y, aún más grave, si cabe, cuando las unidades de procesamiento deben interactuar entre sí mediante el paso de mensajes. Es, pues, en este hecho, donde se explica la degradación de prestaciones cuando se aumenta el número de procesadores en juego.

Por último, cabe añadir un comentario acerca de la fidelidad de los resultados de la simulación atendiendo a las funciones del simulador. Es decir, ¿son coherentes los resultados obtenidos cuando existen situaciones de *rollback* y cuando no?¿Son los tiempos de entrega de paquete similares? Bien, en cada iteración, además de extraer los tiempos de comienzo y finalización, se realiza un análisis del tiempo medio de entrega de paquetes resultando que la media está entorno a los 210 mseg tomando como tiempos de transmisión 50 mseg y de procesado 10 mseg. Este resultado es coherente: en ambos ejemplos el número de saltos

desde los nodos origen a los nodos destino son 3, se consumen 150 mseg a los que hay que añadir los tiempos de procesado de cada evento que, al ser 4 nodos, suman 40 mseg adicionales. En total 190 mseg de manera determinista, sin colisiones ni procesos de marcha atrás, que añadirían, estadísticamente, los milisegundos para alcanzar el valor medio.

CAPÍTULO 5

CONCLUSIONES Y TRABAJOS FUTUROS

5.1 Conclusiones

Durante la realización de este trabajo, se han podido extraer muchas conclusiones, ya no sólo de los resultados de las simulaciones sino, también, de todo el proceso de diseño de la arquitectura. El objetivo último que generó todo este trabajo es poder explotar las capacidades actuales de cómputo sobre GPU para labores de simulación.

Una conclusión importante es que la arquitectura, tal y como está implementada, puede no suponer una mejora en las prestaciones de la simulación por diversos motivos. Al distribuir la computación sobre varias máquinas conectadas a la red, cuando existe un intercambio de mensajes entre dos procesos hospedados en diferentes máquinas, existe una comunicación a través de la misma. Esta comunicación está ligada al estado de congestión de la red y por lo tanto, lo que se gana en tratar el problema de manera distribuida se pierde en comunicaciones. Este mismo concepto se aplica a las comunicaciones entre varios procesadores dentro de una misma máquina y se le conoce como el *cuello de botella de von-Neumann*. En [97] se presenta cómo la arquitectura definida por von-Neumann presenta serios problemas cuando existen comunicaciones procesador-procesador y procesador-memoria.

Según [97], la computadora de von-Neumann se compone de tres partes bien diferenciadas: la unidad de proceso central, el almacenamiento y un elemento de comunicación entre el almacenamiento y la unidad de proceso. Este elemento de comunicación o *tubo*, como lo bautiza, es lo que se denomina *cuello de botella de von-Neumann*. En las máquinas con múltiples procesadores la memoria suele ser, además, compartida por todos, por lo que se añade un factor más a la ecuación. En [98] se plantea este problema con múltiples procesadores. En nuestro trabajo se ha demostrado dicha tesis en la que se aboga por una ralentización de la computación a medida que aumenta el número de procesadores (*parallel slowdown*). En la literatura, el valor empírico, y **no demostrado**, a partir del cual las prestaciones decrecen es 4 procesadores. En nuestro caso ha sido 3. Una explicación a este hecho es que se ha creado una infraestructura software adicional por encima del hardware+software de la máquina, lo cual añade un grado mayor de complejidad y, como consecuencia, mayor número de operaciones a realizar.

Este resultado indica que no por aumentar el grado de paralelismo entre procesos las prestaciones van a mejorar, más aún cuando deben existir comunicaciones entre los procesos. Obviamente, esto no aplica si la ejecución en cada procesador es independiente del resto, aunque sí se ve afectado por el hecho de compartir una misma memoria física de acceso secuencial.

Extrapolando estos resultados al aumento del número de nodos, es decir a redes densas, nos vamos a

encontrar con el mismo problema. En este caso, además, el número de nodos por procesador aumenta considerablemente. Se va a hacer necesario establecer un compromiso entre número de procesadores, que como hemos visto, no puede ser elevado, y procesos. Cabría comprobar estos resultados aplicados en GPU donde, gracias a sus características, se pueden tener numerosos procesos por procesador, amén de disponer de múltiples procesadores.

