UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE MATEMÁTICA, ASTRONOMÍA, FÍSICA Y COMPUTACIÓN.

Plataformas para redes tolerantes a demoras sobre nodos integrados en un chip

Tesis realizada por Rubén Danilo Capkob para la Licenciatura en Ciencias de la Computación en la Universidad Nacional de Córdoba

---

Dirigida por:
Dr. Ing. Pablo Alejandro Ferreyra

Agradecimientos

- A toda mi familia por la paciencia y la contención,
- A mis amigos y compañeros de estudio por el apoyo constante,
- A mi amigo y director Pablo, Gracias.

Resumen

Desde sus inicios y hasta ahora, la concepción de los sistemas tolerantes a fallas se ha basado en el uso de redundancias principalmente locales. Sin embargo, la tecnología de redes ha permitido recientemente el surgimiento de otras alternativas más distribuidas. En efecto, las Redes Tolerantes a Demoras, denominadas ***DTNs*** por sus siglas en inglés, aparecieron en escena hace algunos años y son diseñadas para ser robustas frente a demoras y disrupciones de cualquier índole.
Esto hace que cualquier tipo de falla en un conjunto de nodos DTN pueda mapearse en un retraso, y por lo tanto, puede tolerarse y contrarrestarse adecuadamente. De esta manera, es posible imaginar topologías de redes que posean, entre otras, la característica de degradación paulatina (en inglés: ***Gracefull Degradation***). Con dicha propiedad, las fallas se convierten en demoras o disminución de la performance.
Hasta ahora, las redes DTN se han aplicado en escenarios con distancias relativamente largas y con interrupciones también proporcionalmente largas entre los enlaces de sus nodos. Recientemente, se ha propuesto el estudio de las plataformas que permitan desarrollar redes DTN operando en distancias entre nodos de varios órdenes de magnitud menor: DTN dentro de un Chip o más precisamente System On Chip (SOC).
Para estas redes se ha propuesto el nombre ***WDTNOC*** (por sus siglas en inglés, ***Wireless Delay Tolerant Networks On Chips***, para describir DTNs en una escala a nivel de chip, donde las demoras y los intervalos de tiempo son reducidos en concordancia a este nuevo dominio de aplicación. También se ha propuesto el nombre ***WDTNBC*** para representar un concepto similar de redes entre chips, (***Wireless Delay Tolerant Networks Between Chips***).

Este trabajo se enfoca en investigar acerca de las capacidades de tolerar fallas en algunas plataformas o topologías para las WDTNOCs y WDTNBCs y sus implicancias. Para ello, se realizan implementaciones de modelos de eventos discretos usando el framework OMNeT++ para caracterizar distintas

métricas de interés en dichas redes.
Como conclusiones generales se demuestra que, bajo condiciones de fallas transitorias, estas redes presentan la muy importante y deseada propiedad de ***Degradación Paulatina***. Adicionalmente, se puede ver que esta propiedad es escalable con el número de nodos. Hay un intercambio directo, pero no necesariamente lineal, entre ***performance*** y tasas de fallas transitorias. Es decir, a mayor tasa de fallas transitorias en los nodos, la red tiene una performance menor, pero sigue funcionando.
Además de la capacidad de tolerar fallas transitorias, es posible demostrar que ciertas topologías para las WDTNOCs pueden tolerar también algún conjunto de fallas permanentes en sus nodos.
Con este trabajo se da un primer paso hacia un nuevo paradigma de arquitecturas tolerantes a fallas y escalables, que sin lugar a dudas se terminará imponiendo en el desarrollo de muchos sistemas críticos. Más aún, estos sistemas prometen ser capaces de tolerar tanto a fallas en el estado operativo de su ciclo de vida, como así también durante la etapa de su desarrollo.
De esta manera, además de dar origen a un nuevo paradigma para nuevas arquitecturas de computadoras tolerantes a fallas, dará origen también a nuevas formas de desarrollar y calificar a sistemas críticos complejos.
El uso de tales sistemas críticos complejos y de tiempo real, será una actividad creciente en campos como el aeroespacial y en nuevas tecnologías actualmente en desarrollo tales como los vehículos automáticos de uso civil y sin conductores.

Índice general

Capítulo 1

Introducción

En este capítulo introductorio, se describen los antecedentes, motivaciones, los objetivos, la importancia, el impacto, el contenido y la organización del presente trabajo.

1.1. Antecedentes y Motivaciones

La necesidad de tolerar fallas surge del hecho empírico de que, por más esfuerzos que se hagan para evitarlas, siempre existirá una cierta probabilidad distinta de cero para su ocurrencia. Dichas fallas ocurren desde el momento en que se están redactando las especificaciones funcionales y no funcionales, hasta que los sistemas se encuentran operativos y hasta en operaciones de mantenimiento, pasando por todo el proceso de desarrollo, implementación, verificación, testeo y validación del mismo. Más aún, los sistemas suelen tener ciclos de ***renovación*** donde tienden a ser actualizados o rediseñados y/o re implementados. Dichos procesos de renovación también son susceptibles a las fallas. Para disminuir la probabilidad de ocurrencia de problemas graves durante la fase de operación y mantenimiento de un sistema se usan variantes de las técnicas de redundancia tales como ***NMR***, [1–3], y ***EDACS***, [4–6]. Para disminuir la probabilidad de ocurrencia de problemas graves durante la fase de especificación, diseño, desarrollo y validación, se usan técnicas tales como por ejemplo ***Design Diversity*** [7–9]. Éstos últimos también pueden usarse para las posteriores etapas de actualización y de rediseño y re implementación, también llamadas renovaciones.
En épocas relativamente recientes han surgido conceptos más novedosos tales como las Arquitecturas Segmentadas (***AS***) [10–12] que tratan de resolver de una manera eficiente el problema de las fallas a lo largo de todas las etapas del ciclo de vida de un sistema, incluso su posteriores renovaciones o

actualizaciones. Es decir, desde la etapa de concepción y desarrollo, hasta la operación y mantenimiento del mismo. Más aún, permiten un proceso de mantenimiento, e incluso mejoras o actualizaciones paulatinas del sistema, como así también su degradación paulatina y reconfiguración en operación, entre otras tantas muy deseables propiedades y prestaciones. Dichas AS se basan en la existencia de módulos independientes pero que trabajan de manera colaborativa interconectados en redes principalmente inalámbricas.
Hay estudios recientes que vinculan fuertemente a las AS con las Redes Tolerantes a Demoras conocidas como (DTNs) por sus siglas en inglés. Las DTNs, [13–19], son hoy en día un campo muy activo de investigación y desarrollo y encajan perfectamente en el contexto de las AS. Las DTNs han sido principalmente concebidas para trabajar sobre sistemas con largas distancias entre nodos y en escenarios donde otros servicios de redes regulares no pueden aplicarse por ser altamente disruptivos o por la falta de la infraestructura requerida, [20–22,24]. Ejemplos de la aplicación de las DTNs pueden ser encontrados en los campos de las comunicaciones interplanetarias e intersatelitales donde las distancias entre los nodos es considerable. Sin embargo, es prácticamente lo mismo para un nodo DTN soportar las altas demoras o disrupciones debidos a las largas distancias o a las oclusiones planetarias o a la falla de un nodo vecino. De hecho, un nodo que no está respondiendo debido a una falla transitoria, produce el mismo efecto que un nodo volando en el lado opuesto de un planeta remoto, por ejemplo.
En el micro mundo de los ***SOCs*** ("System On Chips") existen los mismos problemas y desafíos que en el macro mundo de las DTNs interplanetarias, a la hora de tolerar las fallas y a lo largo de todo el ciclo de vida de estos micro sistemas. Lo único que cambia es la escala de tiempos y distancias. En particular, las ***WNOCs*** ("Wireless NetWorks On Chips") son un caso especial de SOCs donde podría implementarse una versión escalada de las DTNs para tolerar y contrarrestar todo tipo de fallas, a lo largo de todo el ciclo de vida del sistema. Como beneficio adicional las WDTNOCs podrían solucionar muchos de los problemas de las NOCs (Networks On Chips) cableadas tradicionales tales como consumo, retrasos, recalentamientos, área ocupada, complejidad de interconexiones, etc.
Resulta entonces muy motivadora e interesante la idea de combinar la tecnología de las DTNs con la tecnología WNOCs. Dicha combinación ha sido nombrada como ***WDTNOCs***, por sus siglas en inglés (Wireless Delay Tolerant Networks on Chips).

1.2. Objetivos del Trabajo

1.2.1. Objetivo General Elegido

Como lo indica el título del presente trabajo, *"Plataformas para redes tolerantes a demoras sobre nodos integrados en un chip"*, el objetivo principal es investigar dichas plataformas. Este objetivo general cubre prácticamente un sin número de posibles temáticas. Dentro de este universo de orientaciones se ha elegido enfocar los esfuerzos principalmente el siguiente objetivo general:

- **Desarrollar Modelos para Estudiar la Performabilidad las DT-NOCs**

Cabe aclarar que el vocablo *Performabilidad* es un abuso de lenguaje para indicar lo que se pretende es estudiar la performance y la confiabilidad o disponibilidad de manera simultánea.

1.2.2. Otros Objetivos Generales

Otros objetivos generales que se habían considerado inicialmente tales como:

- Definir los requerimientos Generales de implementación de los nodos DTNOCs
- Investigar sobre la factibilidad de implementar DTNOCs sobre plataformas basadas en nodos RISC-V o similares
- Investigar sobre la factibilidad de fabricar los nodos DTNOC RISC-V o similar por medio del ecosistema "efabless" [1] o similar

han sido dejados para posteriores continuaciones del presente trabajo.

1.2.3. Objetivos Específicos

En función del objetivo general elegido para desarrollar, se han resuelto los siguientes objetivos específicos:

- Dominar la herramienta OMNET++ para implementar los modelos.
- Implementar una simulación con OMNET++ de una red lineal del tipo TCP.

[1] https://www.efabless.com/

- Implementar una simulación con OMNET++ de una red lineal del tipo DTN.
- Comparar resultados.
- Planificar Trabajos futuros que sean la continuación de éste.

Como se mencionó los objetivos específicos anteriores, están relacionados con el objetivo general elegido. Otros objetivos específicos relacionados con los objetivos generales previstos como continuación de este trabajo, han sido ya abordados. Por ejemplo los siguientes:

- Identificar e Investigar posibles Arquitecturas de Redes DTNOCs.
- Investigar plataformas para desarrollo y validación tales como MyHDL.
- Identificar cadenas de herramientas de software libre para el futuro desarrollo de los nodos DTNOCs.

1.3. Importancia del proyecto - Impactos esperados

Las arquitecturas segmentadas son muy promisorias para tolerar fallas en todos los estados del ciclo de vida de un sistema, desde la etapa de su definición conceptual inicial hasta las etapas de su mantenimiento en estado operativo. Más aún, facilitan los llamados *ciclos de renovación*, en los cuales nuevos diseños y con mayores prestaciones pueden ser desarrollados y puestos en operación mientras sus predecesores siguen operativos. En otras palabras, es posible imaginar sistemas que tienen infinitos ciclos de renovaciones, mientras continúan prestando sus servicios, y pueden ser continuamente mejorados, sin interrupciones de ningún tipo.
Esta idea es aplicable a muchos sistemas críticos embebidos actualmente en desarrollo, tales como las formaciones y constelaciones satelitales, como así también a todo tipo de vehículos no tripulados, entre ellos, los vehículos sin conductor.
El aspecto más relevante a resaltar como impacto y contribución pretendida de este trabajo es que una vez desarrolladas las WDTNOCs, podrían "transmitir" de manera natural sus características "genéticas" hacia niveles más altos de abstracción de los sistemas. Como ya se mencionó, resulta directo extrapolar la idea de las WDTNOCs hacia ***WDTNBCs*** ("Wirless Delay Tolerant Networks Between Chips") es decir la interconexión inalámbrica

tolerante a demoras entre chips. De allí se puede seguir extrapolando hacia arriba, por ejemplo hacia la idea de ***WDTNBBs*** ("Wirless Delay Tolerant Networks Between Boards") es decir la interconexión inalámbrica tolerante a demoras entre placas. Así se puede llegar naturalmente hacia las Arquitecturas Segmentadas más generales heredando en cada paso de abstracción superior, las características ("genéticas") antes mencionadas de las capas de abstracción más bajas. Entre otras, la capacidad de tolerancia a fallas para todos los ciclos de vida de los sistemas, y hasta para sus futuras renovaciones.

Es importante recalcar que este trabajo ha sido presentado en la edición 2020 del Congreso Argentino de Sistemas Embebidos "*(CASE 2020)*". Además es muy grato informar que se ha recibido una invitación por parte del Comité Organizador de *CASE* para enviar una versión ampliada del manuscrito "*Hop Counts and End to End Delays in Linear Wireless Delay Tolerant Network on Chips Subject to Transient Faults*" al número especial "*Latest Advances in Embedded Systems Research in Latin America*" de la revista "*International Journal of Embedded Systems*", "*(IJES)*"[2].

1.4. Estructura de la Tesis

La tesis está estructurada de la siguiente manera:

- En el capítulo 2, se dan nociones generales orientadas a las redes de interconexión utilizadas en los *SOCs* y fallas en los circuitos digitales.
- En el capítulo 3, se dan descripciones básicas de las redes DTN.
- En el capítulo 4, se describen las caracterísitcas principales del simulador OMNET++, que se utiliza para modelar los sistemas estudiados.
- En el capítulo 5, se describe el primer caso de estudio, una red lineal de 4 nodos tipo TCP.
- En el capítulo 6, se describe el segundo caso de estudio, una red lineal de 4 nodos tipo DTN.
- En el capítulo 7, se discuten los resultados obtenidos
- En el capítulo 8, se proponen conclusiones y trabajos futuros.

[2] https://www.inderscience.com/jhome.php?jcode=ijes

Capítulo 2

Nociones Preliminares

2.1. Conceptos elementales

Los sistemas digitales [25] son dominantes en la sociedad moderna. Las computadoras digitales se utilizan para tareas que van desde la simulación de sistemas físicos hasta la gestión de grandes bases de datos y la preparación de documentos. Los sistemas de comunicación digital transmiten llamadas telefónicas, señales de vídeo y datos de Internet. El entretenimiento de audio y vídeo se entrega y procesa cada vez más en forma digital. Finalmente, casi todos los productos, desde automóviles hasta electrodomésticos, están controlados digitalmente.

Como vemos, cada vez más y más elementos alrededor del ser humano están relacionados con esta tecnología y por ende afectan directamente su calidad de vida. Ésto hace que cada vez sea más importante analizar el impacto de las fallas en dichas tecnologías digitales y en cómo contrarrestarlos

Un sistema digital se compone de tres componentes básicos: lógica, memoria y comunicación. La lógica transforma y combina datos, por ejemplo, realizando operaciones aritméticas o tomando decisiones. La memoria almacena datos para su posterior recuperación, moviéndolos cuando sea necesario. La comunicación mueve datos de una ubicación a otra.

En cualquiera de los tres componentes anteriores pueden ocurrir fallas. Así mismo, la manera de contrarrestarlas puede hacerse también en cualquiera de ellos.

En este capítulo tratamos sobre el componente de comunicación de los sistemas digitales. Específicamente, se explora redes de interconexión que se utilizan para transportar datos entre los subsistemas de un sistema digital, quienes reemplazaran a los "*buses*" de comunicación con "*redes de interconexión*".

Consecuentemente, se propone un modelo de fallas apropiado para abstraer el comportamiento de cada nodo de dichas redes.

2.2. Hardware de un sistema digital

Repasando brevemente, y para comprender qué sucede con un programa cuando lo ejecutamos, necesitamos comprender la organización del hardware de un sistema típico, como el que se muestra en la figura 2.1. Esta imagen en particular sigue el modelo de la familia Intel.

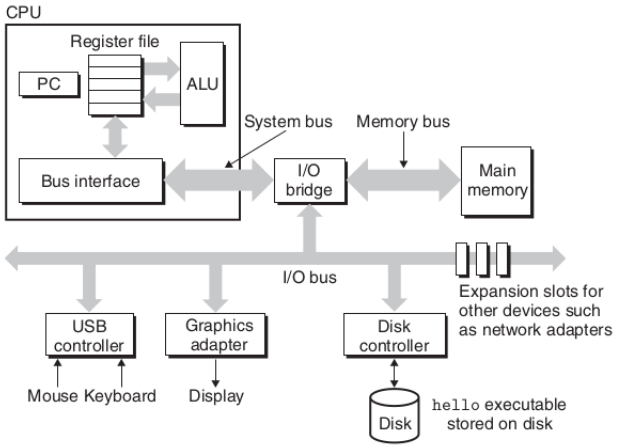


Figura 2.1: Organización del hardware de un sistema típico. **CPU**: Unidad central de procesamiento, **ALU**: Unidad aritmética/lógica, **PC**: Contador de programa, **USB**: Bus serie universal. [25]

1. **Buses**:

   Se encuentran en todo el sistema una colección de conductores eléctricos llamados "*buses*" que transportan bytes de información de un lado a otro entre los componentes. Los "*buses*" suelen estar diseñados para transferir fragmentos de bytes de tamaño fijo conocidos como palabras. El número de bytes en una palabra (el tamaño de la palabra) es un parámetro fundamental del sistema que varía de un sistema a otro. La mayoría de las máquinas actuales tienen tamaños de palabra de 4 bytes (32 bits) u 8 bytes (64 bits).

2. **Dispositivos E/S**:

   Los dispositivos de entrada/salida (E/S) son la conexión del sistema al mundo externo. Nuestro sistema de ejemplo (figura 2.1) tiene cuatro dispositivos de E/S: un teclado y un mouse para la entrada del usuario, una pantalla para la salida del usuario y una unidad de disco (o simplemente un disco) para el almacenamiento a largo plazo de datos y programas. Inicialmente, el programa ejecutable reside en el disco. Cada dispositivo de E/S está conectado al "*bus*" de E/S mediante un controlador o un adaptador. La distinción entre estos dos es principalmente la ubicación del dispositivo. Los controladores son conjuntos de chips en el propio dispositivo o en la placa del circuito impreso principal del sistema (a menudo denominada *placa base* o *madre*). Un adaptador es una tarjeta que se conecta a una ranura en la placa base. Independientemente, el propósito de cada uno es transferir información de ida y vuelta entre el bus de E/S y un dispositivo de E/S.

3. **Memoria Principal**:

   La memoria principal es un dispositivo de almacenamiento temporal que contiene tanto un programa como los datos que manipula mientras el procesador ejecuta el programa. Físicamente, la memoria principal consiste en una colección de chips de memoria dinámica de acceso aleatorio (DRAM). La memoria está organizada como una matriz lineal de bytes, cada uno con su propia dirección única (índice de matriz) que comienza en cero. En general, cada una de las instrucciones de la máquina que constituyen un programa puede consistir en un número variable de bytes. Los tamaños de los elementos de datos que corresponden a las variables del programa varían según el tipo.

4. **Procesador**:

   La unidad central de procesamiento (CPU), o simplemente procesador, es el motor que interpreta (o ejecuta) las instrucciones almacenadas en la memoria principal. En su núcleo hay un dispositivo de almacenamiento (o registro) del tamaño de una palabra llamado contador de programa (PC: Program Counter). En cualquier momento, el PC apunta a (contiene la dirección de) alguna instrucción en lenguaje máquina en la memoria principal.

   Desde el momento en que se aplica energía al sistema, hasta el momento en que se apaga la alimentación, un procesador ejecuta repetidamente la instrucción señalada por el contador del programa y actualiza el

contador del programa para que apunte a la siguiente instrucción. Un procesador parece funcionar de acuerdo con un modelo de ejecución de instrucciones muy simple, definido por la arquitectura del conjunto de instrucciones. En este modelo, las instrucciones se ejecutan en secuencia estricta, y ejecutar una sola instrucción implica realizar una serie de pasos. El procesador lee la instrucción de la memoria señalada por el contador del programa (PC), interpreta los bits en la instrucción, realiza una operación simple dictada por la instrucción y luego actualiza el PC para que apunte a la siguiente instrucción, que puede o no ser contiguo en memoria a la instrucción que se acaba de ejecutar.

Solo hay algunas de estas operaciones simples, y giran en torno a la memoria principal, el archivo de registros y la unidad aritmética/lógica (ALU). El archivo de registro es un pequeño dispositivo de almacenamiento que consiste en una colección de registros del tamaño de una palabra, cada uno con su propio nombre único. La ALU calcula nuevos datos y valores de dirección.

Estos son algunos ejemplos de las operaciones simples que la CPU podría realizar a pedido de una instrucción:

- Cargar: copiar un byte o una palabra de la memoria principal en un registro, sobrescribiendo los contenidos anteriores del registro.
- Almacenar: copiar un byte o una palabra de un registro a una ubicación en la memoria principal, sobrescribiendo el contenido anterior de esa ubicación.
- Operar: copia el contenido de dos registros en la ALU, realiza una operación aritmética con las dos palabras y almacena el resultado en un registro, sobrescribiendo el contenido anterior de ese registro.
- Salto: Extraer una palabra de la instrucción misma y copiarla en el contador del programa (PC), sobrescribiendo el valor anterior de la PC.

Decimos que un procesador parece ser una implementación simple de la arquitectura del conjunto de instrucciones, pero de hecho los procesadores modernos usan mecanismos mucho más complejos para acelerar la ejecución del programa. Por lo tanto, podemos distinguir la arquitectura del conjunto de instrucciones, del procesador, describiendo el efecto de cada instrucción de código máquina, de su microarquitectura, describiendo cómo se implementa realmente el procesador.

2.3. Capas de protocolos y sus modelos de servicio

Pongamos nuestra atención en los protocolos de red [26]. Para proporcionar estructura al diseño de protocolos de red, los diseñadores de red organizan los protocolos, (el hardware y software de red que implementan los protocolos), en **capas**. Cada protocolo pertenece a una de las capas. Estamos interesados en los **servicios** que una capa ofrece a la capa superior, el llamado modelo de servicio de una capa. Cada capa proporciona su servicio (1) realizando ciertas acciones dentro de esa capa y (2) utilizando los servicios de la capa directamente debajo de ella. Por ejemplo, los servicios proporcionados por la capa n pueden incluir la entrega confiable de mensajes desde un borde de la red al otro. Esto podría implementarse utilizando un servicio de entrega de mensajes de borde a borde poco confiable de la capa $n-1$, y agregando funcionalidad a la capa n para detectar y retransmitir mensajes perdidos.

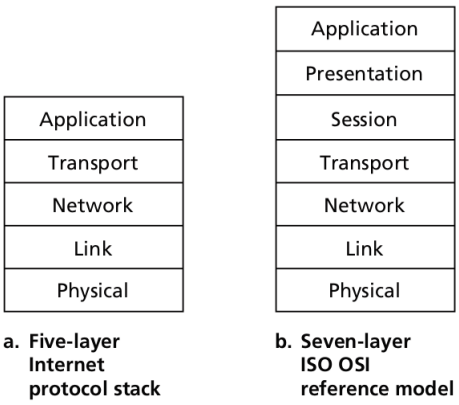


Figura 2.2: La pila de protocolos de Internet (a) y el modelo de referencia OSI (b). [26]

Una capa de protocolo se puede implementar en software, en hardware o en una combinación de los dos. Los protocolos de la capa de aplicación, como **HTTP** y **SMTP**, casi siempre se implementan en software en los sistemas finales; también los protocolos de la capa de transporte. Debido a que la capa física y las capas de enlace de datos son responsables de manejar la comunicación a través de un enlace específico, normalmente se implementan en una tarjeta de interfaz de red (por ejemplo, tarjetas de interfaz Ethernet o WiFi) asociada con un enlace determinado. La capa de red es a menudo una implementación mixta de hardware y software. Un protocolo de capa n está distribuido entre los sistemas finales, conmutadores de paquetes y otros

componentes que componen la red. Es decir, a menudo hay una parte de un protocolo de capa n en cada uno de estos componentes de red.
La superposición de protocolos tiene ventajas conceptuales y estructurales (RFC 3439). Las capas proporcionan una forma estructurada de discutir los componentes del sistema. La modularidad facilita la actualización de los componentes del sistema. Un posible inconveniente de las capas es que una capa puede duplicar la funcionalidad de la capa inferior. Por ejemplo, muchas pilas de protocolos proporcionan recuperación de errores tanto por enlace como de un extremo a otro. Un segundo inconveniente potencial es que la funcionalidad en una capa puede necesitar información (por ejemplo, un valor de una marca temporal) que está presente solo en otra capa; esto viola el objetivo de separación de capas.
Cuando se toman en conjunto, los protocolos de las distintas capas se denominan **pila de protocolos**. La pila de protocolos de Internet consta de cinco capas: las capas ***física***, de ***enlace***, de ***red***, de ***transporte*** y de ***aplicación***, como se muestra en la Figura 2.2(a).

2.3.1. Modelo OSI

Las siete capas del modelo de referencia OSI, que se muestran en la Figura 2.2(b), son: capa de ***aplicación***, capa de ***presentación***, capa de ***sesión***, capa de ***transporte***, capa de ***red***, capa de ***enlace de datos*** y capa ***física***. La funcionalidad de cinco de estas capas es aproximadamente la misma que la de sus contra partes de Internet con nombres similares. Por lo tanto, consideremos las dos capas adicionales presentes en el modelo de referencia OSI: la capa de ***presentación*** y la capa de ***sesión***. La función de la capa de presentación es proporcionar servicios que permitan que las aplicaciones de comunicación interpreten el significado de los datos intercambiados. Estos servicios incluyen la compresión de datos y el cifrado de datos (que se explican por sí mismos) así como la descripción de datos (que, libera a las aplicaciones de tener que preocuparse por el formato interno en el que se representan/almacenan los datos) formatos que pueden diferir de una computadora a otra). La capa de sesión proporciona la delimitación y sincronización del intercambio de datos, incluidos los medios para construir un esquema de puntos de control y recuperación.

En este trabajo (como se verá en los capítulos 5 y 6), se propone la simulación de dos redes inalámbricas lineales de cuatro nodos cada una. La primera de ellas, a la que denominamos ***Red TCP***, es básicamente una red de datagramas, por lo que involucra solo hasta la capa de red para los nodos intermedios para su simulación. La segunda, a la que denominamos ***Red DTN***, es un

poco mas compleja e involucra hasta la capa superior (de aplicación o en nuestro caso llamada “core”) más una capa intermedia entre la capa de aplicación y la capa de red denominada “bundle” (esto se explica con mayor detalle en el capítulo 3).
Sin embargo, si bien la implementación de los modelos respeta la modularidad de las capas, se realizan una serie de hipótesis simplificadoras (capítulo 5) que nos permitieron llevar adelante un primer trabajo y obtener resultados para analizar la factibilidad de estas hipótesis.

2.4. Redes de interconexión en chips

El rendimiento de la mayoría de los sistemas digitales actuales [27], como el descripto anteriormente está limitado por su comunicación o interconexión, no por su lógica o memoria. En un sistema de alta gama, la mayor parte de la energía es usada por la conducción a través de los cables o buses y la mayor parte del ciclo de reloj se gasta en el retraso debido a los cables, y no por el retraso de los puertos. A medida que la tecnología mejora, las memorias y los procesadores se vuelven más pequeños, rápidos y económicos. La velocidad de la luz, sin embargo, permanece sin cambios. La densidad de pines y la densidad del cableado que gobiernan las interconexiones entre los componentes del sistema están escalando a una velocidad menor que los componentes mismos. Además, la frecuencia de comunicación entre los componentes va mucho más allá de la velocidad de reloj de los procesadores modernos. Estos factores se combinan para hacer de la interconexión el factor clave en el éxito de los futuros sistemas digitales.
A medida que los diseñadores se esfuerzan por hacer un uso más eficiente del escaso ancho de banda de interconexión, las redes de interconexión se están convirtiendo en una solución casi universal para los problemas de comunicación a nivel de sistema para los sistemas digitales modernos. Originalmente desarrolladas para los exigentes requisitos de comunicación de las multicomputadoras, las redes de interconexión están comenzando a reemplazar a los buses como la interconexión estándar a nivel de sistema. También están reemplazando el cableado dedicado en sistemas de propósito especial a medida que los diseñadores descubren que los paquetes de enrutamiento son más rápidos y más económicos que los cables de enrutamiento.

1. **¿Qué es una red de interconexión?**

   Como se ilustra en la figura 2.3, una red de interconexión es un sistema programable que transporta datos entre terminales. La figura muestra seis terminales, T1 a T6, conectados a una red. Cuando el terminal T3

desea comunicar algunos datos con el terminal T5, envía un mensaje que contiene los datos a la red y la red entrega el mensaje a T5. La red es programable en el sentido de que realiza diferentes conexiones en diferentes momentos. La red en la figura puede entregar un mensaje de T3 a T5 en un ciclo y luego usar los mismos recursos para entregar un mensaje de T3 a T1 en el siguiente ciclo. La red es un sistema porque está compuesta de muchos componentes: memorias intermedias, canales, conmutadores y controles que trabajan juntos para entregar los datos.

Las redes que cumplen con esta definición amplia se producen a muchas escalas. Las redes en chip (del ingles NOC: Network on Chip) pueden entregar datos entre memoria, registros y unidades aritméticas dentro de un único procesador. Las redes de nivel de placa y nivel de sistema vinculan los procesadores a las memorias o los puertos de entrada a los puertos de salida. Finalmente, las redes de área local y de área amplia conectan sistemas dispares en una empresa o en todo el mundo. En este trabajo, restringimos nuestra atención a las escalas más pequeñas: desde el nivel de chip al nivel de sistema.

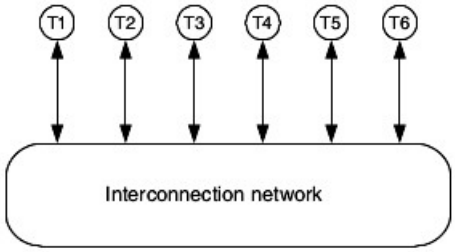


Figura 2.3: Vista funcional de una red de interconexión. [27]

En la figura 2.3 los terminales (etiquetados T1 a T6) están conectados a la red mediante canales. Las puntas de flecha en cada extremo del canal indican que es bidireccional y admite el movimiento de datos tanto dentro como fuera de la red de interconexión

El diseñador de la red debe trabajar dentro de las limitaciones tecnológicas para implementar la topología, el enrutamiento y el control de flujo de la red. Una clave para la eficiencia de las redes de interconexión proviene del hecho de que los recursos de comunicación se comparten. En lugar de crear un canal dedicado entre cada par de terminales, la red de interconexión se implementa con una colección de

nodos enrutadores compartidos conectados por canales compartidos. El patrón de conexión de estos nodos define la topología de la red. Luego se entrega un mensaje entre terminales haciendo varios saltos a través de los canales y nodos compartidos desde su terminal de origen hasta su terminal de destino. Una buena topología explota las propiedades de la tecnología de empaquetado de la red, como la cantidad de pines en el paquete de un chip o la cantidad de cables que se pueden conectar entre gabinetes separados, para maximizar el ancho de banda de la red.

Una vez que se ha elegido una topología, puede haber muchas rutas posibles (secuencias de nodos y canales) que un mensaje podría tomar a través de la red para llegar a su destino. El enrutamiento determina cuál de estas posibles rutas toma realmente un mensaje. Una buena elección de rutas minimiza su longitud, generalmente medida como el número de nodos o canales visitados, al tiempo que equilibra la demanda colocada en los recursos compartidos de la red. La longitud de una ruta obviamente influye en la latencia de un mensaje a través de la red, y la demanda o carga de un recurso es una medida de la frecuencia con la que se utiliza ese recurso. Si un recurso se sobre-utiliza mientras otro permanece inactivo, conocido como desequilibrio de carga, se reduce el ancho de banda total de los mensajes que entrega la red.

El control de flujo dicta qué mensajes obtienen acceso a recursos de red particulares a lo largo del tiempo. Esta influencia del control de flujo se vuelve más crítica a medida que aumenta la utilización de recursos y un buen control de flujo reenvía los paquetes con un retraso mínimo y evita los recursos inactivos bajo cargas elevadas.

2. **¿Dónde encontramos redes de interconexión?**

Se utilizan en casi todos los sistemas digitales que son lo suficientemente grandes como para tener dos componentes para conectar. Las aplicaciones más comunes de las redes de interconexión se encuentran en sistemas informáticos y conmutadores de comunicación. En los sistemas informáticos, conectan los procesadores a las memorias y los dispositivos de entrada/salida (E/S) a los controladores de E/S. En conmutadores de comunicación y enrutadores de red conectan puertos de entrada a puertos de salida. También conectan sensores y actuadores a procesadores en sistemas de control. En cualquier lugar donde los bits se transporten entre dos componentes de un sistema, es probable de encontrar una red de interconexión.

Hacia fines de la década de 1980, la mayoría de estas aplicaciones eran

atendidas por una red de interconexión muy simple: el bus multipunto. Hoy, sin embargo, todas las interconexiones de alto rendimiento se realizan mediante redes de interconexión punto a punto en lugar de buses, sistemas que históricamente han estado basados en buses cambian a redes cada año. Esta tendencia se debe a una escala de rendimiento no uniforme. La demanda de rendimiento de interconexión está aumentando con el rendimiento del procesador (a una tasa del 50 % por año) y el ancho de banda de la red. Los cables, por otro lado, no se vuelven más rápidos. La velocidad de la luz y la atenuación de un cable de cobre de calibre 24 no mejoran con una mejor tecnología de semiconductores. Como resultado, los buses no han podido mantenerse al día con la demanda de ancho de banda, y las redes de interconexión punto a punto, que operan más rápido que los buses y ofrecen concurrencia, están asumiendo rápidamente el control.

3. **¿Por qué son importantes las redes de interconexión?**

   Porque son un factor limitante en el rendimiento de muchos sistemas. La red de interconexión entre el procesador y la memoria determina en gran medida la latencia de la memoria y el ancho de banda de la memoria, dos factores clave del rendimiento en un sistema informático[1]. El rendimiento de la red de interconexión (a veces llamada la "*estructura*") en un conmutador de comunicación determina en gran medida la capacidad (velocidad de datos y número de puertos) del conmutador. Debido a que la demanda de interconexión ha crecido más rápidamente que la capacidad de los cables subyacentes, la interconexión se ha convertido en un cuello de botella crítico en la mayoría de los sistemas.

2.4.1. Uso de las redes de interconexión

Para comprender los requisitos establecidos en el diseño de redes de interconexión, es útil examinar cómo se utilizan en los sistemas digitales. Específicamente, la aplicación determina los siguientes parámetros de red:

- El número de terminales.
- El ancho de banda máximo de cada terminal.
- El ancho de banda promedio de cada terminal.

[1] Esto es particularmente cierto cuando se tiene en cuenta que la mayor parte del tiempo de retraso de la comunicación es durante el acceso de un chip moderno a memoria.

- La latencia requerida.
- El tamaño del mensaje o una distribución de los tamaños del mensaje.
- Los patrones de tráfico esperados.
- La calidad de servicio requerida.
- La fiabilidad y disponibilidad requeridas de la red de interconexión.

La cantidad de terminales, o puertos, en una red corresponde a la cantidad de componentes que deben conectarse a la red. Además de conocer el número de terminales, el diseñador también necesita saber cómo interactuarán los terminales con la red.
Cada terminal requerirá una cierta cantidad de ancho de banda de la red, generalmente expresada en bits por segundo (bps). A menos que se indique lo contrario, asumimos que los anchos de banda de cada terminal son simétricos, es decir, los anchos de banda de entrada y salida del terminal son iguales. El ancho de banda máximo es la velocidad de datos máxima que un terminal solicitará de la red durante un corto período de tiempo, mientras que el ancho de banda promedio es la velocidad de datos promedio que requerirá un terminal. Por ejemplo en el diseño de interconexiones de memoria-procesador, conocer los anchos de banda pico y promedio se vuelve importante cuando se trata de minimizar el costo de implementación de la red de interconexión. Además de la velocidad a la cual los mensajes deben ser aceptados y entregados por la red, el tiempo requerido para entregar un mensaje individual, la latencia del mensaje, también se especifica para la red. Si bien una red ideal admite ancho de banda alto y baja latencia, a menudo existe una compensación entre estos dos parámetros. Por ejemplo, una red que admite un ancho de banda alto tiende a mantener ocupados los recursos de la red, lo que a menudo causa contención para los recursos. La contención ocurre cuando dos o más mensajes desean usar el mismo recurso compartido en la red. Todos menos uno de estos mensajes tendrán que esperar a que ese recurso se libere, aumentando así la latencia de los mensajes. Si, en cambio, la utilización de los recursos se redujera al reducir las demandas de ancho de banda, la latencia también se reduciría.
El tamaño del mensaje, la longitud de un mensaje en bits, es otra consideración importante del diseño. Si los mensajes son pequeños, los gastos generales en la red pueden tener un mayor impacto en el rendimiento que en el caso en que los gastos generales se pueden amortizar a lo largo de un mensaje más grande. En muchos sistemas, hay varios tamaños de mensaje posibles.
La forma en que los mensajes de cada terminal se distribuyen entre todas las terminales de destino posibles define el patrón de tráfico de una red. Por

ejemplo, cada terminal puede enviar mensajes a todos los demás terminales con la misma probabilidad. Este es el patrón de tráfico aleatorio. Si, en cambio, los terminales tienden a enviar mensajes solo a otros terminales cercanos, la red subyacente puede explotar esta localidad espacial para reducir los costos. En otras redes, sin embargo, es importante que las especificaciones se mantengan para patrones de tráfico arbitrarios.
Algunas redes también requerirán calidad de servicio (QoS). En términos generales, QoS implica la asignación equitativa de recursos bajo alguna política de servicio. Por ejemplo, cuando varios mensajes compiten por el mismo recurso en la red, esta disputa se puede resolver de muchas maneras. Los mensajes se pueden servir en orden de llegada, según el tiempo que hayan estado esperando el recurso en cuestión. Otro enfoque da prioridad al mensaje que ha estado en la red durante más tiempo. La elección entre estas y otras políticas de asignación se basa en los servicios requeridos de la red.
Finalmente, la confiabilidad y disponibilidad requeridas de una red de interconexión influyen en las decisiones de diseño. La confiabilidad es una medida de la frecuencia con la que la red realiza correctamente la tarea de entregar mensajes. En la mayoría de las situaciones, los mensajes deben entregarse el 100 % del tiempo sin pérdida. Se puede lograr una red 100 % confiable agregando hardware especializado para detectar y corregir errores, un protocolo de software de nivel superior o usando una combinación de estos enfoques. También es posible que la red descarte una pequeña fracción de mensajes, la disponibilidad de una red es la fracción de tiempo que está disponible y funciona correctamente. En un enrutador de Internet, generalmente se especifica una disponibilidad del 99.999 %, menos de cinco minutos de tiempo de inactividad total por año. El desafío de proporcionar este nivel de disponibilidad es que los componentes utilizados para implementar la red a menudo fallarán varias veces por minuto. Como resultado, la red debe estar diseñada para detectar y recuperarse rápidamente de estas fallas mientras continúa operando.

2.5. Fallas en los subsistemas digitales

La predicción de confiabilidad en sistemas digitales expuestos a perturbaciones de eventos únicos (por sus siglas en ingles: Single Event Upset - SEU) es un aspecto central en el desarrollo de computadoras tolerantes a fallas dedicadas a operar en entornos de radiación (por ejemplo, en el espacio, plantas de energía nuclear, etc) y con la miniaturización, se convierte en una preocupación para las aplicaciones en la atmósfera terrestre [24].
Se puede obtener una sección transversal estática del dispositivo, (σ_D), me-

diante pruebas de radiación estática en tierra. Básicamente, la determinación de σ_D para un registro de memoria consiste en contar el número de cambios de bits que tienen lugar mientras el registro se expone a un flujo de partículas. Si el registro es un elemento interno del microprocesador, la determinación de σ_D es más complicada.
Sin embargo, existen métodos que se han utilizado para obtener σ_D para los elementos de memoria del procesador interno. Además, existen métodos para definir y obtener la sección transversal estática σ_D para todo el procesador. Dado que una aplicación que se ejecuta en el procesador usa los registros del procesador solo durante una fracción del tiempo, σ_D es una sobre estimación de la vulnerabilidad del sistema a las SEU.
Un parámetro más preciso es la sección transversal de la aplicación (σ_{AP}). La determinación de σ_{AP} requiere el uso de la llamada prueba dinámica, que consiste en exponer el dispositivo en estudio a un flujo de partículas mientras el procesador ejecuta un programa lo más cercano posible a la aplicación final. Se debe utilizar una plataforma de prueba adecuada para implementar tales pruebas y contar en línea el número de errores observados durante el experimento de irradiación. El principal problema de este tipo de prueba es que los resultados dependen en gran medida de la aplicación particular que se ejecuta en el procesador. Por lo tanto, cualquier cambio menor realizado en el programa implica la necesidad de una nueva prueba dinámica, lo que resulta en un proceso de costo muy alto.
Para abordar este problema, recientemente se presentó un enfoque de tres pasos. El primer paso consiste en medir la sección transversal estática (σ_D) mediante pruebas de radiación estática en tierra. Tenga en cuenta que σ_D es una característica del procesador en estudio y no depende de la aplicación. El segundo paso utiliza experimentos de inyección de fallas para obtener la tasa promedio de mal funcionamiento del programa por perturbaciones, llamada (τ) a continuación. La determinación de τ consiste en inyectar cambios de bits con un esquema de hardware y software mientras el procesador ejecuta la aplicación. Se debe utilizar una plataforma de prueba dedicada para generar cambios de bits y contar en línea el número de errores observados. Sin embargo, tenga en cuenta que τ se obtiene sin irradiación, lo que resulta en un proceso de bajo costo. Esto permite el estudio de los efectos causados por cambios menores en el programa. En el tercer paso, la sección transversal de la aplicación, σ_{AP}, se estima como el producto $\sigma_{AP} = \sigma_D * \tau$. Por lo tanto, está claro que, el enfoque presentado evita la necesidad de realizar pruebas dinámicas de radiación en tierra. Sin embargo, el único indicador de confiabilidad obtenido con el método anterior es σ_{AP} y este parámetro no proporciona, de manera directa, la confiabilidad del sistema.
Por definición, la confiabilidad del sistema en el tiempo t es la probabilidad

de que el sistema funcione de acuerdo con las especificaciones hasta el tiempo t. La estimación de la confiabilidad de un sistema informático tolerante a fallas (por sus siglas en inglés - FTCS) perturbado por alteraciones de eventos únicos (SEU) requiere obtener primero las funciones de distribución de probabilidad para las variables aleatorias de tiempo de recuperación (TTR) y tiempo de falla (TTF). Para calcular la confiabilidad del sistema, se debe obtener la función de distribución acumulativa TTF. En distintos trabajos, como los citados [23, 24] se muestra que estas funciones se corresponden con variables aleatorias cuya distribución es exponencial.
Por consiguiente es valido pensar que las fallas de subsistemas digitales están representadas por una maquina de estado finita, figura 2.4, de dos estados ***Down*** y ***Up***, entre los que alterna.
El tiempo de permanencia en un estado determinado ya sea Down o Up, está regida por una distribución exponencial cuya variable aleatoria es el tiempo, esto por otra parte lo convierte en un sistema no determinista.
Recordando el significado de una distribución exponencial, tenemos que el estado Up estará caracterizado por un tiempo medio de permanencia en dicho estado dado por $1/\lambda_{up}$ mientras que el estado Down poseerá un tiempo medio de permanencia dado por $1/\lambda_{down}$.

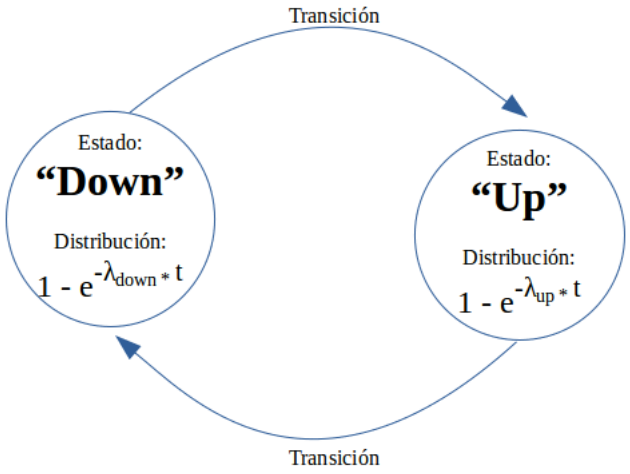


Figura 2.4: Maquina de estado finita. Dos estados “*Down*” y “*Up*”.

En éste trabajo, calculamos (capítulo 7) métricas para la evaluación de las redes simuladas (mencionadas anteriormente), la demora de extremo a extremo y la cantidad de saltos de cada mensaje en función de la relación $(1/\lambda_{up})/(1/\lambda_{down})$, es decir evaluamos las redes en función de los tiempos medios de actividad/inactividad de la falla del subsistema digital.

Capítulo 3

Origen de las redes DTN

La mayoría de los protocolos de transporte, como TCP se basan en la suposición de que el remitente y el receptor están conectados continuamente por alguna ruta de transmisión, si el protocolo falla y los datos no pueden ser entregados el mensaje debe ser reenviado. En algunas redes, no existe una ruta de extremo a extremo. Un ejemplo es una red espacial como LEO [19] (órbita baja de la Tierra) los satélites pasan dentro y fuera del alcance de las estaciones terrestres. Un satélite dado puede ser capaz de comunicarse con una estación de tierra sólo en determinados momentos, y dos satélites nunca pueden ser capaces de comunicarse entre sí en cualquier momento, incluso a través de una estación ubicada en tierra, la razón es que alguno de estos puede encontrarse fuera de rango es decir está apuntando hacia otro lugar. Otro tipo de redes son conexiones para: submarinos, autobuses, teléfonos móviles y otros dispositivos donde los equipos no se conectan de forma constante debido a la movilidad o las condiciones extremas. En este tipo de redes que se conectan ocasionalmente, los datos aún pueden ser comunicados mediante el almacenamiento en los nodos y reenviados más tarde, cuando haya un enlace disponible. Luego finalmente, los datos se transmiten al destino. Una red cuya arquitectura se basa en este enfoque se denomina DTN [13, 16, 28] (Delay Tolerant Network , o redes tolerantes a fallas o retardos).

3.1. Arquitectura y protocolo DTN

Las características principales para la arquitectura de las DTNs, está definida por la RFC 4838 (RFC: Request For Comments). Dicha arquitectura busca hacer frente al problema de comunicación en entornos difíciles a través del almacenamiento y reenvío de mensajes que aprovechan un conjunto de capas de red convergentes que se adaptan a una amplia variedad de transportes

subyacentes. Así, las tecnologías inalámbricas para DTN pueden llegar a ser diversas, incluyendo no sólo la de radio frecuencia (RF), sino también banda ultra-ancha (UWB), medios ópticos y las tecnologías de acústica (sonar o ultrasonidos).
En la arquitectura normalizada se especifican aspectos tales como los tipos de nodos y enlaces (contactos) que pueden existir en una DTN, el principio básico de operación basado en encaminamiento y reenvío, implementado dentro de la capa Bundle (normalizada en la RFC 5050).

3.1.1. Dominio interplanetario

Por un lado, el internet interplanetario [13] es un esfuerzo continuo de investigación y desarrollo que tiene como objetivo conectar nodos en el espacio a distancias interplanetarias permitiéndoles comunicarse aparentemente automáticamente entre sí. Esta comunicación automática y confiable requiere el uso de protocolos específicos que pueden tolerar demoras o interrupciones muy largas causadas por largas distancias y obstáculos en los enlaces inalámbricos. De hecho, los protocolos estándar de Internet como TCP/IP que funcionan en el contexto actual de la Tierra deben modificarse antes de usarse en el escenario interplanetario.
Como resultado, el Grupo de trabajo de ingeniería de Internet *(IETF)* ha desarrollado y estandarizado un nuevo conjunto de protocolos adecuados para este escenario [29] y el Comité Consultivo para Sistemas de Datos Espaciales (CCSDS) [30].

3.1.2. Dominio cercano a la tierra

Por otro lado, los protocolos DTN también se han considerado como la infraestructura de comunicación de referencia para los grupos de satélites de órbita baja. Estos *clusters* están diseñados para realizar un trabajo colaborativo intercambiando continuamente datos, comandos y estado.
Estos *clusters* funcionan en colaboración y necesitan las características de los protocolos DTN para superar las interrupciones de enlace provocadas por la dinámica orbital o el ciclo de trabajo altamente limitado de los transpondedores a bordo que consumen mucha energía [22, 31–33].
Estos desarrollos de protocolos DTN en el contexto de satélites en órbita cercana a la Tierra están estrechamente relacionados con un nuevo paradigma de desarrollos espaciales llamados arquitecturas fragmentadas o segmentadas.