Por otra parte, tal y como se ha planteado la arquitectura, no es trivial realizar determinados cálculos. Durante toda la exposición se ha hablado que la interferencia es una aspecto fundamental en la simulación y que no se ha considerado en este trabajo. Con ParADisE únicamente podemos tener acceso a la potencia que transmiten cada uno de los vecinos, no siendo posible obtener la potencia del resto. Esto lo podemos solventar de varias maneras:

- introduciendo un elemento nuevo en la simulación que tenga la información de transmisión de todos los nodos de la red y al cual se pueda interrogar.
- cada nodo conoce una función de la intensidad de interferencia, $f(p_{tx}, s, x, y, z)$, a su alrededor que depende de la distancia y de las potencias de transmisión de cada nodo y sensibilidad del receptor.

Con la segunda opción, más razonable computacionalmente ya que elimina numerosas comunicaciones, requiere realizar un estudio previo de la red, en cuanto a potencia se refiere, para poder determinar dicha función.

En resumen, no queda muy claro que, sin más que distribuir la computación, se gane en prestaciones en tiempo de simulación. Para conseguir dicho objetivo es necesario tener otros aspecto en consideración que depende del problema en sí mismo. En nuestro caso, en la simulación de Redes de Sensores Inalámbricas, se hace necesario tener en cuenta los flujos de transmisión de mensajes a lo largo de la red. Si se obtienen determinados patrones de esos flujos, puede aplicarse una descomposición del problema que permita un número mínimo de interacciones entre los elementos que se ejecutan en diferentes procesadores. Este hecho, al contrario de como se podría pensar, puede ser un punto a favor a la hora de diseñar un nuevo protocolo de encaminamiento si se tiene en cuenta la naturaleza de las comunicaciones y las posibles agrupaciones. De hecho, los protocolos jerárquicos en Redes de Sensores Inalámbricas son los más extendidos por tener esta cuestión en consideración. En definitiva, si dichas jerarquías se diseñan teniendo en cuenta estos parámetros y, con ayuda de la parelización en simulación, sí se puede obtener una mejora sustancial en las prestaciones de tiempo de simulación.

5.2 Trabajos futuros

Las derivaciones que puede generar este Proyecto Fin de Máster son múltiples. Por un lado se ha definido e implementado una arquitectura de computación distribuida que puede ser empleado en entornos no sólo de simulación, sino también de cálculo distribuido. En concreto, el simulador de Redes de Sensores Inalámbrico en sí mismo se ha implementado adaptándose a la arquitectura presentada.

El simulador presentado es una versión muy básica de un simulador RSI ya que no tiene en cuenta múltiples parámetros necesarios para realizar diseños óptimos de algoritmos de encaminamiento, sobre todo, si se realiza teniendo en cuenta el diseño a través de capas. Uno de los parámetros fundamentales para realizar una correcta simulación es el cálculo de la interferencia. En una Red de Sensores Inalámbrica la

transmisión de datos, dependiendo de la magnitud a medir, puede ser muy elevada y requerir de numerosas comunicaciones. La capa MAC resuelve el problema de las colisiones entre nodos vecinos, pero la interferencia (suma) del resto de nodos puede superar el umbral de sensibilidad de los receptores radio y provocar que el mensaje recibido esté corrupto. Si la herramienta de simulación es capaz de tener en cuenta dicha interferencia, el diseño de las comunicaciones a través de capas resultará en un protocolo, entendiendo protocolo como la fusión de las tres primeras capas, eficiente.

Otra de las mejoras sustanciales en las capacidades del simulador sería la introducción de la movilidad en los nodos. Quizá en un simulador de Redes de Sensores Inalámbricas la movilidad no sea uno de los requerimientos básicos pero sí por ejemplo en estudios de tráfico automovilístico en los que se introduce el concepto de *espira virtual*. Dado que la arquitectura es genérica y no está diseñada para el problema de las Redes de Sensores Inalámbricas en sí mismo, se puede utilizar para la simulación de tráfico microscópico en el que se emulan las interacciones vehículo-vehículo (V2V) o vehículo-infraestructura (V2I). La movilidad está actualmente implementada en la arquitectura distribuida mediante un mecanismo de migración de procesos lógicos de un procesador a otro. Esta migración se realiza en varios pasos:

1. Si el proceso se está ejecutando en el instante de la migración se espera a su finalización.
2. Una vez finalizada la acción, se congela el proceso no permitiendo la ejecución de nuevos eventos de su cola de eventos.
3. Se extraen los parámetros característicos del proceso (`ProcessDescriptor`).
4. Con dicho descriptor, se crea un nuevo proceso en el procesador destino y se inicializa con dichos parámetros.
5. Se destruye el antiguo proceso en el antiguo procesador.