3.1.3. Método de almacenamiento y reenvío

Cuando tenemos un enlace entre dos nodos que se comunican y están al mismo tiempo en movimiento, podemos encontrarnos con el inconveniente de que la conexión entre ambos pueda verse obstruida, ocasionando que la comunicación se inhabilite con el fin de ahorrar energía hasta que el enlace vuelva a estar disponible; esto ocasiona que la conectividad sea intermitente. En Internet, la conectividad intermitente provoca la pérdida de datos. Los paquetes que no pueden ser inmediatamente transmitidos son normalmente desechados y TCP debe volver a retransmitirlos.
En las DTNs, por el contrario, se soporta la comunicación entre nodos con conexiones intermitentes, mediante una técnica clave para estas redes como lo es la del almacenamiento, resguardo y reenvío de datos, con la cual se hace frente al problema de la intermitencia en la conectividad, los retardos variables y tasas de datos asimétricas mediante el uso del método de conmutación de datos a través del almacenamiento, resguardo y reenvío, tales como los utilizados por el correo electrónico o el correo de voz. Al contrario del modelo de Internet (en el cual lo routers utilizan chips de memoria para almacenar los paquetes entrantes durante unos pocos milisegundos mientras esperan por su búsqueda del siguiente salto en la tabla de enrutamiento y un puerto disponible del router de salida), en las DTNs para que el método de almacenamiento, resguardo y reenvío de datos sea posible es necesario que los lugares de acopio de la información (como un disco duro) puedan almacenar los mensajes de forma permanente. Estos son llamados "almacenadores persistentes" (el "store" en la figura 3.1).

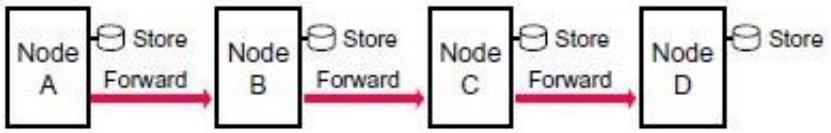


Figura 3.1: Almacenamiento y reenvío de datos empleado por los nodos en las DTNs. [28]

Los routers que se usan en las DTNs necesitan un almacenamiento persistente para encolar sus datos, por las siguientes razones:

1. La vía de comunicación con el siguiente salto pueden no estar disponible durante un largo periodo de tiempo.

2. Un nodo puede enviar o recibir datos mucho más rápidamente o de forma más fiable que el otro nodo.

3. Una vez transmitidos los datos puede necesitar ser retransmitidos si se produce un error en el enlace de transmisión o si un nodo declina hacer el reenvío del mensaje.

Al mover mensajes enteros (o fragmentos de los mismos) en una única transferencia, la técnica de conmutación de mensajes proporciona a los nodos de la red un conocimiento inmediato del tamaño de los mensajes que se transfieren, y por lo tanto los requisitos para el almacenamiento de la información y el ancho de banda necesario. Dado que la constante de este tipo de redes es la movilidad debido a que los nodos suelen estar en movimiento con alimentación limitada, es fundamental que en su arquitectura se contemple el almacenamiento de datos ya que de alguna manera los datos deben resguardarse.

3.2. La capa Bundle

El Bundle es la unidad básica de datos para el intercambio de información para una red DTN. Un nodo DTN puede ser un Host, Router o Gateway que actúa como origen, destino, en una red DTN, utilizando un conjunto común de capas de red (tales como TCP e IP). La Blunde Layer esta situada por debajo de la capa de aplicación como se observa en la figura 3.2, la información en un bundle, es muy similar a un mensaje de correo electrónico, este es luego transmitido a lo largo de una ruta que consta de una serie de máquinas intermedias que pueden cada una almacenar el paquete durante períodos significativos. Por lo tanto, la capa bundle es una red de almacenamiento y reenvío. Incluye una serie de funciones de diagnóstico y gestión.
Para la interoperabilidad, utiliza un esquema de nombres flexible (sobre la base de Identificadores Uniformes de Recursos) capaces de encapsular diferentes esquemas de nombres y números en la misma sintaxis de nomenclatura general. El protocolo bundle se describe en una serie de documentos RFCs. El primero ya mencionado es el RFC 4838 [16], en la cual se expone la arquitectura de la red DTN en términos de la estructura y aplicabilidad. El otro documento es el RFC 5050 [29], el cual especifica el formato de los bundles y las reglas de procesamiento asociados con el envío y la recepción de los mismos. La característica mas importante de la capa bundle, es el soporte para el almacenamiento en tránsito, es decir los paquetes recibidos de un remitente puede ser almacenados en un nodo intermedio una cantidad de tiempo excesiva (minutos, horas o incluso días). Estas operaciones son almacenadas

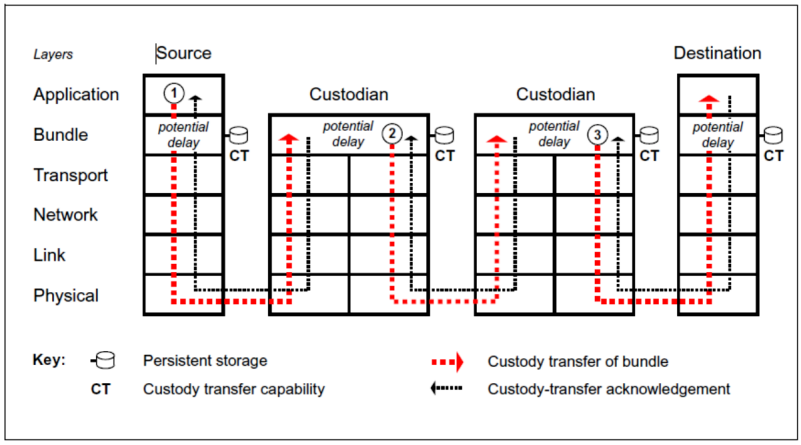


Figura 3.2: Protocolo Bundle. [28]

por el stack de protocolos de la red, en la capa bundle, de forma transparente para la aplicación. El almacenamiento en tránsito es el medio para superar los retrasos y trastornos inducidos, mientras que un bundle realiza saltos hasta llegar a su destino final, para evitar costosas retransmisiones de extremo a extremo debido a los errores y tiempos de espera; y para permitir el intercambio de información entre dos nodos que no comparten ninguna trayectoria de comunicación de extremo a extremo en un momento dado en el tiempo.

Un elemento esencial del bundle basado en el reenvío es que tienen un lugar para esperar en cola hasta que haya una oportunidad de comunicación disponible, esto supone que:

- El almacenamiento está disponible y bien distribuido por toda la red.
- El almacenamiento es suficientemente persistente y robusto para almacenar hasta que el reenvío de paquetes puede ocurrir.

Además los lugares de almacenamiento (como el disco duro) puede almacenar mensajes de forma indefinida, a esto se les llama almacenamiento persistente, a diferencia del almacenamiento a muy corto plazo proporcionada por chips de memoria, como el que utilizan los routers de Internet, las rutas DTN necesitan un almacenamiento persistente para sus colas ya que:

- El enlace de comunicación con el siguiente salto puede o no estar disponible durante mucho tiempo.
- Puede que en un par de nodos exista una sincronización para el envió.

- Un mensaje una vez enviado puede necesitar ser retransmitido si llega ocurrir algún error.

3.2.1. Protocolos de enrutamiento

El problema de enrutamiento en redes DTN es quizás uno de los mayores temas de investigación, la razón es, que no hay un método exacto o preferido para manejar los problemas de enrutamiento [34]. En primer lugar las redes convencionales de internet suponen que la topología de red esta siempre conectada (o particionada para intervalos muy cortos), el objetivo de los algoritmos de enrutamiento es encontrar la mejor ruta disponible para mover el tráfico de extremo a extremo. En una red DTN este principio no se aplica ya que las rutas de extremo a extremo no están siempre disponibles en todo momento, el enrutamiento que se realiza es para alcanzar lograr la entrega del mensaje mediante el empleo a largo plazo del almacenamiento en los nodos intermedios. Para las redes DTN un objetivo importante es aumentar la probabilidad de la entrega del paquete, además de la reducción del retardo en la entrega también suele ser importante para las aplicaciones. La gestión del almacenamiento también está relacionado con el enrutamiento, las redes DTN poseen esquemas de encaminamiento que tienen que manejar problemas de que los nodos no están conectados constantemente.
La Clasificación de los algoritmos de enrutamiento DTN es algo complejo. En [34] se define una estructura de algoritmos de encaminamiento dependiendo de la cantidad de conocimiento que utilizan para calcular las rutas: algoritmos que no utilizan ninguna información de conocimiento son llamados “de conocimiento cero”; los que utilizan información parcial para tomar decisiones llamados “de conocimiento parcial”, y los “de conocimiento completo” aquellos que tienen toda la información para el enrutamiento de los bundles.

Capítulo 4

El simulador OMNeT++

OMNeT++ es un marco de simulación de redes modulares de eventos discretos orientado a objetos [35]. Tiene una arquitectura genérica, por lo que puede usarse y ha sido utilizada en varios dominios problemáticos:

- Modelado de redes de comunicación cableadas e inalámbricas.
- Modelado de protocolos.
- Modelado de redes de colas.
- Modelado de multiprocesadores y otros sistemas de hardware distribuido.
- Validación de arquitecturas de hardware.
- Evaluar aspectos de rendimiento de sistemas de software complejos.
- En general, el modelado y la simulación de cualquier sistema en el que el enfoque de eventos discretos sea adecuado y pueda asignarse convenientemente a entidades que se comunican mediante el intercambio de mensajes.

OMNeT++ es la herramienta elegida en este trabajo para simular las redes y efectos de fallas en subsistemas digitales.

4.1. Introducción

OMNeT++ en sí mismo no es un simulador de nada concreto, sino que proporciona infraestructura y herramientas para escribir simulaciones. Uno de los ingredientes fundamentales de esta infraestructura es una arquitectura de

componentes para modelos de simulación. Los modelos se ensamblan a partir de componentes reutilizables denominados "*módulos*". Los módulos bien escritos son realmente reutilizables y se pueden combinar de varias maneras, como los bloques LEGO®.
Los módulos se pueden conectar entre sí a través de puertas (otros sistemas los llamarían puertos) y combinarlos para formar "*módulos compuestos*". La profundidad de anidamiento del módulo no está limitada. Los módulos se comunican a través del paso de mensajes, donde los mensajes pueden ser estructuras de datos arbitrarias. Los módulos pueden pasar mensajes a lo largo de rutas predefinidas a través de puertas y conexiones, o directamente a su destino; este último es útil para simulaciones inalámbricas, por ejemplo. Los módulos pueden tener parámetros que pueden usarse para personalizar el comportamiento del módulo y/o para parametrizar la topología del modelo. Los módulos en el nivel más bajo de la jerarquía de módulos se denominan "*módulos simples*" y encapsulan el comportamiento del modelo. Los módulos simples se programan en C++ y utilizan la biblioteca de simulación.
Las simulaciones de OMNeT++ se pueden ejecutar en varias interfaces de usuario. Las interfaces de usuario gráficas y animadas son muy útiles para fines de demostración y depuración, y las interfaces de usuario de línea de comandos son las mejores para la ejecución por lotes.
El simulador, así como las interfaces y herramientas de usuario son altamente portátiles. Se ejecutan en los sistemas operativos más comunes (Linux, Mac OS/X y Windows), y se pueden compilar fuera del framework.
OMNeT++ también admite simulaciones distribuidas paralelas. OMNeT++ puede usar varios mecanismos para la comunicación entre particiones de una simulación distribuida paralela, por ejemplo MPI o pipes con nombre. El algoritmo de simulación en paralelo se puede ampliar fácilmente o se pueden conectar otros nuevos. Los modelos no necesitan ninguna instrumentación especial para ejecutarse en paralelo, es solo una cuestión de configuración.
OMNEST es la versión comercialmente compatible de OMNeT++. OMNeT++ es gratis solo para uso académico y sin fines de lucro; para fines comerciales, uno necesita obtener licencias OMNEST de **Simulcraft Inc**.

4.1.1. Historia de OMNeT++

Los inicios del desarrollo de OMNeT++ (Objective Modular Network Testbench in C++) transcurren en 1992 gracias a András Varga que empezó a desarrollar este simulador a partir de OMNeT++ escrito en pascal y desarrollado por su profesor Dr. Gyrgy Pongor en la Universidad Técnica de Budapest. Con el tiempo se ha convertido en una herramienta muy popular en la comunidad científica y en el mundo industrial. Esta herramienta cuenta con

contribuciones de un considerable número de personas. Hoy en día muchas de las universidades apuestan por el uso y desarrollo de esta herramienta ya que su licencia académica es pública. Por otro lado las grandes multinacionales del sector de las telecomunicaciones tales como **Cisco**®, **Alcatel-lucent**®, **Orange**®, **IBM**®, **Intel**® o **HP**® adquieren su versión comercial, desarrollada actualmente por Simulcraft Inc y para cuya utilización es necesario obtener licencias de la Omnest Global, Inc.

4.2. Conceptos de modelado

Un modelo OMNeT++ consta de módulos que se comunican con el paso de mensajes. Los módulos activos se denominan "*módulos simples*"; están escritos en C++, usando la biblioteca de clases de simulación. Los módulos simples se pueden agrupar en "*módulos compuestos*", etc. El número de niveles jerárquicos es ilimitado. Todo el modelo, llamado "*red*" en OMNeT++, es en sí mismo un módulo compuesto. Los mensajes se pueden enviar a través de conexiones que abarcan módulos simples o directamente a otros módulos. En la Figura 4.1, los cuadros (más pequeños) representan módulos simples y los demás son módulos compuestos. Las flechas que conectan cajas pequeñas representan conexiones y puertas.

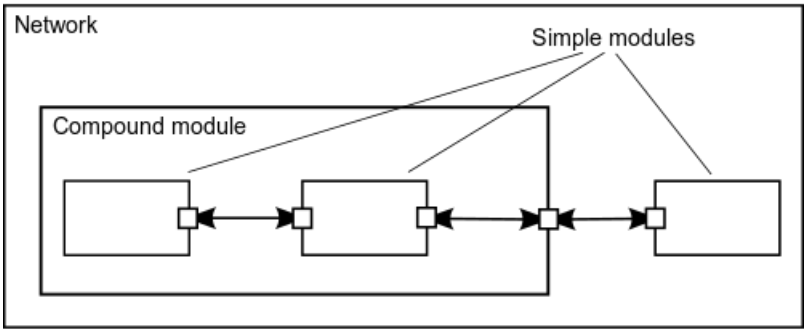


Figura 4.1: Módulos simple y compuestos. [35]

Los módulos se comunican con mensajes que pueden contener datos arbitrarios, además de los atributos habituales, como una marca de tiempo. Los módulos simples generalmente envían mensajes a través de puertas, pero también es posible enviarlos directamente a sus módulos de destino. Las puertas son las interfaces de entrada y salida de los módulos: los mensajes se envían a través de las puertas de salida y llegan a través de las puertas de entrada. Una puerta de entrada y una puerta de salida pueden vincularse mediante

una conexión. Las conexiones se crean dentro de un solo nivel de jerarquía de módulos; dentro de un módulo compuesto, se pueden conectar las puertas correspondientes de dos submódulos, o una puerta de un submódulo y una puerta del módulo compuesto.
No se permiten conexiones que abarquen distintos niveles de jerarquía, ya que dificultarían la reutilización del modelo. Debido a la estructura jerárquica del modelo, los mensajes generalmente viajan a través de una cadena de conexiones, comenzando y llegando a módulos simples. Los módulos compuestos actúan como "cajas de cartón" en el modelo, transmitiendo mensajes de manera transparente entre su mundo interior y el mundo exterior. Se pueden asignar parámetros como retraso de propagación, velocidad de datos y velocidad de error de bit a las conexiones. También se pueden definir tipos de conexión con propiedades específicas (denominados canales) y reutilizarlos en varios lugares.
Los módulos pueden tener parámetros. Los parámetros se utilizan principalmente para pasar datos de configuración a módulos simples y para ayudar a definir la topología del modelo. Los parámetros pueden tomar valores de cadena ("string"), numéricos o booleanos. Debido a que los parámetros se representan como objetos en el programa, los parámetros, además de mantener constantes, pueden actuar de manera transparente como fuentes de números aleatorios, con las distribuciones reales proporcionadas con la configuración del modelo. Pueden solicitar interactivamente al usuario el valor, y también pueden contener expresiones que hacen referencia a otros parámetros. Los módulos compuestos pueden pasar parámetros o expresiones de parámetros a sus submódulos.
OMNeT++ proporciona herramientas eficientes para que el usuario describa la estructura del sistema real.

Algunas de las características principales son las siguientes:

- Módulos jerárquicamente anidados.
- Los módulos son instancias de tipos de módulos.
- Los módulos se comunican con mensajes a través de canales.
- Parámetros flexibles del módulo.
- Lenguaje de descripción de topología.

4.2.1. Módulos Jerárquicos

Un modelo OMNeT++ consta de módulos jerárquicamente anidados que se comunican pasando mensajes entre sí. Los modelos OMNeT++ se denominan

"*redes*". El módulo de nivel superior es el módulo del sistema. El módulo del sistema contiene submódulos que también pueden contener submódulos (Figura 4.1). La profundidad de anidamiento del módulo es ilimitada, lo que permite al usuario reflejar la estructura lógica del sistema real en la estructura del modelo.
La estructura del modelo se describe en el lenguaje NED de OMNeT++.
Los módulos que contienen submódulos se denominan "*módulos compuestos*", a diferencia de los "*módulos simples*" en el nivel más bajo de la jerarquía de módulos. Los módulos simples contienen los algoritmos del modelo. El usuario implementa los módulos simples en C++, utilizando la biblioteca de clase de simulación OMNeT++.

4.2.2. Tipos de módulos

Tanto los módulos simples como los compuestos son instancias del tipo "module". Al describir el modelo, el usuario define los tipos de módulos; Las instancias de estos tipos de módulos sirven como componentes para tipos de módulos más complejos. Finalmente, el usuario crea el módulo del sistema como una instancia de un tipo de módulo definido previamente; todos los módulos de la red se instancian como submódulos y sub-submódulos del módulo del sistema.
Los tipos de módulos se pueden almacenar en archivos por separado del lugar de su uso real. Esto significa que el usuario puede agrupar los tipos de módulos existentes y crear bibliotecas de componentes.

4.3. Mensajes, puertas y enlaces

Los módulos se comunican intercambiando mensajes. En una simulación real, los mensajes pueden representar tramas o paquetes en una red informática, trabajos o clientes en una red de colas u otros tipos de entidades móviles. Los mensajes pueden contener estructuras de datos arbitrariamente complejas. Los módulos simples pueden enviar mensajes directamente a su destino o a lo largo de una ruta predefinida, a través de puertas y conexiones.
El "tiempo de simulación local" de un módulo avanza cuando el módulo recibe un mensaje. El mensaje puede llegar desde otro módulo o desde el mismo módulo (los auto-mensajes se utilizan para implementar temporizadores).
Las puertas son las interfaces de entrada y salida de los módulos; los mensajes se envían a través de puertas de salida y llegan a través de puertas de entrada. Cada conexión (también llamada enlace) se crea dentro de un solo nivel de la jerarquía del módulo: dentro de un módulo compuesto, uno puede conectar las

puertas correspondientes de dos submódulos, o una puerta de un submódulo y una puerta del módulo compuesto (figura 4.1).
Debido a la estructura jerárquica del modelo, los mensajes generalmente viajan a través de una serie de conexiones, comenzando y llegando a módulos simples.

4.4. Modelado de transmisiones de paquetes

Para facilitar el modelado de redes de comunicación, las conexiones se pueden usar para modelar enlaces físicos. Las conexiones admiten los siguientes parámetros: velocidad de datos, retraso de propagación, error de velocidad de bits y error de velocidad de paquete, y pueden deshabilitarse. Estos parámetros y los algoritmos subyacentes se encapsulan en objetos de tipo "*channel*" (canal). El usuario puede parametrizar los tipos de canales proporcionados por OMNeT++ y también crear nuevos.
Cuando las velocidades de datos están en uso, un objeto paquete se entrega por defecto al módulo de destino en el tiempo de simulación que corresponde al final de la recepción del paquete. Dado que este comportamiento no es adecuado para el modelado de algunos protocolos (por ejemplo, Ethernet semidúplex), OMNeT++ ofrece la posibilidad de que el módulo de destino especifique que desea que se le entregue el objeto paquete cuando comience la recepción del paquete.

4.5. Parámetros

Los módulos pueden tener parámetros. Los parámetros se pueden asignar en los archivos NED o en el archivo de configuración **omnetpp.ini**.
Los parámetros se pueden usar para personalizar el comportamiento simple del módulo y para parametrizar la topología del modelo.
Los parámetros pueden tomar valores de cadena ("string"), numéricos o booleanos, o pueden contener árboles de datos XML. Los valores numéricos incluyen expresiones que usan otros parámetros y llaman a funciones, variables aleatorias de diferentes distribuciones y valores ingresados de manera interactiva por el usuario.
Los parámetros con valores numéricos se pueden usar para construir topologías de manera flexible. Dentro de un módulo compuesto, los parámetros pueden definir el número de submódulos, el número de puertas y la forma en que se realizan las conexiones internas.

4.6. Método de descripción de la topología

El usuario define la estructura del modelo en descripciones de lenguaje NED (Descripción de red).
El simulador utiliza el lenguaje de programación NED, basado en C++, como herramienta para modelar topologías de red. Este lenguaje define la estructura de la red y facilita la descripción modular de una red. Un modelo en OMNeT++ se construye con módulos jerárquicos mediante el lenguaje NED. Dichos módulos pueden contener estructuras complejas de datos y tener sus propios parámetros usados para personalizar el envío de paquetes a los destinos a través de rutas, compuertas y conexiones. Básicamente, con el lenguaje NED se definen tres tipos de módulos: módulos simples, módulos compuestos y redes, como ya se explicó anteriormente. En los módulos de red se encuentran los componentes y especificaciones de la descripción de una red de comunicaciones.
Con el fin de facilitar el diseño de redes y la simulación de eventos sobre las mismas, OMNeT++, permite al usuario trabajar gráficamente empleando el editor del lenguaje NED (GNED). Este editor es la interfaz gráfica que permite crear, programar, configurar y simular redes de comunicaciones, sin necesidad de hacerlo utilizando la codificación del lenguaje NED; ya que automáticamente, GNED se encarga de generar el código del lenguaje, de acuerdo al diseño y configuración que realiza el usuario en forma gráfica. Además GNED, permite acceder fácilmente a dicho código una vez se crea dicha red con el GNED. Los archivos que contienen las descripciones de red deben terminar con el sufijo "*.ned*".

La estructura de un archivo NED puede contener los siguientes componentes:

- Directivas import.
- Definiciones de canales.
- Definiciones de módulos simples y complejos.
- Definiciones de red.

Las directivas import permiten importar declaraciones de otros archivos ned, funciona como la sentencia include en C++.
La definición de un canal especifica una conexión con características dadas. Su sintaxis es:

```
channel
    ChannelName
```

```
    //...

endchannel
```

Los parámetros a definir en un canal son: "*delay*", "*error*" y "*datarate*". Los módulos simples son las estructuras básicas para módulos más complejos.

4.7. Programando los algoritmos

Los módulos simples de un modelo contienen algoritmos como funciones en C++. Se puede utilizar toda la flexibilidad y potencia del lenguaje de programación, con el apoyo de la biblioteca de clase de simulación OMNeT++. El programador de simulación puede elegir entre descripciones basadas en eventos y estilo de proceso, y usar libremente conceptos orientados a objetos (herencia, polimorfismo, etc.) y patrones de diseño para ampliar la funcionalidad del simulador.
Los objetos de simulación (mensajes, módulos, colas, etc.) están representados por clases C++. Han sido diseñados para trabajar juntos de manera eficiente, creando un poderoso marco de programación de simulación. Las siguientes clases son parte de la biblioteca de clases de simulación:

- module, gate, parameter, channel (módulo, puerta, parámetro, canal).
- message, packet (mensaje, paquete).
- container classes (clases contenedores, por ejemplo, queue, array).
- clases de recolección de datos.
- clases de estimación estadística y de distribución (histogramas, algoritmo P^2 para calcular cuantiles, etc.)

Las clases también están especialmente instrumentadas, lo que permite visualizar objetos de una simulación en ejecución y mostrar información sobre ellos, como nombre, nombre de clase, variables de estado o contenido. Esta característica hace posible crear una GUI de simulación donde todos los elementos internos de la simulación son visibles.

4.8. Usando OMNeT++

4.8.1. Construyendo y ejecutando simulaciones

Brevemente proporcionamos información sobre cómo trabajar con el framework (OMNeT++) en la práctica, como trabajar con los archivos del modelo, la compilación y ejecución de simulaciones.

Un modelo OMNeT ++ consta de las siguientes partes:

- Descripción(es) de topología con el lenguaje NED ("*archivos.ned*") que describen la estructura del módulo con parámetros, compuertas, etc. Los archivos NED se pueden escribir utilizando cualquier editor de texto, pero el IDE OMNeT++ proporciona un soporte bidireccional excelente para la edición gráfica y de texto.
- Definiciones de mensajes ("*archivos.msg*") que permiten definir tipos de mensajes y agregarles campos de datos. OMNeT++ traducirá las definiciones de mensajes en clases de C++ completas.
- Fuentes de módulos simples. Son archivos C++, con sufijo .h/.cc.

El sistema de simulación proporciona los siguientes componentes:

- Kernel de simulación. Contiene el código que administra la simulación y la biblioteca de clases de simulación. Está escrito en C++, compilado en una biblioteca compartida o estática.
- Las interfaces de usuario. Las interfaces de usuario OMNeT++ se utilizan en la ejecución de simulaciones, para facilitar la depuración, la demostración o la ejecución por lotes de simulaciones. Están escritos en C++, compilados en bibliotecas.

Los programas de simulación se crean a partir de los componentes anteriores. Primero, los archivos **.msg** se traducen a código C++ usando el programa **opp_msgc**. Luego, todas las fuentes C++ se compilan y vinculan con el núcleo de simulación y una biblioteca de interfaz de usuario para formar una biblioteca compartida o ejecutable de simulación. Los archivos NED se cargan dinámicamente en sus formas de texto originales cuando se inicia el programa de simulación.

4.8.2. Ejecutando y analizando los resultados

La simulación puede compilarse como un ejecutable de programa independiente o como una biblioteca compartida para ejecutarse utilizando la utilidad **opp_run** de OMNeT++. Cuando se inicia el programa, primero lee los archivos NED, luego el archivo de configuración generalmente se llama **omnetpp.ini**. El archivo de configuración contiene configuraciones que controlan cómo se ejecuta la simulación, valores para los parámetros del modelo, etc. El archivo de configuración también puede prescribir varias ejecuciones de simulación; en el caso más simple, serán ejecutados por el programa de simulación uno tras otro.
La salida de la simulación se escribe en archivos de resultados: archivos de salida vector ("*.vec*"), archivos de salida escalares ("*.sca*") y posiblemente los propios archivos de salida del usuario. OMNeT++ contiene un entorno de desarrollo integrado (IDE) que proporciona un entorno rico para analizar estos archivos. Los archivos de salida son archivos de texto orientados a líneas que permiten procesarlos con una variedad de herramientas y lenguajes de programación, incluidos Matlab, GNU R, Perl, Python y programas de hoja de cálculo.

Capítulo 5

Estudio de casos - La red TCP

La posibilidad de tener muchos nodos de procesamiento en un solo chip exige una forma conveniente y eficiente de interconectarlos. Las arquitecturas tradicionales de multiprocesador basadas en *"buses"* sufren serias limitaciones relacionadas con la escalabilidad cuando aumenta el número de nodos.
Las arquitecturas NOC flexibles y escalables se están convirtiendo en el marco de comunicación estándar para reemplazar tales redes basadas en buses. En particular, los NOC inalámbricos tienen muchas ventajas en comparación con los NOC cableados, ya que son la solución a los problemas de muchas redes basadas en bus, como los altos costos de energía y los retrasos de propagación. En general, se propone una combinación de arquitecturas de conexiones cableadas e inalámbricas en muchos diseños de NOC [36].

5.1. Comunicaciones entre e inter-Chips

Típicamente, tres o cuatro procesadores vecinos están interconectados con cables y lógica de conmutación, formando un grupo.
Dentro del grupo, cada procesador comparte recursos con otros enlaces cableados de "corta distancia" y circuitos de conmutación lógica. Cada grupo se conecta a otros grupos a través de enlaces inalámbricos de "larga distancia". Es importante tener en cuenta que el término "larga distancia" en los escenarios NOC puede referirse a distancias en el orden de milímetros o centímetros.
La conexión inalámbrica permite no solo reemplazar las interconexiones de subsistemas basados en buses clásicos, sino también desacoplar física y lógicamentemente los módulos microelectrónicos. Además, si los enlaces de transmisión están correctamente diseñados, las comunicaciones inalámbricas entre chips también facilitarán y simplificarán el diseño y la fabricación de Placas

de Circuito Impreso (PCB) y de módulos mayores.

5.2. Análisis de los casos de estudio

Para apoyar, aclarar y visualizar mejor las ideas y discusiones anteriores, se comparan dos redes simples como un caso de estudio.
De hecho, por simplicidad, y como un análisis preliminar de las ideas propuestas, ambos tienen una topología lineal conectada a través de enlaces inalámbricos como se muestra en la figura 5.1. Las fuentes y el destino están representados por rectángulos rellenos, mientras que los nodos intermedios entre ellos están representados por rectángulos vacíos.

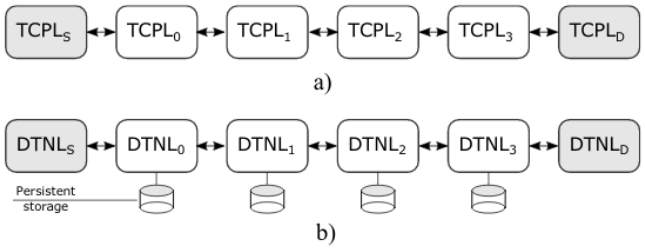


Figura 5.1: Ambas redes tienen la misma topología lineal simple conectada a través de enlaces inalámbricos. Sin embargo, a) representa una red cuyos nodos usan un protocolo basado en TCP, mientras que b) representa una red cuyos nodos usan un protocolo basado en DTN. [37]

La principal diferencia entre las dos redes es que en la topología de la figura 5.1 a), los nodos no tienen características DTN (más precisamente, la capacidad de custodia de datos), mientras que en la figura 5.1 b) los nodos implementan características DTN.
En otras palabras, se asume que la primera red implementa un protocolo del tipo TCP, mientras que la segunda implementa un protocolo del tipo DTN. Por esa razón, en la primera red, los nodos se denominan $TCPL_i$, que significa nodo tipo TCP Lineal i, con $i = 0, 1, 2, 3$. Por la misma razón, en la segunda red, los nodos se denominan $DTNL_i$, que significa nodo tipo DTN Lineal i, con $i = 0, 1, 2, 3$.
En ambos casos, el objetivo es transmitir datos desde un nodo de origen a un nodo de destino.
Como ya se mencionó, los NOC inalámbricos son un esfuerzo por reemplazar los buses por redes y resolver los problemas y limitaciones de los buses a través de las capacidades de las redes inalámbricas.

De esta manera, en la figura 5.1, las fuentes y los destinos se conectan a través de una red inalámbrica lineal en lugar de un bus.
Como consecuencia, las fuentes y el destino pueden ubicarse en otros lugares en diferentes ubicaciones físicas, en diferentes chips o una combinación de ambos. Los dispositivos de origen representan dispositivos que generan datos, como sensores de cualquier tipo (incluidos los sensores de imagen, por ejemplo) y los dispositivos de destino representan a los que reciben los datos, ya sea de forma cruda o procesada. Esto último significa que los datos se pueden transferir de un nodo a otro con o sin procesamiento en el nodo que los recibió. El tráfico desde los destinos a las fuentes (típicamente estados de los sensores o respuestas a comandos) también está permitido. En aplicaciones reales, la topología de red dependerá finalmente de los aspectos físicos y operativos y los requisitos de los sistemas que se desarrollarán bajo este paradigma tecnológico.
Con el fin de comparar la confiabilidad y disponibilidad (dependibilidad) de las dos redes (figura 5.1), se realizaron las simulaciones[1] en OMNeT++.
Vale la pena señalar que no se hicieron esfuerzos en este trabajo para lograr mejoras u optimizaciones del rendimiento de la red o del sistema, sino sólo realizar un primer análisis preliminar sobre la confiabilidad de los dos escenarios propuestos en la figura 5.1.

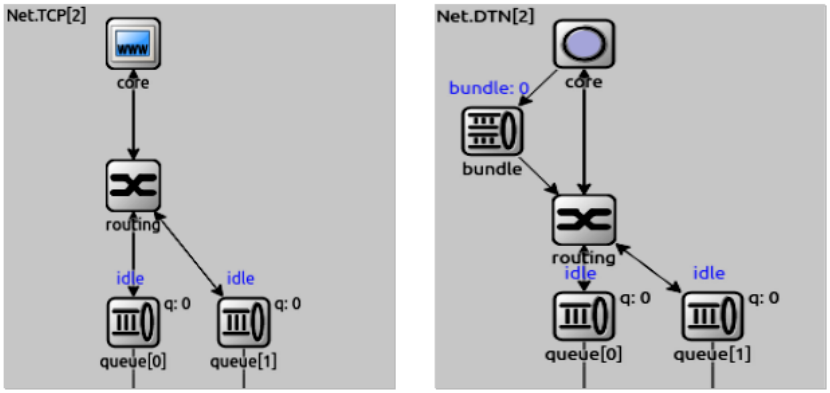
Para implementar dicha simulación, se supone a demás de la red lineal, el siguiente conjunto de hechos de simplificación para ambas redes, donde cada nodo tiene una estructura como se muestra en la figura 5.2:

1. Los nodos de las redes son módulos compuestos como se muestra en la figura 5.2, integrados por los siguientes módulos simples: colas de ingreso/egreso denominadas "*queue*" y un router denominado "*routing*" lo que conforma una estructura de red típica para el manejo de datagramas, luego un módulo simple de administración del nodo denominado "*core*" y para el caso de la red DTN se agrega un módulo simple más, denominado "*bundle*", dedicado al almacenamiento y reenvío de los datos.

2. Cada nodo tiene capacidades de reparación automática y estados distribuidos exponencialmente "*UP*" y "*DOWN*" como se describe en [24,33]. Mas precisamente solo los router son los elementos que fallarán de esta manera aleatoria.

   Es decir, están alternando continuamente entre un estado activo y uno inactivo, debido a la presencia de fallas recuperables y reparables y

[1] Todo el código para reproducir el trabajo completo se encuentra en: https://gitlab.com/danilocapkob/trabajo_especial_cs_danilocapkob



a) **b)**

Figura 5.2: a) diseño del nodo para la red TCP, b) diseño del nodo para la red DTN.

mecanismos de recuperación de fallas. Los tiempos de estadía UP y DOWN son distribuciones exponenciales con diferentes tiempos medios.

Además, para poder realizar la comparación más fácilmente, se asume que solo el router ("*routing*") del nodo 2 tiene la falla cuya distribución exponencial de tiempo de actividad está dada por la ecuación (5.1) y una distribución exponencial de tiempo de inactividad dada por la ecuación (5.2).

$$F_{UP} = 1 - exp(-\lambda_{up}t) \tag{5.1}$$

$$F_{DOWN} = 1 - exp(-\lambda_{down}t) \tag{5.2}$$

3. Se asume también que si un nodo está en estado activo, *("UP")* entonces puede recibir los mensajes que se le envían y reenviarlos.

   Por otra parte, si un nodo está en el estado inactivo, *("DOWN")* el mensaje se perderá definitivamente.

4. Se considera que el origen y los destinos tienen una confiabilidad y disponibilidad perfectas, es decir, siempre están en estado activo con probabilidad 1.

5.3. La red TCP

En el framework OMNeT++ la topología de la red, está completamente descripta por el archivo "*Ned.ned*", como se explica en el capitulo 4. A continuación se muestra una figura de como se ve la red ya implementada y el archivo correspondiente a la red TCP:

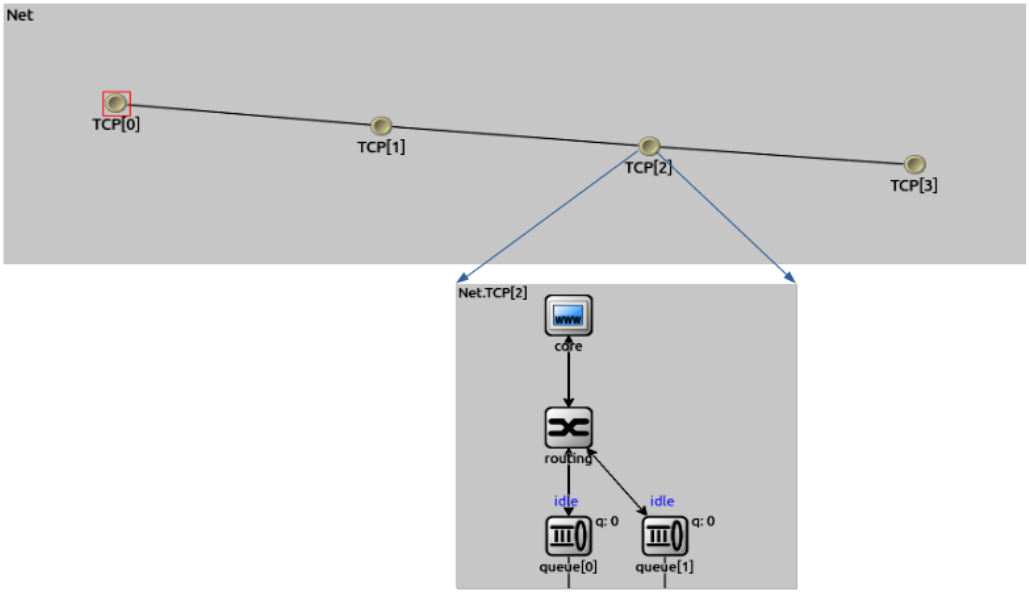


Figura 5.3: Red TCP, topología lineal, enlaces inalámbricos y el "Node" 2 como módulo compuesto.

El archivo "*Ned.ned*", donde se describe la topología de la red TCP se muestra a continuación:

Ned.ned

```
1  import ned.DatarateChannel;
2
3  //
4  // A four-node network topology
5  //
6  network Net
7  {
8      types:
9          channel C extends DatarateChannel
10         {// 1 ms channel delay
11             delay = 1ms;
```

```
12          }
13      submodules:
14          // array of 4 elements of type Node
15          TCP[4]: Node {
16              address = index;
17          }
18      connections:
19          // linear topology
20          for i = 0 .. 2 {
21              TCP[i].port++ <--> C <--> TCP[i+1].port++;
22          }
23  }
```

Como se puede observar del código, la red tiene definido un canal cuya retardo es de 1ms, los submódulos que conforman la red es un arreglo de 4 elementos del tipo "*Node*", la conexión es lineal ya que conecta el elemento TCP[i] con el elemento TCP[i+1]. Para la red DTN el archivo "*Ned.ned*" es esencialmente el mismo, con el único cambio en el nombre del arreglo de elementos, de "**TCP**" en las lineas 15 y 21 por "**DTN**", la diferencia entre las redes es la implementación del Node.

5.4. Modulo compuesto: Node

El módulo compuesto ***"Node"***, figura 5.4, define el comportamiento de cada nodo de manera general, al igual que para la red, y por ser estos módulos compuestos no tiene asignada una implementación directa en C++, es decir que a través del lenguaje "*.ned*" se describe su comportamiento.

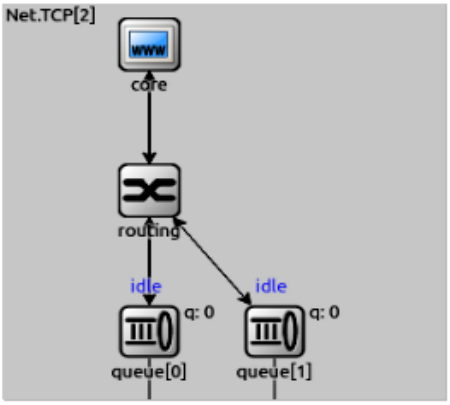


Figura 5.4: Nodo TCP y módulos que lo componen: "queue", "routing" y "core".

La topología del nodo, está completamente descripta por el archivo "*Node.ned*", a continuación se muestra el archivo correspondiente al nodo TCP:

Node.ned

```
//
// A "Node" consists of a routing, core  plus an
// queue module.
//
module Node {
    parameters:
        // identification of each node
        int address;

    gates:
        // node ports
        inout port[];

    submodules:
        // node elements
        core: Core {
            parameters:
                address = address;
        }
        routing: Routing {
            gates:
                in[sizeof(port)];
                out[sizeof(port)];
        }
        // queue array, size of the number of ports
        queue[sizeof(port)]: L2Queue

    connections:
        // routing-core connection
        routing.localOut --> core.in;
        routing.localIn  <-- core.out;
        // routing-queue-Node connection
        for i=0..sizeof(port)-1 {
            routing.out[i] -->  queue[i].in;
            routing.in[i]  <--  queue[i].out;
            queue[i].line  <--> port[i];
        }
}
```

En el código se puede apreciar que el parámetro del nodo ("*address*", un número entero) es su dirección, es decir, el índice que posee en el arreglo. La puerta de conexión con otros nodos denominada "*port[ ]*", es un arreglo,

cuya dimensión está dada por el número de conexiones que tenga cada nodo, la puerta es del tipo "***inout***", esto significa en el lenguaje NED que es una puerta tanto de entrada como de salida para los mensajes.
Luego los submódulos que lo componen denominados "*core*" del tipo **Core**, "routing" del tipo **Routing** y "queue" del tipo **L2Queue**, los tipos serán definitivamente las clases en C++ que implementarán a los módulos simples que se denominan en el modelo.
Finalmente están las conexiones dentro del nodo, las cuales unen a todos los elementos. Cada submódulo tiene sus propias puertas de conexión, definidas en el archivo NED de cada submódulo, por ejemplo "*routing*" tiene las puertas "localOut" solo de salida (los mensajes solo puede salir por ella) y "localIn" solo de entrada (los mensajes solo puede entrar por ella), estas puertas lo conectan al "*core*".

5.5. Módulos simples

En OMNeT++ los eventos ocurren dentro de módulos simples. Los módulos simples encapsulan el código C++ que genera eventos y reacciona a los eventos, implementando el comportamiento del módulo.
A continuación se describe brevemente la implementación de cada módulo simple (o tipo) que conforman el nodo compuesto y que es donde se establece el comportamiento algorítmico de cada nodo y por ende de la red.

5.5.1. Queue

La clase ***L2Queue*** implementa al módulo simple "*queue*", este conecta el interior de cada nodo con el exterior del nodo, por este módulo ingresan y egresan todos los mensajes que arriban al nodo.
Este módulo simple fue reutilizado completamente y sin modificaciones, el cual es provisto con el framework OMNeT++ en "*sample/routing*".
La figura 5.5 muestra las conexiones (o puertas) que unen a este módulo simple con el resto de los módulos del nodo y al nodo mismo, se debe tener en cuenta que habrá tantas colas de entrada/salida ("*queue*") como conexiones tenga el nodo. Es decir para los nodos de los extremos habrá solo una, mientras que para los nodos intermedios habrá dos.
En este caso la denominada puerta "*line*" de la cola de entrada/salida es la que comunica al nodo ("*node*") con la puerta del nodo, denominada "*port[]*", mencionada anteriormente y por tanto la que comunica el exterior con el interior del nodo, esta puerta es del tipo *"**inout**"*, es decir que los mensajes pueden tanto entrar como salir por ella.

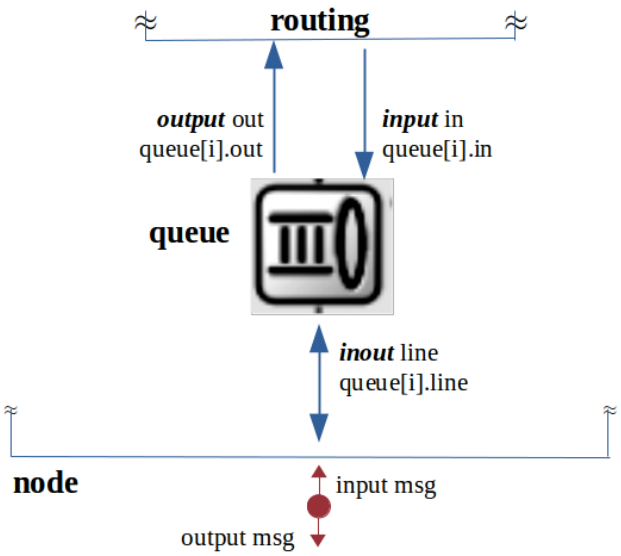


Figura 5.5: Esquema de comunicación del Modulo simple "**queue**" con el resto de los elementos del Nodo.

Luego las puertas que comunican la cola de entrada/salida con el router ("*routing*") son dos, una solo de entrada denominada *"in"* del tipo *"**input**"*, por la cual solamente llegan los mensajes desde el router hacia la cola de entrada/salida. La otra puerta denominada *"out"* del tipo *"**output**"* es una puerta solo de salida por la cual los mensajes van desde la cola de entrada/-salida hacia el router únicamente.
Puesto que este módulo es el "*buffer*" de entrada y salida para el manejo de los mensajes que arriban o salen de cada nodo, está implementado por una cola del tipo FIFO. En la figura 5.6 se muestra, a modo de ilustración, el diagrama UML de la clase **L2Queue.cc** que lo implementa.
Los atributos de la clase, por mencionar algunos son: "*frameCapacity*", del tipo **long** (tipo heredado de C++) determina la capacidad del buffer, "*queue*" la cola propiamente dicha, cuyo tipo es un contenedor del tipo **cQueue**, "*endTransmisionEvent*" un puntero al tipo **cMessage** y luego siguen una serie de atributos como "*qlenSignal*" del tipo **simsignal_t** todos ellos propios del framework OMNeT++, estos últimos atributos del tipo **simsignal_t** son usados para registrar, guardar y luego procesar ciertos eventos de interés.
Los mensajes están representados por instancias de la clase **cMessage** y sus subclases. Los mensajes se envían de un módulo a otro; esto significa que el lugar donde "ocurrirá el evento" es el módulo de destino del mensaje, y el tiempo de simulación cuando ocurre el evento es el momento de llegada del mensaje. El módulo también puede implementar eventos como "tiempo

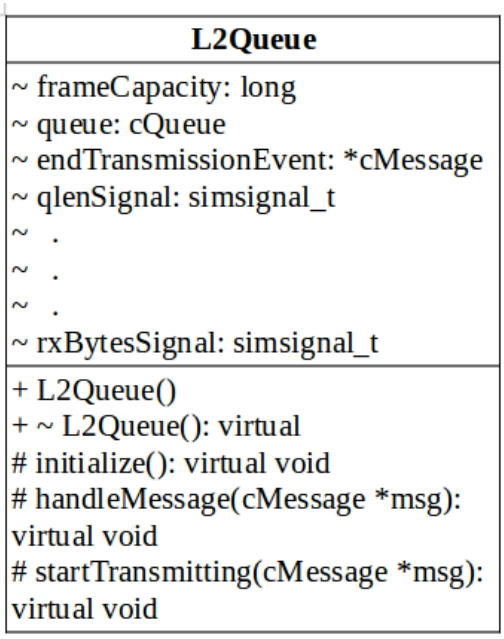


Figura 5.6: Modulo simple "*queue*" representado por la clase **L2Queue**.

de espera expirado" enviándose un auto-mensaje, este el caso del atributo "*endTransmissionEvent*" que es usado como auto-mensaje en el módulo.
El atributo "*queue*" es una clase de contenedor que actúa como una cola. **cQueue** puede contener objetos de tipo derivados de **cObject** (casi todas las clases de la biblioteca OMNeT++), como **cMessage**, **cPar**, etc. Normalmente, los elementos nuevos se insertan en la parte posterior y se eliminan de la parte frontal, figura 5.7.

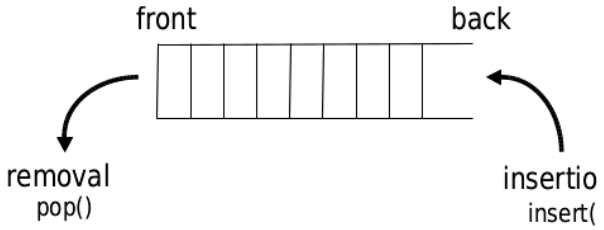


Figura 5.7: **cQueue** es un tipo proporcionado por el framework OMNeT++.

A continuación se describe en seudo código las funcionalidades de cada método de la clase, las funciones resaltadas en negrita dentro del algoritmo pertenecen al framework OMNeT++ o bien son funciones/métodos de la clase. Todas las clases son heredadas de **cSimpleModule** del framework, a su vez

todas los módulos simples deben poseer un método **initialize** y uno **handleMessage**.