El simulador de Redes de Sensores Inalámbricas implementado en este trabajo no explota las capacidades de movilidad intrínsecas del sistema distribuido ya que, dado el tipo de simulación que se quería realizar, no ha sido necesario.

Dentro de la propia arquitectura distribuida se pueden realizar numerosas mejoras. La primera puede ser una generalización del concepto de `INode`. En la implementación realizada en Java, los nodos son clases específicas que indican las tareas que se deben realizar. Estas acciones, definidas en los nodos, residen o pueden residir en máquinas remotas y las clases pueden ser descargadas mediante `ClassLoader`s de Java. Una vez descargada la clase, se instancia localmente y se ejecuta. La generalización se realizaría mediante programas/aplicaciones no específicas Java. Es decir, desaparecerían los `ClassLoader` de Java y lo que se descargarían serían directamente ejecutables. A esto, además, se le podría añadir un grado más de seguridad utilizando conexiones SSH para la descarga de dichas acciones ejecutables independientemente.

En cuanto a componentes propios de la arquitectura expuesta en [84] mejorables, se puede actuar sobre el concepto, ya explicado, de los programadores o *schedulers*. En la versión más simple, el programador selecciona el evento con menor tiempo de ejecución como el siguiente a ser ejecutado de entre un grupo de eventos. Existen numerosas aproximaciones de diferentes implementaciones del programador [90][89][91]. En estas implementaciones, el programador explota determinadas características de incertidumbre temporal para establecer una ordenación óptima de los eventos a ejecutar. La implementación de estos nuevos

algoritmos significarían una mejora en la forma en la que se seleccionan los eventos para su ejecución en la arquitectura.

Otro componente de la arquitectura en sí misma es el cálculo del reloj global de la simulación (GVT). La implementación realizada únicamente tiene en cuenta el valor de los relojes locales de cada proceso, tomando com GVT el valor mínimo de todos ellos. Como ya se explicó, el cálculo de GVT es siempre aproximado ya que al tratarse de un sistema distribuido, en el instante en el que se calcula el nuevo valor ya *han sucedido* más cosas en el sistema y los estados de los procesos pueden haber cambiado. En [99] y en [100] se presentan dos métodos diferentes para el cálculo de lo que denominan *snapshots* o fotografías de la simulación en las que siempre declaran que no dejan de ser métodos aproximados para el cálculo debido a las características intrínsecas del problema.

Por último, una mejora sustancial, como ya se ha comentado, en cuanto a prestaciones del sistema, reside en su implementación. Al tratarse de un sistema paralelizable es fácil reproducirlo o implementarlo en las actuales herramientas hardware de *computación gráfica* (GPU, *Graphical Processing Units*). La arquitectura expuesta, íntegramente implementada en Java, debería migrarse al lenguaje típico de gestión de dichas unidades. Los sistemas más avanzados en la actualidad son los de la empresa nVidia [93] con su tecnología CUDA. Sin embargo, la migración no sería dramática ya que, de manera inteligente, se eligió ZeroC ICE para la definición de la arquitectura distribuida que, como ya se comentó, es multilenguaje y multiplataforma. En este caso, el código a migrar sería el correspondiente a las implementaciones del `Processor` y del `LogicalProcess`. La implementación, acorde con la definición *slice*, debería ser realizada en el lenguaje C de manera que se utilicen las particularidades que CUDA introduce en este lenguaje. La misma debe tener en cuenta la arquitectura CUDA de modo que se puedan explotar al máximo las capacidades de paralelización que la computación gráfica ofrece.

BIBLIOGRAFÍA

[1] R. Ahlswede, N. Cai, S. Y. Li, and R. Yeung, "Network information flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[2] ISO, "Open systems interconnection. basic reference model," International Organization for Standarization, Tech. Rep., 1984.

[3] R. Jurdak, *Wireless Ad Hoc and Sensor Networks. A Cross-Layer Design Perspective*. Springer, 2007.