El método **initialize()** se invoca después de que OMNeT++ ha configurado la red (es decir, creado módulos y los ha conectado de acuerdo con las definiciones establecidas en los archivos NED), entonces proporciona un lugar para el código de inicialización.

Algorithm 5.1: initialize

Input : void
Output: void

```
1 // Initialize all attributes of class
2 frameCapacity = par("frameCapacity")
3 queue.setName("queue")
4 endTransmissionEvent = new cMessage("endTxEvent")
5 qlenSignal = registerSignal("qlen")
6       .
7       .
8       .
9 rxBytesSignal = registerSignal("rxBytes")
```

Para este módulo los atributos se inicializan, como se puede observar para algunos a modo de ejemplo, la capacidad del buffer ("*frameCapacity*") es establecida por la función "***par***" del framework que toma como argumento el string "frameCapacity" que es un parámetro del modulo establecido en el archivo de configuración "*omnetpp.ini*".
La cola de entrada/salida ("*queue*") es seteada por la función ***setName*** con un string, que es su nombre.
La dirección del auto-mensaje ("*endTransmissionEvent*") (ya que es un puntero al tipo **cMessage**) se inicializa con un string, el cual es el nombre del mensaje.
Los atributos que representan señales (como "*qlenSignal*) deben registrase con la función del framework **registerSignal**, estas señales toman un nombre de señal como parámetro y devuelve el valor "*simsignal_t*" correspondiente, por ejemplo a "*qlenSignal*", etc.

El método **handleMessage**(cMessage ***msg**) se invoca con el mensaje como parámetro cada vez que el módulo recibe un mensaje. Se espera que **handleMessage** procese el mensaje y luego lo devuelva. El tiempo de simulación nunca transcurre dentro de las llamadas de **handleMessage**, solo entre ellas.

Algorithm 5.2: handleMessage

Input : *msg
Output: void

```
1  if (msg == endTransmissionEvent) {
2      // Transmission finished, we can start next one.
3      if (queue.isEmpty())
4          busySignal = false
5      else {
6          msg = queue.pop()
7          startTransmitting(msg)
8      }
9  }
10 else if (msg -> arrivedOn("line in"))
11     // pass up
12     send(msg, "out")
13 else {
14     // arrived on gate "in"
15     if (endTransmissionEvent -> isScheduled()) {
16         // We are currently busy, so just queue up the packet.
17         if (frameCapacity and queue.getLength() >= frameCapacity)
18             delete msg
19         else
20             queue.insert(msg)
21     }
22     else {
23         // We are idle, so we can start transmitting right away.
24         startTransmitting(msg)
25         busySignal = true
26     }
27 }
```

Veamos brevemente la algoritmia del método **handleMessage**, si el "msg" recibido es un auto-mensaje, es decir si es igual a "*endTransmissionEvent*", entonces inspecciona el buffer ("*queue*"), si está vacío, o sea no hay más mensajes para despachar establece una señal y termina, por otra parte si hay mensajes encolados comienza la transmisión de los que están al frente del buffer, mediante el método auxiliar ***startTransmitting***.
Luego si el mensaje arriba al nodo, es decir por la puerta "*line (input)*", lo envía directamente al router por la puerta "*out*", usando la función del framework ***send***.

La tercera posibilidad es que el mensaje ingrese al módulo proveniente desde el router, en este caso se trata de un mensaje que debe ser transmitido fuera del nodo, entonces verifica si hay algún auto-mensaje (esto significa que el buffer está enviando mensajes fuera del nodo, recordemos que los auto-mensajes funcionan como "timer", por lo que estaría el "timer" activado), si la capacidad del buffer fue superada el mensaje se perderá, si la capacidad del buffer no ha sido superada lo encola para su posterior envío. En el caso que no hubiera auto-mensaje (el buffer está disponible de manera inmediata, no hay "timer" activado), por lo que los mensajes son transmitidos fuera del nodo inmediatamente.

El método **startTransmitting**(cMassage ***msg**) es un método auxiliar usado en el método **handleMessage** que toma como argumento el mensaje y lo transmite por la puerta de salida del nodo ("*line (out)*"), luego establece un "*tiempo de espera expirado*" (o "timer") de duración "endTransmission" a través la función ***scheduleAt*** que establece el "disparo" del auto-mensaje "endTransmissionEvent".

Algorithm 5.3: startTransmitting

Input : *msg
Output: void

```
1 // Check and casts of "msg"
2 check_and_cast < msg >

3 // Send "msg" through the corresponding "gate"
4 send(msg, "line out")

5 // Schedule an event ("endTransmissionEvent") for the
6 // time ("endTransmission") when last bit will leave the gate
7 scheduleAt(endTransmission, endTransmissionEvent)
```

5.5.2. Routing - modelo de fallas en nodos

La clase ***Routing*** que implementa al módulo simple "*routing*", es el módulo que conecta la (o las) colas de entrada/salida ("*queue*") con el "*core*", y es el encargado de establecer el camino (o enrutamiento) que tendrán los mensajes, es decir este modulo es el router por lo que posee una tabla de enrutamiento determinada por la topología de la red y un algoritmo de enrutamiento.
En la figura 5.8 se esquematizan las puertas que unen a este módulo simple con el resto de los módulos del nodo.
La conexión con la (o las) colas de entrada/salida se realiza por intermedio de dos arreglos de puertas, denominados "*in*[ ]" y "*out*[ ]", cuyos tipos son

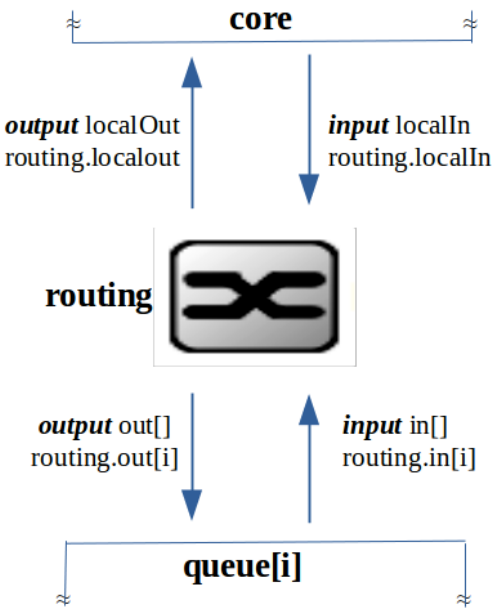


Figura 5.8: Esquema de comunicación del Modulo simple "*routing*" con el resto de los elementos del Nodo.

*"**input**"* y *"**output**"* respectivamente. La dimensión de estos arreglos estará determinada por el número de colas de entrada/salida que posea cada nodo. Ambos arreglos son unidireccionales, es decir, por "*in*[ ]" solo entran mensajes y por "*out*[ ]" solo salen mensajes.
Luego las puertas que lo comunican con el módulo "*core*" son dos, una solo de entrada denominada *"localIn"* del tipo *"**input**"*, por la cual solamente llegan los mensajes desde el core hacia el routing. y la otra puerta denominada *"localOut"* del tipo *"**output**"* es una puerta solo de salida por la cual los mensajes van desde el routing hacia el core únicamente.
Cabe mencionar que para el problema planteado de topología lineal y para la red que nos ocupa (TCP) los router de los nodos intermedios solo pasan los mensajes al nodo contiguo, sin procesar ni modificar los mensajes que llegan al nodo.
Como ya se mencionara anteriormente los módulos simples están implementados por una clase en C++, en la figura 5.9 se muestra el diagrama UML de la clase **Routing.cc** que lo implementa.
Los atributos de la clase son: "*myAddress*", del tipo **int** (tipo heredado de C++) determina el índice (o dirección) del nodo en la red.
La falla del subsistema digital será introducida en los router, de acuerdo a lo explicado anteriormente, esta falla estará implementada por una maqui-

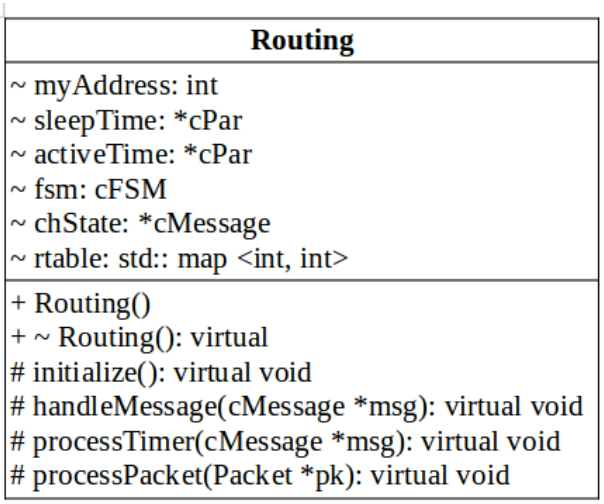


Figura 5.9: Modulo simple representado por la clase **Routing**.

na finita de estados (FSM por sus siglas en ingles), de dos estados "*UP*" y "*DOWN*". Los atributo "*sleepTime*" y "*activeTime*" son punteros al tipo ***cPar***, tipo propio del framework, serán proporcionados como parámetros desde el archivo de configuración "*omnetpp.ini*". Cada uno de ellos determinará el tiempo de actividad y el tiempo de inactividad o falla respectivamente, usando para ello variables aleatorias exponenciales.
El atributo "*fsm*" del tipo ***cFSM*** también de OMNeT++, proporciona un conjunto de macros para construir maquinas de estado finitas.

Los puntos clave de FSM son:

- Hay dos tipos de estados: *transitorios* y *estables*. En cada evento (es decir, en cada llamada a **handleMessage**), el FSM pasa del estado actual (estable), sufre una serie de cambios de estado (se ejecuta a través de varios estados transitorios) y finalmente llega a otro estado estable. Así, entre dos eventos, el sistema está siempre en uno de los estados estacionarios (o estable). Por lo tanto, los estados transitorios no son realmente obligatorios: existen solo para agrupar las acciones que se tomarán durante una transición de manera conveniente.

- Puede asignar un código de programa para manejar la entrada y salida de un estado (conocido como código de entrada/salida). Permanecer en el mismo estado se maneja como salir y reingresar al estado.

- El código de entrada no debe modificar el estado (esto es verificado por OMNeT++). Los cambios de estado (transiciones) deben colocarse en

el código de salida.

El cambio de estados en la FSM mencionada en el párrafo anterior, del estado "*UP*" al estado "*DOWN*" se realiza implementando eventos como "tiempo de espera expirado" (un "timer") al enviarse, el módulo, un auto-mensaje. El auto-mensaje para realizar esto es el atributo "*chState*", un puntero al tipo ***cMessage***.
Por ultimo el atributo "*rtable*" del tipo ***map***, es un diccionario heredado de la librería estándar de C++. El cual será la tabla de enrutamiento para direccionar los mensajes.

A continuación se describe en seudo código las funcionalidades de cada método de la clase, las funciones resaltadas en negrita pertenecen al framework OMNeT++ o bien son funciones auxiliares del método.

Como antes el método **initialize**() se invoca después de que OMNeT++ ha configurado la red (es decir, creado módulos y los ha conectado de acuerdo con las definiciones), en este caso el método debe configurar la tabla de enrutamiento "*rtable*".

Como punto de partida, para ello se usa la clase ***cTopology***. Un objeto cTopology almacena una representación abstracta de la red en forma de grafo:

- cada nodo de ***cTopology*** corresponde a un módulo (simple o compuesto), y
- cada arista de ***cTopology*** corresponde a un enlace o serie de enlaces de conexión.

Se puede especificar qué módulos incluir en el grafo. Los módulos compuestos también pueden seleccionarse.
El grafo incluirá todas las conexiones entre los módulos seleccionados. Las conexiones que se extienden a través de los límites del módulo compuesto también se representan como una arista del grafo. Los bordes del grafo están dirigidos, al igual que las puertas del módulo.
El algoritmo de enrutamiento usado por defecto es el algoritmo de Dijkstra para encontrar el camino mas corto de cada nodo al resto del grafo.
Luego que se establece la tabla de enrutamiento se borra el objeto ***cTopology***, se establece el nombre para la maquina finita de estados, y para nuestro ejemplo de red, establecemos que solo falle el router correspondiente al nodo 2.

Algorithm 5.4: initialize

Input : void
Output: void

```
1  // Initialize attributes of class
2  sleepTime = &par("sleepTime")
3  activeTime = &par("sleepTime")
4  // The routing map is built
5  // A cTopology object stores an abstract representation of the network
6  cTopology *topo = new cTopology("topo")
7  // Extracts model topology by the fully qualified NED.
8  topo -> extractByNedTypeName("vector with Nodes")
9  // Returns the graph node which corresponds to the given module.
10 cTopology::Node *thisNode = topo->getNodeFor("name of Node")
11 // find and store next hops
12 for i = 0 to topo ->getNumNodes() {
13     if (topo -> getNode(i) == thisNode)
14         continue // skip ourselves
15     // Apply the Dijkstra algorithm to find all shortest paths
16     // to the given graph node.
17     topo->calculateUnweightedSingleShortestPathsTo("Node i")
18     if (thisNode->getNumPaths() == 0)
19         continue // not connected
20     // finally we get the routing table
21     rtable[address] = gateIndex
22 }
23 delete topo
24 // OMNeT++ provides a class and a set of macros to build FSM
25 fsm.setName("fsm")
26 // we establish that only router 2 can have failures
27 if (myAddress == 2 ) {
28     chState = new cMessage("chState")
29     scheduleAt(0, chState)
30 }
```

El método **handleMessage**(cMessage ***msg**) se invoca con el mensaje, **msg** como parámetro, cada vez que el módulo recibe un mensaje o bien un auto-

mensaje, se espera que **handleMessage** procese el mensaje y luego lo devuelva. El tiempo de simulación nunca transcurre dentro de las llamadas de **handleMessage**, solo entre ellas.
El handleMessage realiza dos acciones, una a través del método ***processTimer*** que implementa la maquina finita de estados para simular la falla en el nodo, solo es llamado cuando el mensaje entrante es un auto=-mensaje.
El otro método ***processPacket*** que determina el destino y envió de los mensajes que no son auto-mensajes, usando la tabla de enrutamiento "*rtable*".

Algorithm 5.5: handleMessage

Input : msg
Output: void

```
1  // Process the self-message or incoming message
2  // only node 2 fails
3  if (myAddress == 2) {
4      if (msg -> isSelfMessage())
5          // Process the self-message in the FSM
6          processTimer(msg)
7      else {
8          if (fsm.getStateName() == "UP")
9              // Process the incoming message
10             processPacket(msg)
11         else if (fsm.getStateName() == "DOWN")
12             // Delete message
13             delete(msg)
14     }
15 }
16 else
17     processPacket(msg)
```

El método **processTimer**(cMessage ***chState**) implementa la maquina finita de estados, se invoca con el parámetro "*chState*", es decir un auto-mensaje.
El estado de la FSM se almacena en un objeto de tipo **cFSM**.
Los estados posibles están definidos por una enumeración; la enumeración también es un lugar para definir qué estado es transitorio y cuál es estable.
En nuestro caso solo se tienen estados estacionarios, **DOWN** y **UP**.

```
enum {
    INIT = 0,
    DOWN = FSM_Steady(1),
```

```
        UP    = FSM_Steady(2),
    }
```

La FSM está incrustada en una declaración de tipo **switch**, ***FSM_Switch***(), con casos para ingresar y salir de cada estado.
Las transiciones de estado se realizan mediante llamadas a ***FSM_Goto***(), que simplemente almacena el nuevo estado en el objeto **cFSM**.

Algorithm 5.6: processTimer

Input : chState
Output: void

```
1  d   // length of period for change of state
2  FSM_Switch(fsm) {
3      case FSM_Exit(INIT):
4          // transition to DOWN state
5          FSM_Goto(fsm, DOWN)
6          break
7      case FSM_Enter(DOWN):
8          // schedule end of sleep period
9          d = sleepTime
10         scheduleAt(d, chState)
11         break
12     case FSM_Exit(DOWN):
13         // schedule end of active period
14         d = activeTime
15         scheduleAt(d, chState)
16         // transition to UP state
17         FSM_Goto(fsm, UP)
18         break
19     case FSM_Enter(UP):
20         // status change
21         break
22     case FSM_Exit(UP):
23         // transition to DOWN state
24         FSM_Goto(fsm, DOWN)
25         break
26 }
```

Cuando el auto-mensaje "*chState*" ingresa al ***handleMessage***, éste entra al

estado “INIT” e inmediatamente pasa al estado “DOWN”, ahí se establece la duración (el “timer”) de dicho estado mediante el atributo “*sleepTime*” (parametrizado). Por intermedio de la función ***scheduleAt*** se especifica que el auto-mensaje se vuelva dispara transcurrido dicho tiempo.
Después de cumplirse el tiempo “*sleepTime*” el auto-mensaje “*chState*” ingresa por el caso “Exit” del estado “DOWN” y antes de dejar el estado “DOWN” establece el temporizador con la duración “*activeTime*” y pasa al estado “UP”, cuando se cumple el tiempo “*activeTime*”, nuevamente se dispara el auto-mensaje “*chState*” para pasar al estado “DOWN” y así sucesivamente.

Algorithm 5.7: processPacket

Input : pk
Output: void

```
1  // Get the address of the message
2  int destAddr = pk -> getDestAddr()
3  if (myAddress == destAddr) {
4      // Local delivery of packet
5      send(pk, “localOut”)
6      return
7  }
8  // Routing table iterator
9  iterator it = rtable.find(destAddr)
10 if (it == rtable.end()) {
11     // Address “destAddr” unreachable, discarding message
12     delete(pk)
13     return
14 }
15 // Forwarding packet on gate index “outGateIndex”
16 int outGateIndex = (*it).second()
17 send(pk, “out”, outGateIndex)
```

El método **processPacket**(Packet ***pk**) se invoca con el parámetro **pk**, finalmente este método es el encargado de enviar el mensaje según lo establecido por el atributo “*rtable*”, el diccionario confeccionado en el método **initialize**, por lo tanto el mensaje puede tener destino el “*core*” del nodo o ser reenviado a otro nodo según lo establecido por la tabla de enrutamiento.

5.5.3. Core

El módulo simple "*core*", implementado por la clase "***Core***", está conectado al router ("routing"), y es el encargado de generar el mensaje (o paquete) que será enviado, de igual manera es el encargado de recibir el mensaje (o paquete) cuando el nodo en cuestión sea el de destino, a demás de llevar las estadísticas de saltos y tiempos de demora de extremo a extremo.

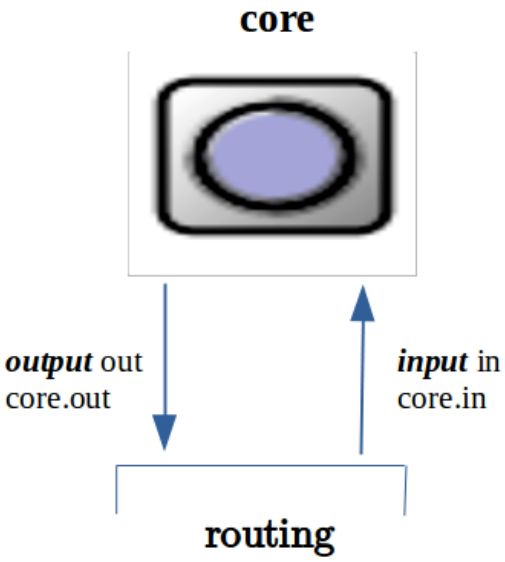


Figura 5.10: Esquema de comunicación del módulo simple "*core*" con "*routing*".

En este caso el módulo cuenta con dos puertas únicamente, la denominada puerta "*in*" es una puerta del tipo *"**input**"*, es decir que los mensajes provenientes del router ingresan al core por esta puerta.
Y la denominada puerta "*out*" es una puerta de salida y es del tipo *"**output**"*, es decir que los mensajes generados o procesados en el core solo pueden salir de éste hacia el router por ella.
Para el problema planteado de topología lineal y para la red que nos ocupa (TCP), el único core que generará mensajes es el core cero y el único core que recibirá los mensajes será el core tres, este a su vez le retransmitirá un mensaje de reconocimiento al core cero.
La clase en C++ de la figura 5.11 muestra el diagrama UML de Core.cc.
El atributo "*myAddress*", del tipo **int** (tipo heredado de C++) determina el índice (o dirección) del nodo en la red. "*destAddresses*" es un vector de enteros (tipo heredado de C++) que contiene los lugares de cada nodo en la red. "*buffer*" es una cola FIFO, del tipo **cQueue**, donde se guardan los

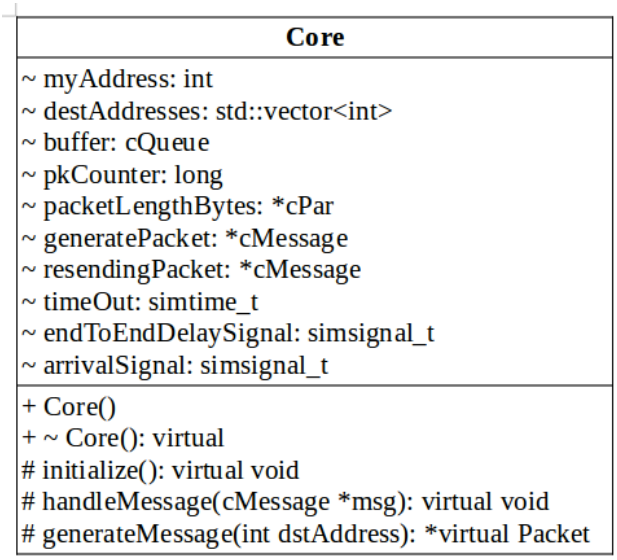


Figura 5.11: Modulo simple representado por la clase "*Core*".

mensajes generados para ser enviados, el hecho de guardar este mensaje responde al hecho de que podría perderse en el trayecto (encontrar un router en estado "DOWN") y por lo tanto pasado el tiempo ideal de espera (6 ms), aguardando el mensaje de reconocimiento enviado por el core 3, el core 0 reenvía el mismo mensaje nuevamente, que se obtiene del frente del buffer. "*pkCounter*" es el número del mensaje, del tipo **long** heredado de C++. "*packetLengthBytes*" es un puntero a un parámetro que se establece desde el archivo "*omnetpp.ini*", representa el tamaño en bytes del mensaje.
Luego los auto-mensajes "*generatePacket*" y "*resendingPacket*", punteros al tipo **cMessage**, estos son los encargados de generar un nuevo mensaje si el mensaje de reconocimiento llega en el tiempo ideal desde el nodo 3 al nodo 0, o de reenviar el mensaje guardado por si nunca llegara el mensaje de reconocimiento, respectivamente. El atributo "*timeOut*", del tipo **simtime_t** del framework, almacena el tiempo ideal (de 6 ms) en el que debería llegar el mensaje desde que es enviado por el nodo 0, receptado por el nodo 3 mas la demora del mensaje de reconocimiento enviado por el nodo 3 hacia el nodo 0. Por último los atributos "*endToEndDelaySignal*" y "*arrivalSignal*" son del tipo **simsignal_t** propio de OMNeT++ y son los encargados de grabar los resultados del tiempo de demora de extremo a extremo y la cantidad de saltos de cada mensaje respectivamente.
A continuación se describe en seudo código las funcionalidades de cada método de la clase, las funciones resaltadas en negrita pertenecen al framework OMNeT++.

Como antes el método **initialize**() se invoca después de que OMNeT++ ha configurado la red (es decir, creado módulos y los ha conectado de acuerdo con las definiciones), en este caso el método debe configurar las direcciones de cada nodo y por ende de cada core, estas direcciones (o índices del arreglo, nodos en la red) son proporcionados como parámetros a través del archivo "*omnetpp.ini*" como un string por lo que tiene que ser parceado por el método, a través de la función ***tokenizer***, del framework.

Algorithm 5.8: initialize

Input : void
Output: void

```
1  // Initialize attributes of class
2  myAddress = &par("address")
3  packetLengthBytes = &par("packetLength")
4  buffer.setName("buffer")
5  // The "core" indices is built
6  // It considers the input string to consist of tokens, separated by one
7  // or more delimiter characters
8  const char *destAddressesPar = &par("destAddresses")
9  cStringTokenizer tokenizer(destAddressesPar)
10 const char *token
11 int k = 0
12 while ((token = tokenizer.nextToken()) != nullptr) {
13     destAddresses.push_back(atoi(token))
14     k++
15 }
16 // we establish that only core 0 can generate message
17 if (myAddress == 0) {
18     // ideal delay time in seconds
19     timeOut = 0.006
20     generatePacket = new cMessage("nextPacket")
21     resendingPacket = new cMessage("resendingPacket")
22     scheduleAt(0, generatePacket)
23 }
```

El método **handleMessage**(cMessage ***msg**) se invoca con el mensaje como parámetro cada vez que el módulo recibe un mensaje. Se espera que **handleMessage** procese el mensaje y luego lo devuelva.