[4] *IEEE Std. 802.11-1997 Information Technology-Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks. Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.*, IEEE, 1997.

[5] S. Corson and J. Macker, *Mobile Ad Hoc Networking (MANET): Routing Protocol Performance issues and Evaluation Considerations. RFC 2501*, IETF Network Working Group, 1999.

[6] Y. Chen and E. Fleury, *Handbook of Wireless Ad Hoc and Sensor Networks*. Springer, 2008.

[7] F. Theoleyre and F. Valois, *Wireless Ad Hoc and Sensor Networks*. ISTE Ltd (Hermès Science Publications/Lavoisier Company), 2008.

[8] T. Watteyne, M. Dohler, I. Augé-Blum, and D. Barthel, *Localization Algorithms and Strategies for Wireless Sensor Netowks*. IGI Global, 2008.

[9] T. S. Rappaport, *Wireless Communications: Principles and Practice*, 2nd ed. Prentice Hall PTR, 2002.

[10] W. Stallings, *Wireless Communications and Networks*, 2nd ed. Prentice Hall, 2004.

[11] A. F. Molisch, *Wireless Communications*. John Wiley and Sons, 2005.

[12] A. J. Goldsmith, *Wireless Communications*. Cambridge University Press, 2005.

[13] B. D. P. and R. Gallager, *Data Networks*, 2nd ed. Prentice Hall, 1992.

[14] M. C. Jeruchim, P. Balaban, and K. S. Shanmugan, *Simulation of Communication Systems. Modeling, Methodology and Techniques*, ser. Jack Keil Wolf. Kluwer Academic Publishers, 2000.

[15] W. H. Tranter, K. S. Shanmugan, T. S. Rappaport, and K. K. L., *Principles of Communication Systems Simulation with Wireless Applications*. Prentice Hall, 2004.

[16] M. K. Simon and M. S. Alouini, *Digital Communication over Fading Channels*, 2nd ed. John Wiley and Sons, 2005.

[17] M. K. Simon, S. M. Hinedi, and W. C. Lindsey, *Digital Communication Techniques: Signal Design and Detection*. Prentice Hall, 1995.

[18] S. L. Wu, Y. C. Tseng, and S. J. P., "Intelligent medium access for mobile ad hoc networks with busy tones and power control," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 9, pp. 1647–1657, 2000.

[19] R. Hekmat and P. Van Mieghem, "Interference power statistics in ad hoc and sensor networks," *Wireless Networks Journal (WINET), Springer*, vol. 14, no. 5, pp. 591–599, 2008.

[20] C. K. Toh, V. G. Vassiliou, and C. H. Shih, "March: A medium access control protocol for multihop wireless ad hoc networks," in *Proceedings of the IEEE Military Communications Conference*, MILCOM'00, Ed., 2000, pp. 512–516.

[21] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *Proceedings of the IEEE International Conference on Computer Communications*, INFOCOM'02, Ed., 2002, pp. 1567–1576.

[22] N. Jain, S. R. Das, and A. Nasipuri, "A multichannel mac protocol with receiver-based channel selection for multihop wireless networks," in *Proceedings of the IEEE International Conference on Computer Communication and Networks*, ICCN'01, Ed., 2001.

[23] Z. Haas and J. Deng, "Dual busy tone multiple access (dbtma): A multiple access control scheme for ad hoc networks," *IEEE Transactions on Communications*, vol. 50, no. 6, pp. 975–985, 2002.

[24] S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, *Mobile Ad Hoc Networking*. Wiley-IEEE Press, 2004.

[25] S. Ramanathan and M. Steenstrup, "A survey of routing techniques for mobile communications networks," *Mobile Networks and Applications*, vol. 1, no. 2, pp. 89–104, 1996.

[26] E. M. Royer and C. K. Toh, "A review of current routing protocols for ad hoc mobile wireless networks," *IEEE Personal Communications*, vol. 6, no. 2, pp. 46–55, 1999.

[27] J. J. Vinagre-Díaz, "Theory of routing in wireless ad hoc networks," Ph.D. dissertation, Carlos III University, 2007.

[28] V. Srivastava and M. Motani, "Cross-layer design: a survey and the road ahead," *IEEE Communications Magazine*, vol. 43, no. 12, pp. 112–199, 2005.

[29] T. Girici and A. Ephremides, "Joint routing and scheduling metrics in wireless ad hoc networks," in *Proceedings of 36th Asilomar Conference on Signals, Systems and Computers*, 2002.

[30] T. ElBatt and A. Ephremides, "Joint scheduling and power control for wireless ad hoc networks," *IEEE Transactions on Wireless Communications*, vol. 3, no. 1, pp. 74–85, 2004.

[31] R. Jurdak, "Modeling and optimization of ad hoc and sensor networks." Ph.D. dissertation, University of California, 2005.

[32] R. Merz, J. Widmer, J. Le Boudec, and B. Radunovic, "A joint phy/mac architecture for low radiated power th-uwb wireless ad hoc networks," *Wiley Wireless Communications and Mobile Computing Journal*, vol. 5, no. 5, pp. 567–580, 2004.

[33] K. Chen, S. H. Shah, and K. Nahrstedt, "Cross-layer design for data accesibility in mobile ad hoc networks," *Wireless Personal Communications: An International Journal*, vol. 21, no. 1, pp. 49–76, 2002.

[34] X. Wang and K. Kar, "Cross-layer rate control for end-to-end proportional fairness in wireless networks with random access," in *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2005, pp. 157–168.

[35] G. Lu and B. Krishnamachari, "Energy efficient joint scheduling and power control in wireless sensor networks," in *Proceedings of the 2nd IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, SECON'05, Ed., 2005, pp. 362–373.

[36] R. Madan, S. Cui, S. Lall, and A. J. Goldsmith, "Cross-layer design for lifetime maximization in interference-limited wireless sensor networks," *IEEE Transaction on Wireless Communications*, vol. 5, no. 11, pp. 3142–3152, 2006.

[37] G. Lu and B. Krishnamachari, "Minimum latency joint scheduling and routing in wireless sensor networks," *Ad Hoc Networks Journal (Elsevier)*, vol. 5, no. 6, pp. 832–843, 2007.

[38] S. Cui, R. Madan, A. J. Goldsmith, and S. Lall, "Cross-layer energy and delay optimization in small-scale sensor networks," *IEEE Transactions on Wireless Communications*, vol. 6, no. 10, pp. 3688–3699, 2007.

[39] M. L. Sichitiu, "Cross-layer scheduling for power efficiency in wireless sensor networks," in *Proceedings of the IEEE International Conference on Computer Communications*, INFOCOM'04, Ed., vol. 3, 2004, pp. 1740–1750.

[40] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shnker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys'05, Ed., 2005, pp. 76–89.

[41] F. Fitzek and M. Katz, *Cooperation in Wireless Networks: Principles and Applications*. Springer, 2006.

[42] E. C. Van der Meulen, "Transmission of information in a terminal discrete memoryless channel," Ph.D. dissertation, University of California. Berkeley, 1968.

[43] ——, "Three-terminal communication channels," *Advanced Applied Probability*, vol. 3, pp. 120–154, 1971.

[44] T. Cover and A. El Gamal, "Capacity theorems for the relay channel," *IEEE Transactions on Information Theory*, vol. 25, no. 5, pp. 572–584, 1979.

[45] J. N. Laneman, "Cooperative diversity in wireless networks: Algorithms and architectures," Ph.D. dissertation, Washington University, 2002.

[46] A. Sendonaris, E. Erkip, and B. Aazhang, "User cooperation diversity. part i: System description," *IEEE Transaction on Communications*, vol. 51, no. 11, pp. 1927–1938, 2003.

[47] ——, "User cooperation diversity. part ii: Implementation aspects and performance analysis," *IEEE Transaction on Communications*, vol. 51, no. 11, pp. 1939–1948, 2003.

[48] J. N. Laneman and G. W. Wornell, "Distributed space-time coded protocols for exploiting cooperative diversity in wireless networks," *IEEE Transaction on Information Theory*, vol. 49, no. 10, pp. 2415–2425, 2003.

[49] M. O. Hasna and M. S. Alouini, "Harmonic mean and end-to-end performance of transmission systems with relays," *IEEE Transaction on Communications*, vol. 52, no. 1, pp. 130–135, 2004.

[50] J. N. Laneman, D. N. C. Tse, and G. W. Wornell, "Cooperative diversity in wireless networks: Efficient protocols and outage behaviour," *IEEE Transaction on Information Theory*, vol. 50, no. 12, pp. 3062–3080, 2004.