Algorithm 5.9: handleMessage

Input : *msg
Output: void

```
1  if (msg == generatePacket) {
2      // Creation and sending message, destination address is generated
3      int destAddress = destAddresses.size()-1
4      Packet *msg = generateMessage(destAddress)
5      // We record the creation time of a new message
6      msg->setTimestamp()
7      buffer.insert(msg)
8      Packet *copy = buffer.front()->dup()
9      send(copy, "out")
10     scheduleAt(timeOut, resendingPacket)
11 }
12 else if (msg == resendingPacket) {
13     // If the message was lost and the acknowledgment
14     // did not arrive in the stipulated time "timeOut",
15     // we will resend the message
16     msg = buffer.front()->dup()
17     send(msg, "out")
18     scheduleAt(timeOut, resendingPacket)
19 }
20 else {
21     // Handle incoming packet
22     if (myAddress == 0 && msg->getDestAddr() == 0) {
23         // Record the message arrival time and generate new message,
24         // delete the message in the buffer and the message arrived
25         scheduleAt(simTime(), generatePacket)
26     }
27     else if (myAddress == destAddresses.size()-1) {
28         // The message reached the last node
29         // We change the origin and destination and forward it
30         send(msg, "out")
31     }
32 }
```

Este método considera 3 casos:

1. Si el mensaje es el auto-mensaje "*generatePacket*", entonces se genera un mensaje nuevo, se establece el tiempo de creacion del mensaje, se

guarda en el buffer, se envia una copia al destino y se establece un "timer" de duración "*timeOut*" con el otro auto-mensaje "*resendingPacket*".

2. Si el mensaje es el auto-mensaje "*resendingPacket*", entonces se toma el mensaje desde el buffer, se envía una copia al destino y se establece un "timer" de duración "*timeOut*" con el mismo auto-mensaje "*resendingPacket*".

3. Si el mensaje no es un auto-mensaje, significa que ha arribado un mensaje externo al nodo, acá a su vez se diferencian dos casos. Uno, si el nodo en cuestión es el nodo 0, lo que significaría que arribó el mensaje de reconocimiento, en cuyo caso se guardan los valores de señal para la estadística y se genera un nuevo mensaje. Y dos, que el nodo sea el de destino, en cuyo caso se modifica levemente el mensaje cambiando el destino y origen y se reenvía.

El método **generateMessage**(int destAddress) es un método auxiliar usado en el método **handleMessage** que toma como argumento un entero, el destino del mensaje, y retorna un mensaje (o paquete).

Algorithm 5.10: generateMessage

Input : int
Output: *Packet

```
1 // Generate a new message or packet
2 Packet *pk = new Packet("name_of_packet")
3 // set message attributes and message number
4 pkCounter++
5 return pk
```

De esta manera, el simulador OMNET++ puede capturar las estadísticas del comportamiento relevantes para este trabajo. Dichas estadísticas se mostrarán en el capítulo 7.

Capítulo 6

Estudio de casos - La red DTN

Las Redes Tolerantes a Demoras (DTNs), son principalmente concebidas para largas distancias entre nodos y en escenarios donde otros servicios de redes regulares son altamente disruptivos o no pueden aplicarse debido a la falta de la infraestructura requerida. Sin embargo, es imposible para un nodo DTN diferenciar entre las altas demoras o disrupciones debidos a las largas distancias o a las oclusiones planetarias o la falla de un nodo vecino.
De hecho, un nodo que no está respondiendo debido a una falla transitoria, produce el mismo efecto que un nodo volando en el lado opuesto de un planeta remoto. De acuerdo con esta visión, se puede afirmar que las principales características y servicios de los protocolos DTN se pueden aplicar también a escenarios de muy cortas distancias y muy rápida respuesta donde los nodos componentes evidencian una alta tasa de fallas.
Como consecuencia, las soluciones DTN pueden adaptarse e implementarse a escala de chip, como un paradigma de conectividad confiable y de soporte de las denominadas redes en chips o (NOCs). El paradigma NOC se está convirtiendo rápidamente en la infraestructura de comunicación estándar para proporcionar comunicación escalable entre núcleos. La investigación ha demostrado que las interconexiones metálicas causan una alta latencia y consumen un exceso de energía en las arquitecturas NoC. Las tecnologías emergentes, como las interconexiones inalámbricas en chip, pueden aliviar los problemas de potencia y ancho de banda de los NoC metálicos tradicionales.
Aunque los NOC inalámbricos permiten una flexibilidad sin precedentes y transmisiones de un solo salto a larga distancia, su complejidad supera significativamente la de los NOC cableados tradicionales. Como resultado, los NOC inalámbricos son potencialmente menos confiables y más propensos a fallas transitorias o permanentes provocadas no solo por errores de diseño sino también por fenómenos externos no deseados, como el efecto de la radiación que afecta actualmente a la electrónica moderna.

Introducimos el término WDTNOC ("*Wireless Delay Tolerant Networks On Chips*") que significa DTN en un chips, o más precisamente SOCs (System On Chips) con conexiones inalámbricas. No obstante, afirmamos que los principios estudiados en este trabajo también se aplican a DTN entre chips, a la que denotaremos WDTNBC "*Wireless Delay Tolerant Networks Between Chips*".
Después de discutir la adaptación de DTN en una escala de chip, presentamos un estudio de caso simple pero atractivo que es modelado y analizado teóricamente. Los resultados muestran que la confiabilidad general del NOC inalámbrico se puede mejorar dramáticamente si el flujo de datos entre nodos puede tolerar los retrasos impuestos por los nodos defectuosos.

6.1. Definiciones Básicas de Robustez

El término *robustez* se usa en varias áreas con diferentes significados [38]. En esta sección, se discute un significado más preciso en el contexto de este trabajo y de futuros trabajos relacionados con éste. En particular, se distingue entre dos tipos principales de robustez de acuerdo a su estado en el ciclo de vida del sistema: Robustez en el Estadío de Desarrollo (RED) y Robustez en el Estadío de Operación y Mantenimiento (REOM).

6.1.1. Robustez en estadío de desarrollo (RED)

Es indiscutible que los requerimientos siempre son susceptibles de sufrir cambios o correcciones durante todo el proceso de desarrollo de un sistema. Esto es especialmente cierto en el caso de sistemas de vanguardia y muy complejos con altos requisitos de seguridad y confiabilidad. Por lo tanto, el enfoque tradicional y lineal de establecer los requisitos de un sistema como primer paso en el proceso de desarrollo es inherentemente una fuente seria de debilidad y fragilidad del sistema.
En este contexto, una combinación adecuada de los NOC inalámbricos, y particularmente los enfoques WDTNOCs y WDTNBC, permiten cambios en los requisitos y correcciones a lo largo de todo el ciclo de vida del sistema.
De hecho, los módulos (es decir, nodos) dentro de un diseño determinado de chip o entre diferentes chips ya no están físicamente acoplados en un enfoque de red inalámbrica. Además, si el protocolo entre los módulos se especifica y es estable, como en el caso de los protocolos DTN estandarizados, los módulos estarán también lógicamente desacoplados.
Esto permite que nodos específicos del sistema en red incurran en cambios drásticos de diseño y requisitos con un impacto mínimo en el resto del siste-

ma, a medida que atraviesa su etapa de desarrollo. Este enfoque desacoplado ya se está habilitando en diferentes dominios, como los sistemas espaciales fraccionados [22, 31–33] y sistemas de nube basados en contenedores de plataforma como servicio (PaaS) [39, 40].
Las NOC inalámbricas y las WDTNOCs son la misma expresión para sistemas micro electrónicos complejos en un chip. La misma idea puede extenderse a las WDTNBCs, es decir entre *SOCs*. En ambos casos comunicaciones ópticas o de radio de ultra alta frecuencia permiten impresionantes velocidades de transferencias de datos y bajas latencias.

6.1.2. Robustez en operación y mantenimiento (REOM)

Una vez desarrollado y fabricado, el siguiente objetivo es poner el sistema en funcionamiento lo antes y el mayor tiempo posible. Una vez en funcionamiento, el sistema requiere mantenimiento para sostener el servicio según lo definido por las especificaciones de diseño.
El mantenimiento puede ser automático, mediante intervención humana o ambos. En cualquier caso, una plataforma inalámbrica y WDTNOC sirve como una infraestructura muy conveniente y cualquier falla en cualquiera de los módulos se puede mapear a un retraso de transmisión de datos. En este contexto, la robustez se puede definir rigurosamente no solo en términos de las conocidas fórmulas de confiabilidad y disponibilidad, sino también en términos de métricas de rendimiento del sistema.
Como primera conclusión y beneficio en el contexto de las redes WDTNOCs y WDTNBCs, se puede afirmar que la robustez se puede traducir o mapear en una tasa de degradación paulatina o en un requisito de degradación paulatina [38].
Sin embargo, las ecuaciones clásicas para los modelos de confiabilidad y disponibilidad aún se aplican en este contexto. En este trabajo, nos enfocamos en REOM como un proceso de optimización de confiabilidad y disponibilidad. De hecho, el caso de estudio en la sección 5.2 está enfocado en lograr una mejora en la confiabilidad y disponibilidad del sistema al aprovechar el manejo de datos de las DTN. La confiabilidad y disponibilidad del sistema (también conocida como dependibilidad) puede verse comprometida debido a agentes ambientales como la radiación ionizante [41]. La hipótesis es que la presencia de múltiples nodos DTN en el contexto de las WDTNOCs y las WDTNBCs puede usarse para mejorar la disponibilidad.

6.2. De DTN a WDTNOCs y WDTNBCs

Si reducimos drásticamente las distancias de las redes de clústeres de satélites interplanetarios o cercanos a la Tierra en varios órdenes de magnitud, pero conservando las capacidades de tolerar retraso o interrupción, llegamos al dominio de las WDTNOCs y/o de las WDTNBCs.
Por supuesto, en una distancia tan reducida, se pueden requerir cambios en los protocolos para operar de manera adecuada y eficiente en tal escenario. El Cuadro 6.1 sirve como una comparación resumida entre DTNs, WDTNOCs y WDTNBCs y como un indicador de la complejidad de los diferentes aspectos de DTN en cada uno de los escenarios.

Cuadro 6.1: Comparación Indicativa de Escenarios de Aplicación de DTN. [37]

Parameter	DTN	WDTNBC	WDTNOC
Distancia	10^3 a 10^7 m	10^{-3} a 10^{-1} m	10^{-6} a 10^{-3} m
Demora	10^2 a 10^5 s	10^{-2} a 10^{-1} s	10^{-4} a 10^{-2} s
Ruteo	Complejo	Simple	Simple
Tráfico	Complejo	Simple	Simple
Congestión	Compleja	Moderada	Moderada
Planning	Complejo	Simple	Simple
Buffering	Complejo	Moderado	Moderado
Acceso al Medio	Complejo	Moderado	Moderado

Como se notó anteriormente, la primera y principal diferencia es la distancia entre nodos. Si la distancia se reduce, la demora tolerada por los nodos en la red también disminuye. Nótese que no existe una relación lineal directa entre la reducción de distancia y la reducción de retardo. Sin embargo, como se puede ver en el Cuadro 6.1, también hay varios órdenes de magnitud de diferencia en la tolerancia de retraso de tiempo manejadas en cada caso. El retraso tolerado por la red, por otro lado, no solo está directamente relacionado con la distancia dividida por la velocidad de la luz. En cambio, el retraso también depende de factores operativos y de los requisitos generales de la arquitectura de red.
Por otro lado, además del hecho de que la demora tolerada en WDTNOCs y

WDTNBCs es mucho menor que en el escenario normal de DTN, los algoritmos de enrutamiento, tráfico y gestión de congestión se vuelven mucho más simples.
De hecho, no se espera que la capacidad de procesamiento en tiempo real o que la necesidad de recursos excesivos puedan aparecer como un limitante para implementar WDTNOCs y WDTNBCs. En particular, la planificación de contactos es un problema muy complejo en DTN para uso espacial y una fuente principal de fragilidad [14], pero en WDTNOCs y WDTNBCs, se puede simplificar en gran medida.
El hecho de que la gestión de la planificación de contactos y otros problemas complejos de DTN puedan simplificarse dramáticamente producirá una simplificación correspondiente en el enrutamiento y la gestión del tráfico en el caso de WDTNOC y WDTNBC.
Con un mecanismo de control de acceso al medio y almacenamiento en búfer adecuados, el problema de la congestión también se puede simplificar. Como consecuencia principal, la mayoría de los problemas en general de DTN se pueden reducir y manejar en tiempo real por hardware en WDTNOCs y WDTNBCs. Cada nodo DTN tiene la capacidad de recibir un paquete de datos, almacenarlo, habilitar el modo de custodia y esperar a que el siguiente nodo pueda recibir estos datos. Una vez que el nodo puede recibir el paquete, el nodo anterior reenvía el paquete y abandona el modo de custodia. Este manejo de datos de almacenamiento y transporte es el aspecto principal que debe heredarse en WDTNOCs y WDTNBCs. De hecho, esta propiedad de DTNs se utilizará como el habilitador principal para mejorar la confiabilidad en WDTNOCs y WDTNBCs inalámbricas.

6.3. La red DTN

Al igual que para la red TCP, la topología de la red DTN, está completamente descripta por el archivo "*Ned.ned*" con mínimos cambios, a continuación se muestra la figura 6.1 de como se ve la red ya implementada, en el framework.

6.4. Modulo compuesto Node

El módulo compuesto "***Node***", como se aprecia en la figura 6.1 en particular el nodo 2, define el comportamiento de cada nodo de manera general, al igual que para la red (también un modulo compuesto), estos módulos compuestos no tiene asignado una implementación directa en C++, si no que a través del lenguaje *.ned* se describe su comportamiento.

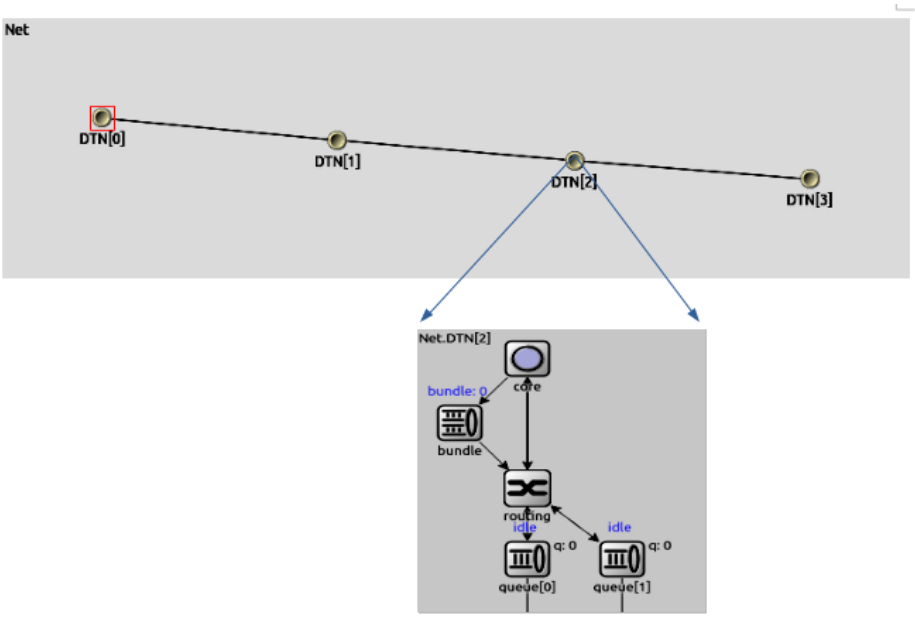


Figura 6.1: Red DTN con topología lineal. Con enlaces inalámbricos y el "*Node 2*" como módulo compuesto.

La topología del nodo, está completamente descripta por el archivo "*Node.ned*", a continuación se muestra el archivo correspondiente al nodo DTN:

Node.ned

```
//
// A "Node" consists of a Routing, an core plus bundle.
//
module Node
{
    parameters:
        // identification of each node
        int address;
    gates:
        // node ports
        inout port[];
    submodules:
        // node elements
        core: Core {
            parameters:
                address = address;
        }
        routing: Routing {
            gates:
```

```
20                  in[sizeof(port)];
21                  out[sizeof(port)];
22          }
23          // queue array, size of the number of ports
24          queue[sizeof(port)]: L2Queue
25
26          bundle: Bundle
27      connections:
28          // routing-core connection
29          routing.localOut --> core.in;
30          core.outExit--> routing.localInExit;
31          // core-bundle connection
32          core.outStore --> bundle.inBundle;
33          // bundle-routing connection
34          routing.localInStore <-- bundle.outBundle;
35          // routing-queue-Node connection
36          for i=0..sizeof(port)-1 {
37              routing.out[i] -->  queue[i].in;
38              routing.in[i]  <--  queue[i].out;
39              queue[i].line  <--> port[i];
40          }
41  }
```

En el código se puede apreciar que el parámetro del nodo es su dirección ("*address*"), es decir, el índice que posee en el arreglo, las puertas de conexión con otros nodos denominadas "*port[ ]*", es un arreglo dado por el número de conexiones que tenga cada nodo, luego los submódulos que lo componen denominados "*core*" del tipo **Core**, "*routing*" del tipo **Routing**, "*bundle*" del tipo **Bundle** y "*queue*" del tipo **L2Queue**, finalmente las conexiones dentro del nodo uniendo a todos los elementos. Los tipos o nodos simples están representados en OMNeT++ por clases en C++ que los implementan.

6.5. Módulos simples

En OMNeT++, los eventos ocurren dentro de módulos simples. Los módulos simples encapsulan el código C++ que genera eventos y reacciona a los eventos, implementando el comportamiento del módulo.
A continuación se describe brevemente la implementación de aquellos nodos o la parte de cada módulo simple (o tipo) que conforman el nodo compuesto que tenga variaciones con respecto al de la red TCP, y que es donde se establece el comportamiento algorítmico de cada nodo y por ende de la red DTN.
Se deja de lado la descripción de las colas de entrada al nodo pues estas no sufren ninguna modificación a lo explicado para la red TCP.

6.5.1. Routing - modelo de fallas en nodos

El módulo simple "*routing*" implementa al tipo **Routing**, éste es el que conecta la (o las) colas ("*queue*") de entrada/salida con el "*core*" y el "*bundle*", y es el encargado de establecer el camino (o enrutamiento) que tendrán los mensajes, es decir este modulo posee una tabla de enrutamiento determinada por la topología de la red.

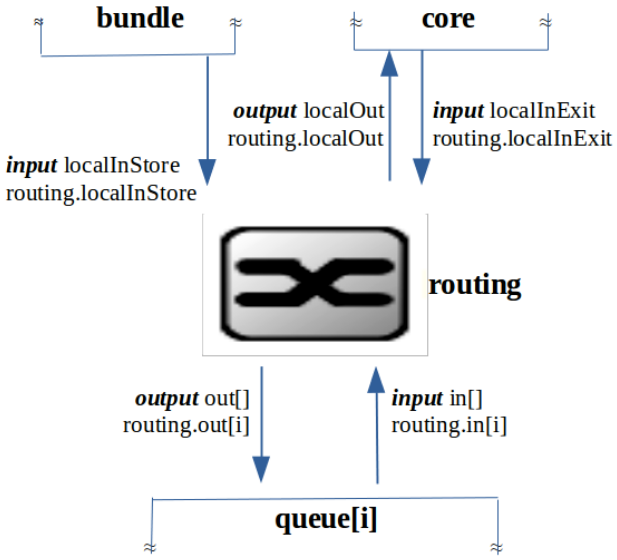


Figura 6.2: Esquema de comunicación del Modulo simple Routing con el resto de los elementos del Nodo.

En este problema planteado de topología lineal y para la red que nos ocupa (DTN) los router de los nodos intermedios pasan los mensajes al core si es que el router se encuentra activo (en el caso que el router estuviera inactivo el mensaje se pierde), el mensaje, luego de ser procesado por el core y almacenado en el bundle será enrutado según la tabla de enrutamiento que tenga el router.
En la figura 6.2 se esquematizan las puertas que unen a este módulo simple con el resto de los módulos del nodo, la única diferencia en cuanto a conexiones con respecto al router explicado y empleado en la red TCP es la conexión con el modulo bundle, ya que este modulo no existe en la red TCP.
Esta conexión extra con el bundle, denominada "*localInStore*" del tipo "***input***" comunica los mensajes que salen del bundle hacia el router.
Como ya se mencionara anteriormente los módulos simples están implementados por una clase en C++, en la figura 5.9 se muestra el diagrama UML de la clase **Routing.cc** que lo implementa. Hay que resaltar que la estructura

de la clase no cambia en nada, es decir los atributos y métodos son los mismos, la única variación es en la algoritmia de la implementación del método **handleMessage** que la compone.

Algorithm 6.1: handleMessage

Input : msg
Output: void

```
1  // Process the self-message or incoming message
2  if (myAddress == 2) { // only node 2 fails
3      if (msg -> isSelfMessage())
4          // Process the self-message in the FSM
5          processTimer(msg)
6      else {
7          // If the message is not your own message
8          if (myAddress != msg -> getSrcAddr()) {
9              if (fsm.getStateName() == "ACTIVE")
10                 // Appropriate message, send to the core
11                 send(msg, "localOut")
12             else if (fsm.getStateName() == "SLEEP")
13                 // Delete message
14                 delete(msg)
15         }
16         else
17             // Own message, send to another node
18             processPacket(msg)
19     }
20 }
21 else {// other nodes
22     if (myAddress != msg -> getSrcAddr())
23         // Appropriate message, send to the core
24         send(msg, "localOut")
25     else
26         // Own message, send to another node
27         processPacket(msg)
28 }
```

El método **handleMessage**(cMessage ***msg**) se invoca con el mensaje, **msg** como parámetro, cada vez que el módulo recibe un mensaje o bien un automensaje. El método **handleMessage** del routing realiza dos acciones, como antes, una a través del método **processTimer** que implementa la maquina

finita de estados para simular la falla en el nodo manejada por auto-mensajes y el otro método **processPacket** que determina el enrutamiento de los mensajes, ninguno de estos métodos posee variaciones en su implementación. Sin embargo en los nodos DTN todos los mensajes que llegan al nodo y por ende al router son pasados primero al core para ser apropiados por el nodo y almacenados en el bundle, para luego ser despachados, siempre que el nodo no sea el destino del mensaje.
Esto último se evidencia en las lineas 11 del algoritmo, en donde, si el nodo está en el estado **UP** envía el mensaje al core por la puerta "*localOut*" en lugar de pasarlo directamente al destino según indique la tabla de enrutamiento. Y luego para para los nodos que no fallan, el cambio en la algoritmia se manifiesta en la linea 24 que causa el mismo efecto descripto en el párrafo anterior.

6.5.2. Core

El módulo simple "*core*" está implementado por la clase o tipo **Core**, este modulo simple se conecta al router ("*routing*") y al ("*bundle*"), el core se encarga de procesar los mensajes, es decir establecer como nuevo origen del mensaje su propia dirección y dejar el destino inalterado para ser reenviado. Luego el mensaje se envía por la puerta "*outStore*", al bundle, donde será almacenado y despachado al destino establecido. A demás, a partir del mensaje recibido el core genera un nuevo mensaje, de reconocimiento ("*acknowledgment*") que es remitido al origen desde el cual arribara el mensaje procesado, el cual es despachado directamente al router por la puerta "*outExit*".

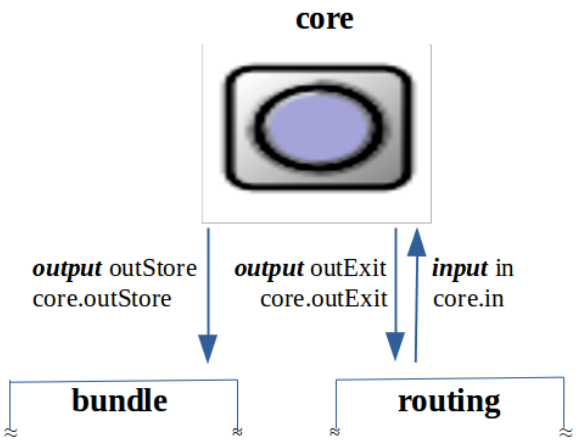


Figura 6.3: Esquema de comunicación del Modulo simple "*core*" con "*routing*" y "*bundle*".

En la figura 6.3 se esquematizan las puertas que unen a este módulo simple con los demás módulos del nodo. Los cambios con respecto al core empleado en TCP son los siguientes: se agrega la denominada puerta "*outStore*" que es una puerta que comunica al core con el bundle y es del tipo *"**output**"*, es decir que los mensajes apropiados y procesados por el core son enviados al bundle por esta puerta.
La clase **Core.cc** (en C++) representada en la figura 6.4 por el diagrama UML, muestra algunos cambios con respecto al core de la red TCP. Se disminuyen la cantidad de atributos, tanto el destinado a registrar el tiempo de demora de extremo a extremo como el atributo para contar la cantidad de saltos que realiza el mensaje hasta alcanzar su destino.
Sin embargo se agregan dos métodos nuevos **backwardMessage** y **forwardMessage** que son los encargados de generar y enviar el mensaje de reconocimiento ("*acknowledgment*") y el mensaje para el bundle respectivamente.

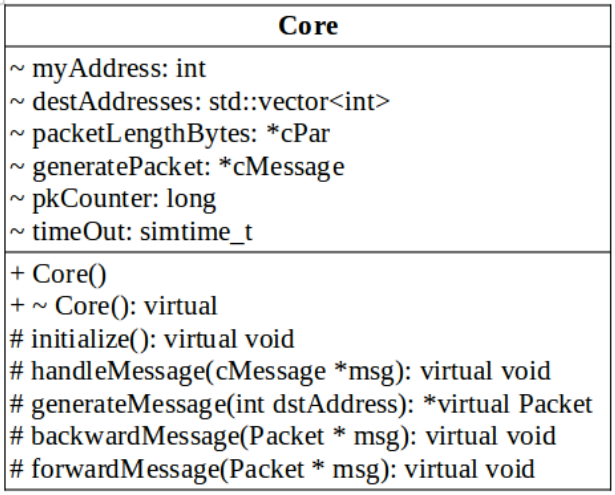


Figura 6.4: Modulo simple representado por la clase **Core**.

Como antes el método **initialize**() se invoca después de que OMNeT++ ha configurado la red (es decir, creado módulos y los ha conectado de acuerdo con las definiciones), en este caso el método no cambia, por lo que no se cree necesario volver a mostrarlo.

El método **handleMessage**(cMessage ***msg**) se invoca con el mensaje como parámetro cada vez que el módulo recibe un mensaje. Se espera que **handleMessage** procese el mensaje y luego lo devuelva.

Algorithm 6.2: handleMessage

Input : *msg
Output: void

```
1  if (msg == generatePacket) {
2      // Creation and sending message to "bundle"
3      int destAddress = destAddresses.size()-1
4      Packet *msg = generateMessage(destAddress)
5      send(msg, "outStore")
6      scheduleAt(simTime() + timeOut, generatePacket)
7  }
8  else {
9      // Handle incoming message
10     if (myAddress == destAddresses.size()-1) {
11         // If is a final node
12         backwardMessage(msg)
13         send(msg, "outStore")
14     }
15     else if (myAddress == msg->getDestAddr()) {
16         // Message is an "acknowledgment"
17         send(msg, "outStore")
18     }
19     else {
20         backwardMessage(msg)
21         forwardMessage(msg)
22     }
23 }
```

Este método considera dos casos, si el mensaje es un auto-mensaje, entonces se genera el mensaje nuevo y se envía al bundle.
El otro caso es cuando el mensaje es un mensaje que llega desde fuera del nodo, dentro de este caso hay que considerar tres casos más: Si el nodo es el nodo de destino (el nodo 3, para nuestro ejemplo de red) en cuyo caso se genera el mensaje de reconocimiento (con la función ***backwardMessage***) y el mensaje se envía al bundle con ***send***. El segundo caso es si el mensaje es un mensaje de reconocimiento que ha recibido el nodo, en cuyo caso se envía al bundle directamente y si el nodo fuera el 0 se genera un nuevo mensaje. El tercer y último caso es cuando el mensaje no es ni de reconocimiento ni auto-mensaje, en cuyo caso se lo apropia vía la función ***forwardMessage*** se lo remite al bundle y se envía reconocimiento a través de ***backwardMessage*** al origen desde donde proviene el mensaje.

El método **generateMessage**(int destAddress) es un método auxiliar usado en el método **handleMessage** que toma como argumento un entero, el destino del mensaje, y retorna un mensaje (o paquete). Es exactamente igual a la red TCP, por lo que no se muestra nuevamente.

El método **backwardMessage**(Packet * msg) es un método auxiliar usado en el método **handleMessage**, toma como argumento el mensaje recibido, y se encarga de generar una copia de este y entonces construir el mensaje de reconocimiento ("*acknowledgment*"), para luego enviarlo al nodo que originó el mensaje recibido a través del router.

Algorithm 6.3: backwardMessage

Input : msg
Output: void

```
1 // Generate a message or packet "acknowledgment"
2 int dstAddrs = msg->getSrcAddr()
3 Packet *copyBackward = msg ->dup()
4 // Set message attributes, number and send to "routing"
5 send(msg, "outExit")
```

El método **forwardMessage**(Packet * msg) es un método auxiliar usado en el método **handleMessage**, toma como argumento el mensaje recibido, y se encarga de apropiarse de dicho mensaje, esto es, establece el origen del mensaje como su propia dirección mientras que el destino permanece como el que trae originalmente el mensaje, luego se envía este mensaje modificado al bundle para que sea almacenado y desde ahí posteriormente reenviado.

Algorithm 6.4: forwardMessage

Input : msg
Output: void

```
1 // Generate a message or packet "acknowledgment"
2 int dstAddrs = msg->getDestAddr()
3 Packet *copyBackward = msg ->dup()
4 // Set message attributes, number and send to "routing"
5 send(msg, "outStore")
```

6.5.3. Bundle

El módulo simple "*bundle*" es implementado por la clase **Bundle**, este módulo está conectado al router y al core. Todos los mensajes llegan al bundle

provenientes desde el core y salen del bundle hacia el router, el objetivo de este módulo es simplemente almacenar y reenviar los mensajes nuevos que llegan al nodo y borrar aquellos mensajes almacenados para los cuales llega al nodo el mensaje de reconocimiento ("*acknowledgment*"), con el objeto de no almacenar mensajes indefinidamente. Como es el encargado de almacenar, enviar y borrar los mensajes este modulo debe registrar los saltos y retardos de los mensajes que reenvía y de los mensajes que borra respectivamente.

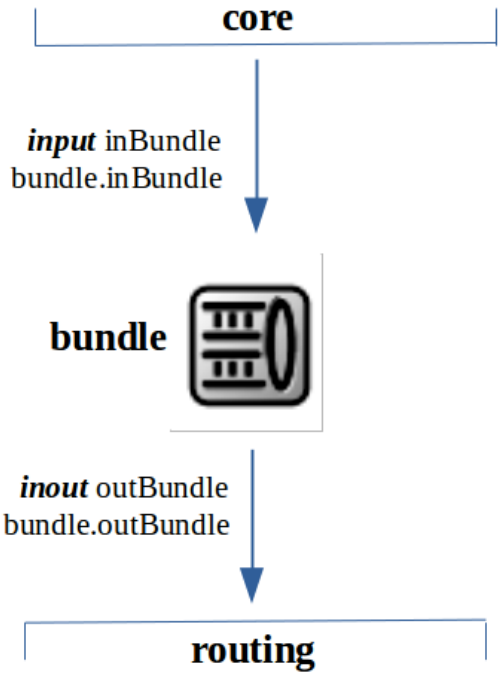


Figura 6.5: Esquema de comunicación del Modulo simple "*bundle*" con "*core*" y "*routing*".

En la figura 6.5 se esquematizan las puertas que unen a este módulo simple con el resto del nodo.
En este caso el módulo cuenta con 2 puertas únicamente, la denominada puerta "*inBundle*" es una puerta que comunica al core con el bundle y es del tipo *"**input**"*, es decir que los mensajes provenientes del core solo pueden entrar al bundle por esta puerta.
La denominada puerta "*outBundle*" es una puerta que comunica al bundle con el router y es del tipo *"**output**"*, es decir que los mensajes desde el bundle hacia router son enviados a través de esta puerta.

La clase **Bundle.cc** (en C++) representada en la figura 6.6 por el diagrama UML, muestra los atributos y métodos que la componen.

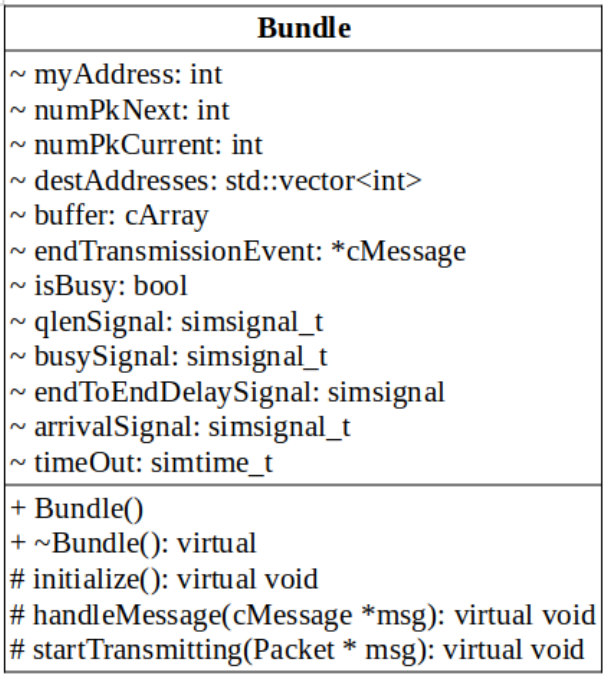


Figura 6.6: Modulo simple representado por la clase ***Bundle***.

Puesto que este módulo es el módulo donde se almacenan los mensajes para el posterior reenvío de los mismos, está implementado por un arreglo del tipo **cArray**. El atributo "*buffer*" del tipo cArray es una clase de contenedor que contiene objetos derivados de **cObject** propio del framework. cArray implementa una matriz de tamaño dinámico: su capacidad crece automáticamente cuando se llena. cArray almacena punteros de objetos insertados en lugar de hacer copias.
El atributo "*myAddress*", del tipo **int** (tipo heredado de C++) determina el índice (o dirección) del nodo en la red. "*destAddresses*" es un vector de enteros (tipo heredado de C++) que contiene los lugares de cada nodo en la red.
El módulo también puede implementar eventos como "tiempo de espera expirado" enviándose un auto-mensaje, este el caso del atributo "*endTransmissionEvent*" que es usado como auto-mensaje en el módulo.
luego siguen una serie de atributos como "*qlenSignal*" del tipo **simsignal_t** todos ellos propios del framework OMNeT++, estos últimos atributos del tipo **simsignal_t** son usados para registrar, guardar y luego procesar ciertos eventos de interés.
Por último los atributos "*endToEndDelaySignal*" y "*arrivalSignal*" son del

tipo **simsignal_t** propio de OMNeT++ y son los encargados de grabar los resultados del tiempo de demora de extremo a extremo y la cantidad de saltos de cada mensaje respectivamente. Mientras que el atributo "*timeOut*" almacena el valor ideal del tiempo de demora de un mensaje (de 2 ms) en ser enviado al nodo contiguo y recibir su mensaje de reconocimiento.
A continuación se describe en seudo código las funcionalidades de cada método de la clase, las funciones resaltadas en negrita pertenecen al framework OMNeT++ o son llamadas a otros métodos.

Como antes el método **initialize**() se invoca después de que OMNeT++ ha configurado la red (es decir, creado módulos y los ha conectado de acuerdo con las definiciones), en este caso el método debe configurar las direcciones de cada nodo y por ende de cada bundle.

Algorithm 6.5: initialize

Input : void
Output: void

```
1  // Initialize attributes of class
2  myAddress = getParentModule()->par("address")
3  buffer.setName("buffer")
4  endTransmissionEvent = new cMessage("endTxEvent")
5  // The "bundle" indices is built
6  // It considers the input string to consist of tokens, separated by one
7  // or more delimiter characters
8  const char *destAddressesPar = &par("destAddresses")
9  cStringTokenizer tokenizer(destAddressesPar)
10 const char *token
11 int k = 0
12 while ((token = tokenizer.nextToken()) != nullptr) {
13     destAddresses.push_back(atoi(token))
14     k++
15 }
```

El método **handleMessage**(cMessage ***msg**):

Algorithm 6.6: handleMessage

Input : *msg
Output: void

```
1  if (msg == endTransmissionEvent) {
2      // Transmission finished, we can start next one
3      if (buffer.size() == 0)
4          isBusy = false
5      else { // Resending msg in Array
6          for (index = 0 to buffer.size()-1) {
7              msg = buffer[index]
8              Packet *copy = msg->dup()
9              startTransmitting(copy)
10         }
11     }
12 else { // Arrived on gate "inBundle" from "core"
13     if (myAddress == destAddresses.size()-1) { // Final node
14         int numPk = msg->getNumberPk()
15         if (numPk == numPkCurrent)
16             // Start with the current message
17             int index = buffer.add(msg)
18         else if (numPk == numPkNext) {
19             // Continue with the next message
20             for (i = 0 to buffer.size()-1) {
21                 int index = buffer.find("name_message")
22                 Packet *pkOld = buffer[index]
23                 delete buffer.remove(index)
24             }
25             int indexNew = buffer.add(msg)
26             numPkCurrent++
27             numPkNext++
28         }
29     }
30     else if (myAddress == msg->getDestAddr()) {
31         // arrived an "acknowledgment"
32         if (buffer.size() != 0) {
33             cancelEvent(endTransmissionEvent)
34             int index = buffer.find("name_message")
35             delete buffer.remove(index)
36         }
37     }
38     else { // arrived a new message
39         cancelEvent(endTransmissionEvent)
40         // Insert message in buffer
41         int index = buffer.add(msg)
42         scheduleAt(simTime(), endTransmissionEvent)
43     }
44 }
```

Este método se invoca con el mensaje como parámetro cada vez que el módulo recibe un mensaje. Se espera que **handleMessage** procese el mensaje y luego lo devuelva.
Este método considera dos casos, uno si el mensaje es un auto-mensaje entonces debe chequear el estado del buffer, si esta vacío no hace nada y si hubiera mensajes para se enviado realiza un ciclo de copia de cada uno y los envía a todos simultáneamente, usando la función ***startTransmitting***.
El otro caso es que el mensaje sea proveniente desde el core (no sea un auto-mensaje), se deben considerar tres casos dentro de esta posibilidad:

1. Si el nodo en cuestión es el nodo de destino, es decir el nodo 3, compara el número del mensaje con el atributo "*numPkCurrent*" que lleva en cuenta el número corriente del mensaje, ambos números comienzan cero, si son iguales lo almacena en buffer. Si el mensaje entrante posee el número que concuerda con el atributo "*numPkNext*", es decir es el mensaje siguiente, guarda los valores de interés del mensaje anterior (los atributos "*endToEndDelaySignal*" y "*arrivalSignal*") y borra el mensaje. A continuación guarda el mensaje nuevo en el buffer. Y actualiza los atributos "*numPkCurrent*" y "*numPkNext*".

2. Si el mensaje es un mensaje de reconocimiento, lo busca en el buffer y lo borra.

3. Si el mensaje es un mensaje nuevo se lo agrega al buffer y se dispara automáticamente un timer con el auto-mensaje "*endTransmissionEvent*".

El método **startTransmitting**(Packet ***msg**) es un método auxiliar usado en el método **handleMessage** que toma como argumento el mensaje (o paquete) y lo transmite por la puerta de salida "*outBundle*" del modulo simple bundle hacia el router.

Algorithm 6.7: startTransmitting

Input : msg
Output: void

```
1 // Starting transmission of message
2 isBusy = true
3 cancelEvent(endTransmissionEvent)
4 send(msg, "outBundle")
5 simtime_t endTransmission = simTime() + 0.002
6 scheduleAt(endTransmission, endTransmissionEvent)
```

Al igual que en el caso de la red TCP, con las implementaciones C++ anteriores, el simulador OMNET++ puede capturar las estadísticas del comportamiento relevantes para este trabajo. Dichas estadísticas se mostrarán en el capítulo 7.

Capítulo 7

Resultados de las Simulaciones

En este capítulo se analiza la efectividad de los modelos propuestos. El experimento se realizó en CPU(Intel$^{(R)}$ Core$^{(TM)}$ i7-8700k CPU @ 3.70GHz x 12)
La función de registro de eventos y las herramientas relacionadas se han agregado en OMNeT++ con el objetivo de ayudar al usuario a comprender modelos de simulación complejos e implementar correctamente los comportamientos de componentes deseados. Usando estas herramientas, uno puede examinar los detalles del historial registrado de una simulación, centrándose en el comportamiento, antes de los resultados estadísticos. Posteriormente y con mayor seguridad se analizan los resultados estadísticos.

7.1. Red TCP

El archivo de registro de eventos se crea automáticamente durante una ejecución de simulación a solicitud explícita configurable en el archivo *ini*. El archivo resultante se puede ver en el IDE de OMNeT++ utilizando el Gráfico de secuencia y la Tabla de registro de eventos o se puede procesar mediante la herramienta de registro de eventos de línea de comando.
Estas herramientas permiten filtrar los datos recopilados para permitirle centrarse en eventos relevantes para lo que está buscando. Permiten examinar las relaciones de causalidad y proporcionan filtros basados en tiempos de simulación, números de eventos, módulos y mensajes.
Usando esta última forma de visualizar los mensajes, como se muestra en la figura 7.1, se observa a modo de ejemplo cómo en la red TCP, se emite un mensaje del nodo 0 al nodo 3 (circulo rojo 1). Si un mensaje encuentra un nodo (en nuestro caso el enrutador del nodo 2) en su estado inactivo (circulo rojo 2), no puede avanzar y el mensaje se destruye (o pierde). Dado que el

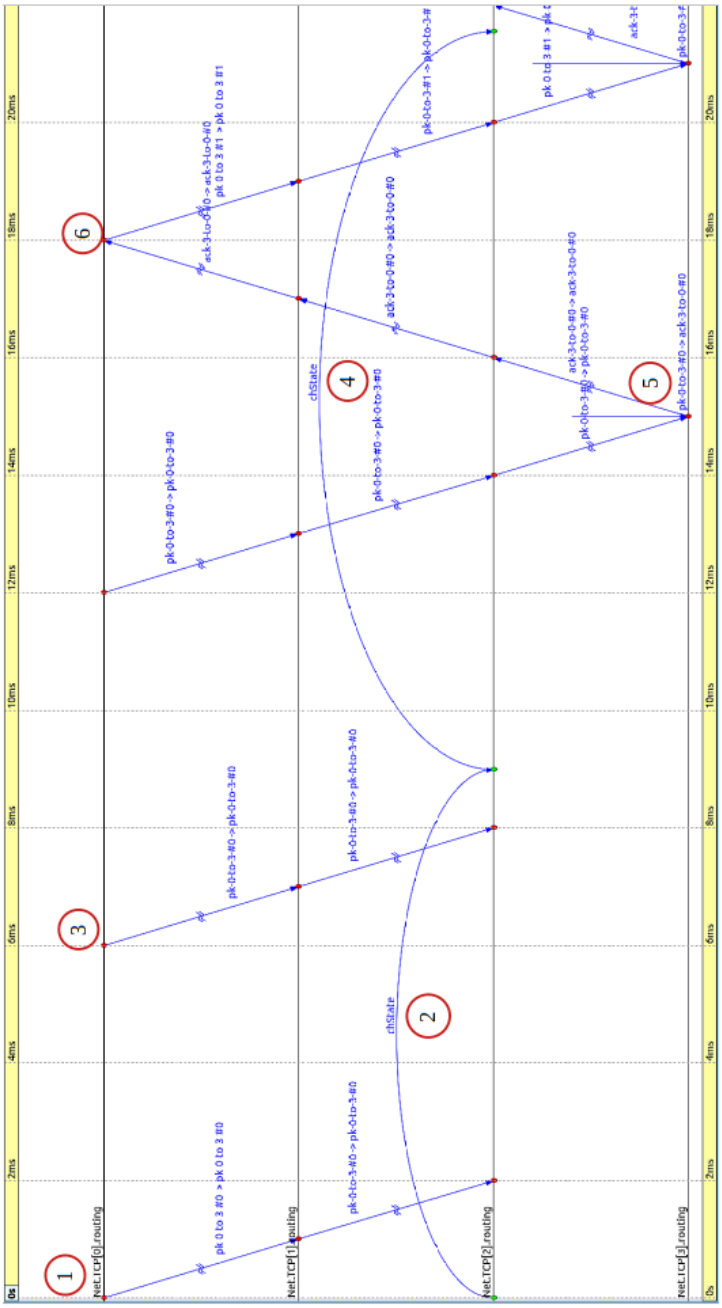


Figura 7.1: Mensajes en la interfaz gráfica de usuario “Eventlog”. Red TCP.

nodo 0 de la red TCP tiene un contador de tiempo de espera interno (un auto-mensaje), puede detectar si el último mensaje no llegó en el tiempo esperado y puede emitir el mismo mensaje nuevamente (circulo rojo 3), para el ejemplo de la figura el estado "Down" del enrutador 2 es lo suficientemente largo (9 milisegundos) como para que se pierda el segundo envío del mismo mensaje. Una vez que llega al nodo 3 (circulo rojo 5), dado que ahora el enrutador 2 se encuentra en un estado "Up" (circulo rojo 4) el mensaje puede continuar, y un acuse de recibo regresa del nodo 3 al nodo 0 (circulo rojo 6). De esta manera, el nodo 0 de TCP puede saber que su mensaje llegó con éxito a su destino. Por lo tanto, en la red TCP, el retardo ideal de extremo a extremo es de 6 milisegundos y el conteo de saltos ideal es de 6 saltos. Si hubiere una demora como en el ejemplo de muestra, los retrasos de extremo a extremo y los conteos de saltos se acumulan continuamente, hasta que el mensaje finalmente completa su recorrido.

7.1.1. Retardo y números de saltos en TCP

Para comprender los resultados de las figuras 7.2 y 7.3, debemos entender como se obtuvieron dichos resultados, a saber:

1. El tiempo medio de permanencia en el estado Down (o tiempo medio de inactividad del sistema que posee la falla), representado por $1/\lambda_{down}$ se mantiene fijo ($1/\lambda_{down} = 10ms$). Esto significa que el tiempo de duración de cada periodo de inactividad será una variable aleatoria con distribución exponencial (como se explicó en el capítulo 2) con tiempo medio fijo.

2. Los tiempos medios de permanencia en el estado Up (o tiempo medio de actividad del sistema que posee la falla), representado por $1/\lambda_{up}$, varia en el intervalo [$10ms$, $20ms$, .., $100ms$]. Esto significa que el tiempo de duración de cada periodo de actividad será una variable aleatoria con distribución exponencial con tiempo medio fijo.

3. Una simulación se ejecuta con una semilla en particular, la duración de cada simulación es de $900s$, y con un valor determinado para el cociente $\lambda_{down}/\lambda_{up} \in$ [1, 2, .., 10]. Los resultados de una simulación los llamamos ***endToEndDelay*** y ***hopCount***, son los promedio de retardos y saltos respectivamente, de todos los paquetes enviados durante la duración de una simulación.

4. Para cada valor de la relación $\lambda_{down}/\lambda_{up} \in$ [1, 2, .., 10], se realizan 10 simulaciones, cada una con distintas semillas. Luego se realiza una

estadística (promedio y desviación estándar), esto arroja como resultado cada punto de la gráfica que llamamos ***E2ED:mean*** y ***hopCount:mean***, con sus respectivas barras de error.

Gráficas cuya falla se encuentra en el nodo 2

Resultados típicos encontrados por la simulación, para las demoras de extremo a extremo y saltos respectivamente en los valores limites del intervalo, cuando la falla se establece en el nodo 2:

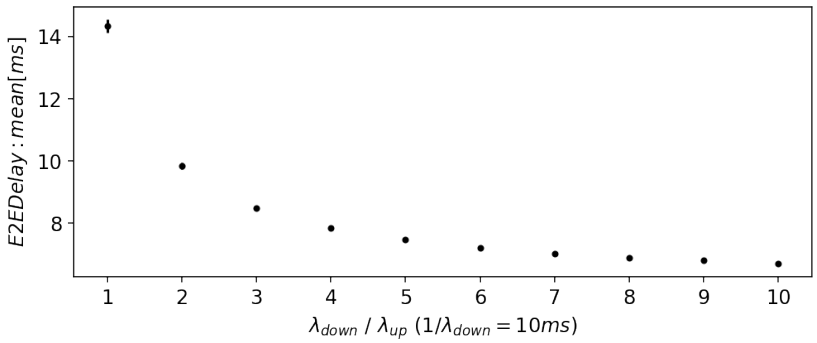


Figura 7.2: Demora de extremo a extremo en ms versus relación de tiempo medio de actividad/tiempo medio de inactividad (del nodo 2).