[51] A. Stefanov and E. Erkip, "Cooperative codeing for wireless networks," *IEEE Transactions on Communications*, vol. 52, pp. 1470–1476, 2004.

[52] J. Boyer, D. Falconer, and H. Yanikomeroglu, "Cooperative connectivity models for wireless relay networks," *IEEE Transactions on Wireless Communications*, vol. 6, no. 6, pp. 1992–2000, 2007.

[53] C. Fragouli, J. Y. Le Boudec, and J. Widmer, "Network coding: An instant primer," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 63–68, 2006.

[54] P. Sanders, S. Egner, and L. Tolhuizen, "Polynomial time algorithms for network information flow," in *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'03, Ed., 2003, pp. 286–294.

[55] C. Chekuri, C. Fragouli, and E. Soljanin, "On average throughput and alphabet size in network coding," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2410–2424, 2006.

[56] Z. Li and B. Li, “Network coding: The case of multiple unicast sessions,” in *Proceedings of the 42th Allerton Annual Conference on Communication, Control and Computing*, 2004.

[57] D. S. Lun, M. Medard, and R. Koetter, “Network coding for efficient wireless unicast,” in *International Zurich Seminar on Communications*, 2006, pp. 74–77.

[58] Z. Guo, B. Wang, P. Xie, W. Zeng, and J. H. Cui, “Efficient error recovery with network coding in underwater sensor networks,” *Ad Hoc Networks Journal. Elsevier Science Publishers B. V.*, vol. 7, no. 4, pp. 791–802, 2009.

[59] D. Lucani, M. Medard, and M. Stojanovic, “Underwater acoustic networks: Channel models and network coding based lower bound to transmission power for multicast,” *IEEE Journal on Selected Areas in Communications*, vol. 26, no. 9, pp. 1708–1719, 2008.

[60] S. Y. Li, R. Yeung, and N. Cai, “Linear network coding,” *IEEE Transactions on Information Theory*, vol. 49, no. 2, pp. 371–381, 2003.

[61] Y. Wu, “Network coding for multicasting,” Ph.D. dissertation, Princeton University, 2006.

[62] C. Fragouli, D. Katabi, A. Markopoulou, M. Medard, and H. Rahul, “Wireless network coding: Opportunities and challenges,” in *Proceedings of the Military Communications Conference*, MILCOM’07, Ed., 2007, pp. 1–8.

[63] Y. E. Sagduyu and A. Ephremides, “On joint mac and network coding in wireless ad hoc networks,” *IEEE Transactions on Information Theory*, vol. 53, no. 10, pp. 3697–3713, 2007.

[64] A. Eryilmaz, A. Ozdaglar, M. Medard, and E. Ahmed, “On the delay and trhoughput gains of coding in unreliable networks,” *IEEE Transactions on Information Theory*, vol. 54, no. 12, pp. 5511–5524, 2008.

[65] C. Fragouli, J. Widmer, and J. Y. Le Boudec, “Efficient broad-casting using network coding,” *IEEE/ACM Transactions on Networking*, vol. 16, no. 2, pp. 450–463, 2008.

[66] Y. E. Sagduyu and A. Ephremides, “Cross-layer optimization of mac and network coding in wireless queueing tandem networks,” *IEEE Transaction on Information Theory*, vol. 54, no. 2, pp. 554–571, 2008.

[67] X. Bao and J. Li, “Adaptive network coded cooperation (ancc) for wireless relay networks: Matching code-on-graph with network-on-graph,” *IEEE Transactions on Wireless Communications*, vol. 7, no. 2, pp. 574–583, 2008.

[68] D. N. Yang and M. S. Chen, “Data broadcast with adaptive network coding in heterogeneous wireless networks,” *IEEE Transactions on Mobile Computing*, vol. 8, no. 1, pp. 109–125, 2009.

[69] K. Fall and K. Varadhan, *The ns Manual*.

[70] E. Altman and T. Jiménez, *NS Simulator for Beginners*, Los Andes University, 2003.

[71] [Online]. Available: http://pcl.cs.ucla.edu/projects/parsec/

[72] [Online]. Available: http://www.scalable-networks.com/products/qualnet/

[73] F. Meyer auf der Heide, C. Schindelhauer, K. Volbert, and M. Grönewald, “Congestion, dilation and energy in radio networks,” *Theory of Computing Systems*, vol. 37, no. 3, pp. 343–370, May 2004.

[74] E. C. of Postal and T. A. (CEPT), “Monte-carlo simulation methodology for the use in sharing and compatibility studies between different radio services or systems,” European Radiocommunications Committee, Tech. Rep., June 2002.

[75] B. D. P. and T. J. N., *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[76] B. V. C., *Massively Parallel Models of Computation*. Ellis Horwood Limited, 1993.

[77] D. L. Mills, “Internet time synchronization: the network time protocol,” *IEEE Transaction on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[78] J. Liu, “Scalable synchronization of clocks in wireless sensor networks,” *Ad Hoc Networks*, vol. 6, pp. 791–804, 2008.

[79] D. L. Mills, “Network time protocol (version 3) specification, implementation and analysis network time protocol. specification, implementation and analysis,” IETF Network Working Group, RFC 1305, March 1992.

[80] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[81] R. M. Fujimoto, “Parallel discrete event simulation,” in *Proceedings of the 1989 Winter Simulation Conference*, 1989.

[82] ——, “Parallel and distributed simulation systems,” in *Proceedings of the 2001 Winter Simulation Conference*, 2001.

[83] K. M. Chandy and J. Misra, “A distributed algorithm for detecting resource deadlocks in distributed systems,” in *PODC ’82: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1982, pp. 157–164.

[84] D. R. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[85] G. A., “Rollback mechanisms for optimistic distributed simulation systems,” in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, no. 3. SCS Multiconference on Distributed Simulation, July 1988, pp. 61–67.

[86] L. M. Sokol, D. P. Briscoe, and A. P. Wieland, “Ntw: A strategy for scheduling discrete simulation events for concurrent execution,” in *SCS Multiconference on Distributed Simulation*, 1988, pp. 34–44.

[87] T. K. Som and R. G. Sargent, “A probabilistic event scheduling policy for optimistic parallel discrete event simulation,” in *Proceedings 12th Workshop on Parallel and Distributed Simulation*, 1998.

[88] F. Quaglia, “A state-based scheduling algorithm for time warp synchronization,” in *Annual Simulation Symposium*, 2000.

[89] R. M. Fujimoto, “Exploiting temporal uncertainty in parallel and distributed simulations,” in *Proceedings of the 13th Workshop on Parallel and Distributed Simulations*, 1999.

[90] R. Beraldi and L. Nigro, “Exploiting temporal uncertainty in time warp simulations,” in *Proceedings of the IEEE 4th International Workshop on Distributed Simulation and Real-Time Applications*, 2000.

[91] B. Lubachevsky, A. Shwartz, and A. Weiss, “Rollback sometimes works... if filtered,” in *Proceedings of the 1989 Winter Simulation Conference*, 1989.

[92] E. Morgado-Reyes, “Prestaciones de las redes ad hoc inalámbricas: Teoría a través de capas,” Ph.D. dissertation, Universidad Rey Juan Carlos, 2009.

[93] (2010). [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[94] [Online]. Available: www.zeroc.com

[95] T. K. Som and R. G. Sargent, “A probabilistic event scheduling policy for optimistic parallel discrete event simulation,” in *12th Workshop on Parallel and Distributed Simulation.*, January 1998.

[96] F. Quaglia, “A state-based scheduling algorithm for time warp synchronization,” in *Proceedings of the 33rd Annual Simulation Symposium*, April 2000.

[97] J. Backus, “Can programming be liberated from the vonneumann style? a functional style and its algebra of programs,” *Communications of the ACM*, vol. 21, no. 8, pp. 613–642, August 1978.

[98] A. Z. Salamon and V. Galpin, “Bounds on series-parallel slowdown,” *Distributed, Parallel, and Cluster Computing*, vol. Submitted, 2009.

[99] R. Koo and S. Toueg, “Checkpointing and rollback-recovery for distributed systems,” *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, January 1987.

[100] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, February 1985.

[101] J. von Neumann, “First draft of a report,” US Army Ordenance Dept. and University of Pennsylvania, Report, 1945.

[102] X. Li, *Wireles Ad Hoc and Sensor Networks. Theory and Applications*. Cambridge University Press, 2008.