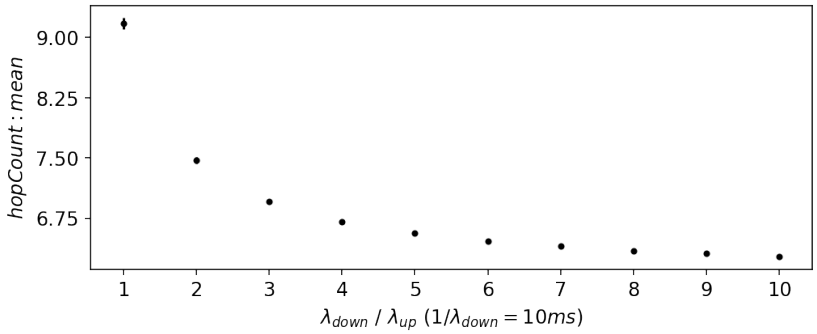


Figura 7.3: Saltos medios de extremo a extremo versus relación de tiempo de actividad/tiempo de inactividad (del nodo 2).

E2ED:mean$(1) = (14{,}34 \pm 0{,}07)ms$,
E2ED:mean$(10) = (6{,}71 \pm 0{,}01)ms$,
hopCount:mean$(1) = (9{,}18 \pm 0{,}03)$ saltos,
hopCount:mean$(10) = (6{,}275 \pm 0{,}004)$ saltos.

En la gráfica se colocaron como barras de error los valores de 3σ, sin embargo son apenas apreciables.

Gráficas cuya falla se encuentra en el nodo 1

Las gráficas que se muestran a continuación, son también resultados de simulaciones realizadas sobre la red TCP, pero en esta ocasión situamos la falla sobre el nodo 1.

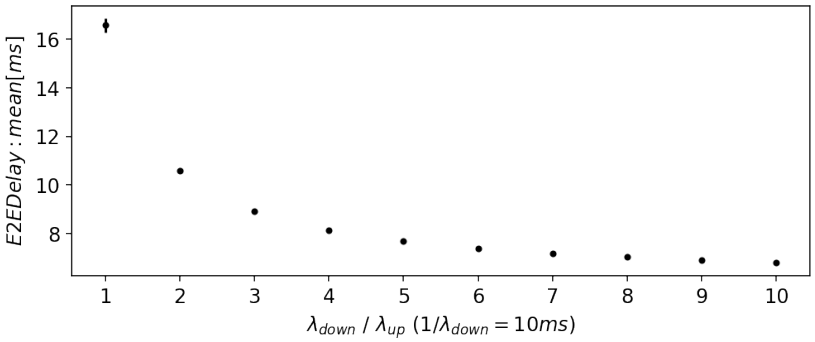


Figura 7.4: Demora de extremo a extremo en ms versus relación de tiempo medio de actividad/tiempo medio de inactividad (del nodo 1).

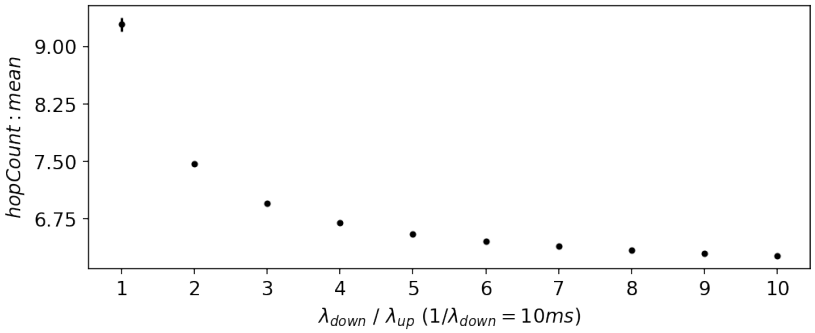


Figura 7.5: Saltos medios de extremo a extremo versus relación de tiempo de actividad/tiempo de inactividad (del nodo 1).

E2ED:mean$(1) = (16{,}6 \pm 0{,}1)ms$,
E2ED:mean$(10) = (6{,}81 \pm 0{,}01)ms$,
hopCount:mean$(1) = (9{,}29 \pm 0{,}03)$ saltos,
hopCount:mean$(10) = (6{,}268 \pm 0{,}004)$ saltos.

En la gráfica se colocaron como barras de error los valores de 3σ, al igual que en el caso anterior los errores estadísticos siguen siendo prácticamente inapreciables.

7.2. Red DTN

El archivo de registro de eventos de la figura 7.6 se crea automáticamente durante la ejecución de una simulación.
Con el objeto de visualizar las diferencias entre ambas redes y por tanto para observar sus diferentes implementaciones, es que se tomaron las mismas condiciones de simulación que para el caso anterior mostrado de la red TCP, figura 7.1. El caso que se muestra es para la condición $\lambda_{down}/\lambda_{up} = 1$.
En la red DTN, también se emite un mensaje del nodo 0 al nodo 3 (circulo rojo 1). Pero los nodos DTN tienen un esquema particular de almacenamiento y reenvío, el modulo simple "bundle" explicado en el capitulo anterior. De esta manera, los nodos pueden recibir simultáneamente mensajes de sus nodos vecinos mientras mantienen mensajes anteriores en sus "bundles", el nodo 1 se apropia del mensaje enviado por el nodo 0 (circulo rojo 2) lo transmite al nodo 2 y a su vez envía un mensaje de reconocimiento al nodo 0 (circulo rojo 3).
El nodo 0 emite y envía un nuevo mensaje al nodo 3 (circulo rojo 5). En caso de fallas, si, por ejemplo, un mensaje emitido desde el nodo 0 encuentra al nodo 2 en su estado inactivo, es decir, el enrutador 2 en estado inactivo (circulo rojo 4), entonces el mensaje se perderá, pero este mensaje permanece en el "bundle" correspondiente al nodo 1 hasta que el enrutador 2 regrese a su estado activo y puede enviarlo al nodo siguiente 3. Sin embargo el nodo 1, no reenvía solo el mensaje original perdido, sino que envía todos los mensajes que hubieran arribado a este nodo, (circulo rojo 6) se ve al nodo 1 enviando dos mensajes al nodo 2.

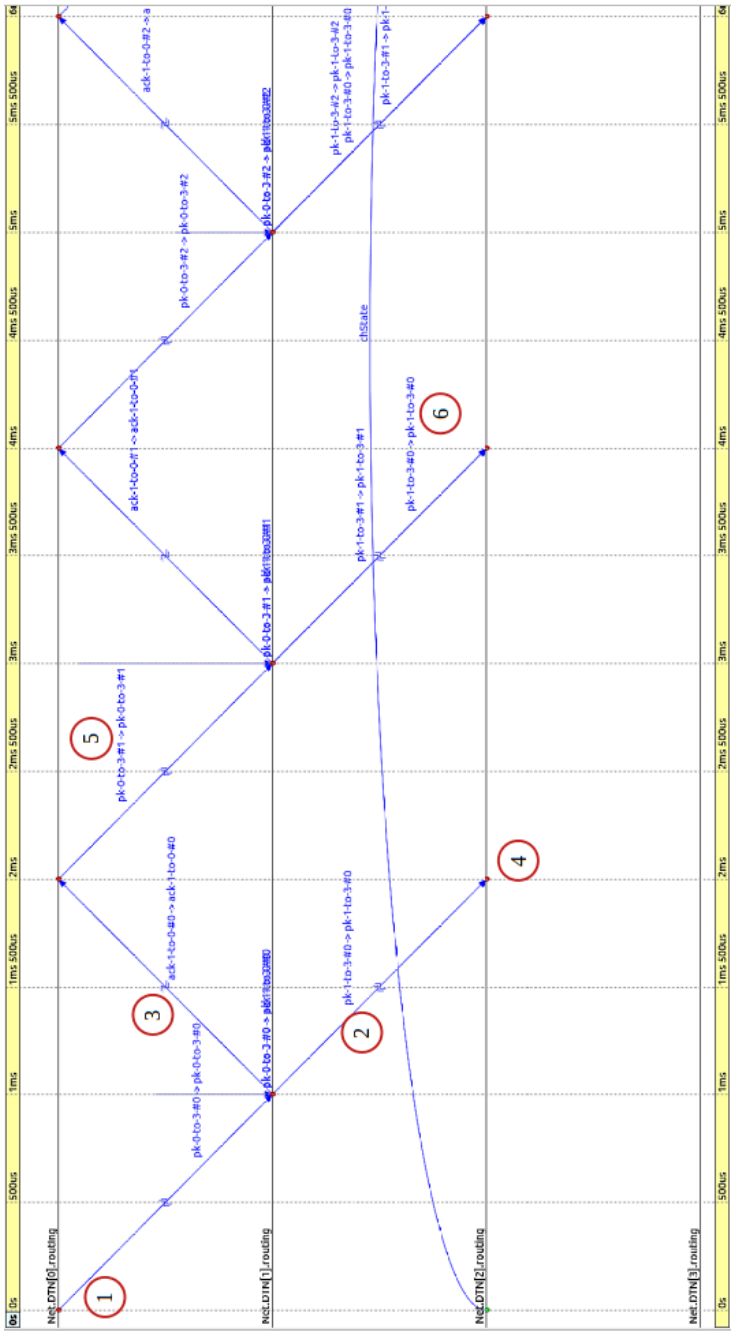


Figura 7.6: Mensaje en la interfaz gráfica de usuario “Eventlog” en la red DTN.

7.2.1. Retardo y números de saltos en DTN

El retraso del canal, en esta red, también es de un milisegundo, por lo tanto el retraso ideal en la red DTN es de 3 milisegundos en 3 saltos. (Se debe tener en cuenta que las redes DTN son dos veces más rápidas que la red TCP en condiciones ideales).
Los resultados de las figuras 7.9 y 7.10, se obtuvieron bajo las mismas condiciones de simulación que para la red TCP, las reiteramos nuevamente con el afán de esclarecer que las condiciones son exactamente las mismas y lo único que varia es el funcionamiento interno de la red, a saber:

1. El tiempo medio de permanencia en el estado Down (o tiempo medio de inactividad del sistema que posee la falla), representado por $1/\lambda_{down}$ se mantiene fijo ($1/\lambda_{down} = 10ms$). Esto significa que el tiempo de duración de cada periodo de inactividad será una variable aleatoria con distribución exponencial (como se explicó en el capítulo 2) con tiempo medio fijo.

2. Los tiempos medios de permanencia en el estado Up (o tiempo medio de actividad del sistema que posee la falla), representado por $1/\lambda_{up}$, varia en el intervalo [$10ms$, $20ms$, .., $100ms$]. Esto significa que el tiempo de duración de cada periodo de actividad será una variable aleatoria con distribución exponencial con tiempo medio fijo.

3. Una simulación se ejecuta con una semilla en particular, la duración de cada simulación es de $900s$, y con un valor determinado para el cociente $\lambda_{down}/\lambda_{up} \in$ [1, 2, .., 10]. Los resultados de una simulación los llamamos ***endToEndDelay*** y ***hopCount***, son los promedio de retardos y saltos respectivamente, de todos los paquetes enviados durante la duración de una simulación.

4. Para cada valor de la relación $\lambda_{down}/\lambda_{up} \in$ [1, 2, .., 10], se realizan 10 simulaciones, cada una con distintas semillas. Luego se realiza una estadística (promedio y desviación estándar), esto arroja como resultado cada punto de la gráfica que llamamos ***E2ED:mean*** y ***hopCount:mean***, con sus respectivas barras de error.

Gráficas cuya falla se encuentra en el nodo 2

Resultados típicos encontrados por la simulación, para las demoras de extremo a extremo y saltos respectivamente en los valores limites del intervalo, cuando la falla se establece en el nodo 2:

E2ED:mean$(1) = (10{,}07 \pm 0{,}06)ms$,
E2ED:mean$(10) = (4{,}19 \pm 0{,}03)ms$,
hopCount:mean$(1) = (6{,}54 \pm 0{,}03)$ saltos,
hopCount:mean$(10) = (3{,}59 \pm 0{,}02)$ saltos.

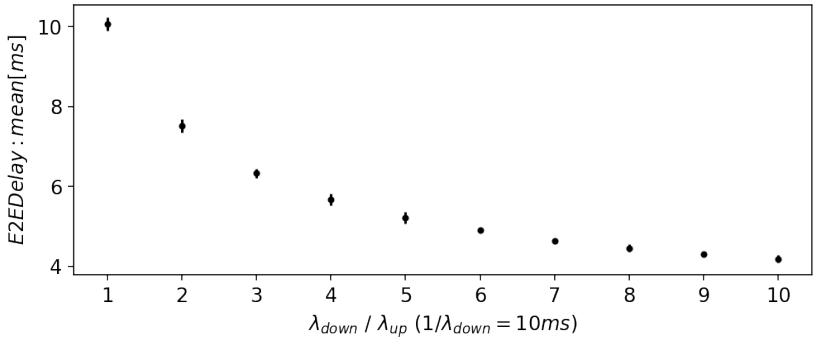


Figura 7.7: Demora de extremo a extremo en segundos versus relación de tiempo medio de actividad/tiempo medio de inactividad.

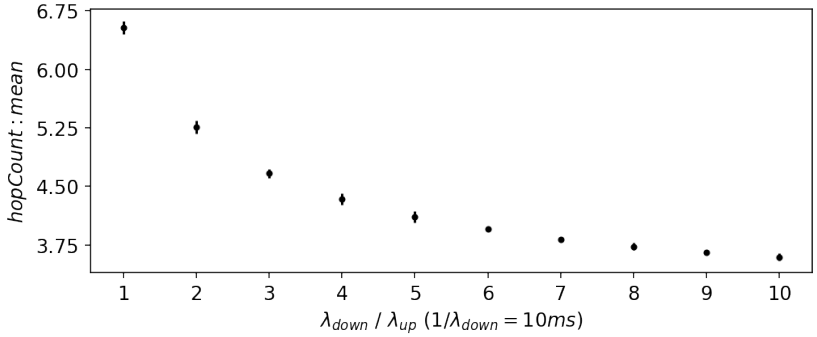


Figura 7.8: Saltos medios de extremo a extremo versus relación de tiempo medio de actividad/tiempo medio de inactividad.

En la gráfica se colocaron como barras de error los valores de 3σ, sin embargo son apenas apreciables.

Gráficas cuya falla se encuentra en el nodo 1

Resultados típicos encontrados por la simulación, para las demoras de extremo a extremo y saltos respectivamente en los valores limites del intervalo, cuando la falla se establece en el nodo 2:
E2ED:mean$(1) = (10{,}06 \pm 0{,}04)ms$,

E2ED:mean$(10) = (4{,}19 \pm 0{,}03)ms$,
hopCount:mean$(1) = (7{,}03 \pm 0{,}02)$ saltos,
hopCount:mean$(10) = (3{,}69 \pm 0{,}02)$ saltos.

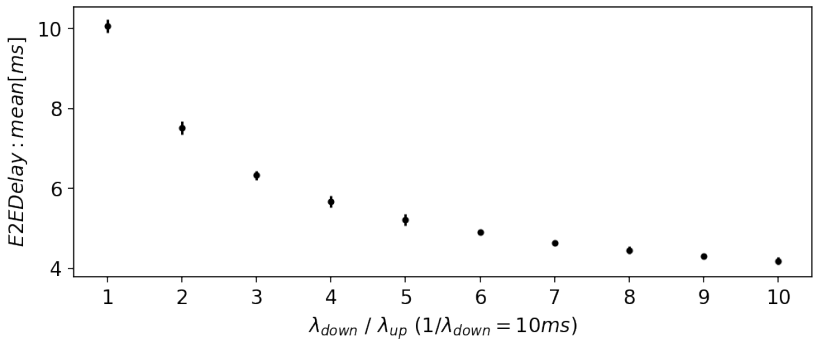


Figura 7.9: Demora de extremo a extremo en segundos versus relación de tiempo de actividad/tiempo de inactividad para la red DTN. ($\lambda_{down} = 10ms$)

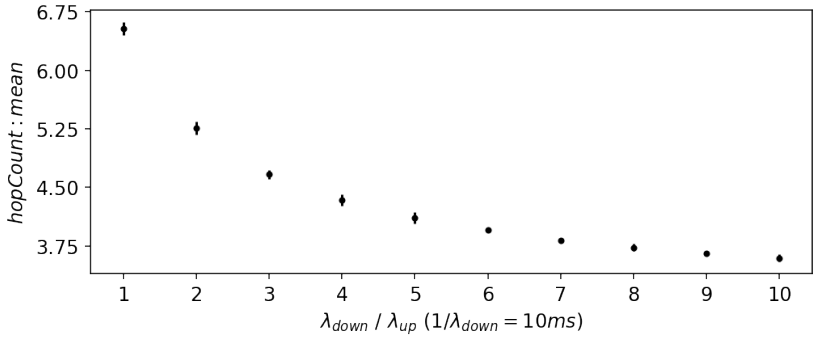


Figura 7.10: Saltos medios de extremo a extremo versus relación de tiempo medio de actividad/tiempo medio de inactividad para la red DTN.

En la gráfica se colocaron como barras de error los valores de 3σ, sin embargo son apenas apreciables.

7.2.2. Acumulación de paquetes debido a fallas

Fallas en el nodo 2, almacenamiento en el nodo 1

Las figuras 7.11 y 7.12 muestran los resultados obtenidos para el tamaño medio del buffer del bundle del nodo 1 y los valores máximos en cantidad de paquetes (o mensajes).

sizeBundle[1]:mean$(1) = (4{,}04 \pm 0{,}03)$ paquetes,
sizeBundle[1]:mean$(10) = (1{,}50 \pm 0{,}01)$ paquetes,
sizeBundle[1]:max$(1) = (63 \pm 9)$ saltos,
sizeBundle[1]:max$(10) = (47 \pm 3)$ saltos.

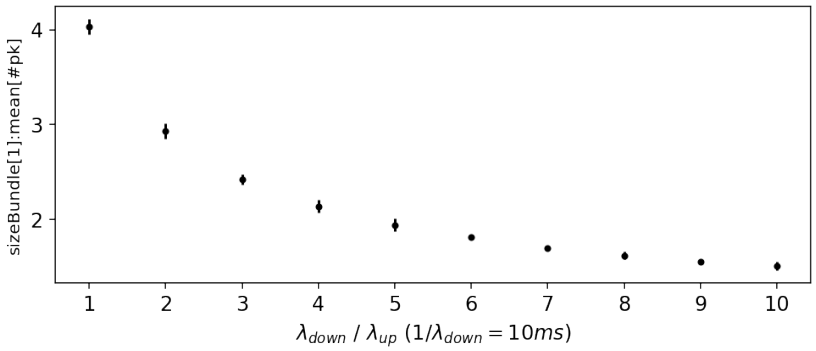


Figura 7.11: Tamaño medio del buffer del bundle versus relación de tiempo medio de actividad/tiempo medio de inactividad.

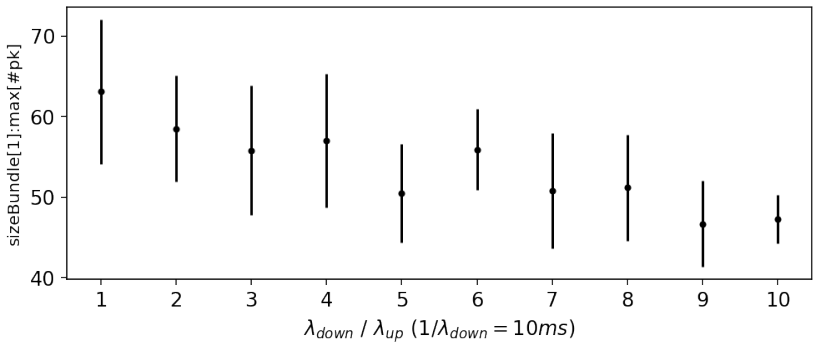


Figura 7.12: Tamaños medios de los valores máximos del bundle del nodo 1 versus relación de tiempo medio de actividad/tiempo medio de inactividad.

En la gráfica se colocaron las barras de error, σ, la gráfica correspondiente al numero máximo de paquetes almacenados en el buffer del nodo anterior al que posee la falla son notables.

Fallas en el nodo 1, almacenamiento en el nodo 0

Las figuras 7.13 y 7.14 muestran los resultados obtenidos para el tamaño medio del buffer del bundle del nodo 0 y los valores máximos en cantidad de paquetes (o mensajes) respectivamente versus la relación entre los tiempos

medios de actividad y tiempo medios de inactividad, $\lambda_{down}/\lambda_{up}$, siempre bajo las mismas condiciones de simulación.

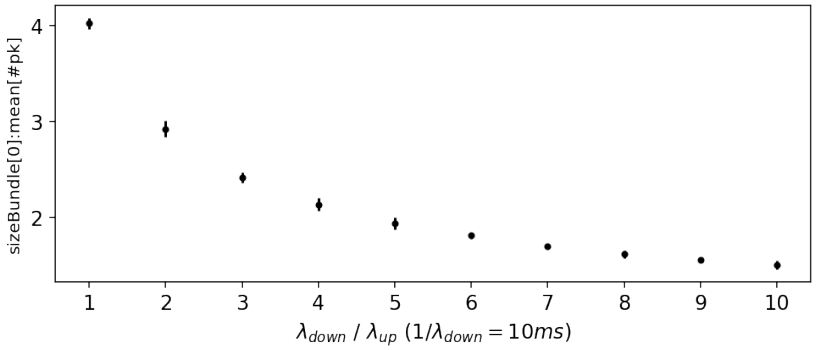


Figura 7.13: Tamaño medio del buffer del bundle del nodo 0 versus relación de tiempo medio de actividad/tiempo medio de inactividad.

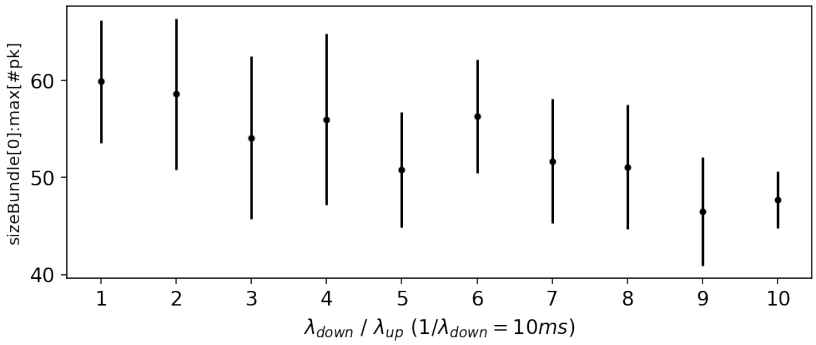


Figura 7.14: Tamaños medios de los valores máximos del bundle del nodo 0 versus relación de tiempo de actividad/tiempo de inactividad.

sizeBundle[0]:mean$(1) = (4{,}02 \pm 0{,}02)$ paquetes,
sizeBundle[0]:mean$(10) = (1{,}50 \pm 0{,}01)$ paquetes,
sizeBundle[0]:max$(1) = (60 \pm 6)$ saltos,
sizeBundle[0]:max$(10) = (48 \pm 3)$ saltos.

En la gráfica se colocaron las barras de error, σ, la gráfica correspondiente al numero máximo de paquetes almacenados en el buffer del nodo anterior al que posee la falla son notables.

Capítulo 8

Conclusiones y Trabajos Futuros

Los resultados anteriores revelan que la penalización del rendimiento puede ser inaceptable si los tiempos de actividad y los tiempos de inactividad de los nodos son similares.

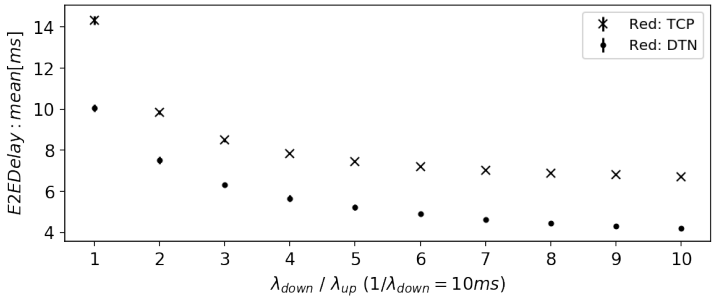


Figura 8.1: Demora de extremo a extremo. ($1/\lambda_{down} = 10ms$)

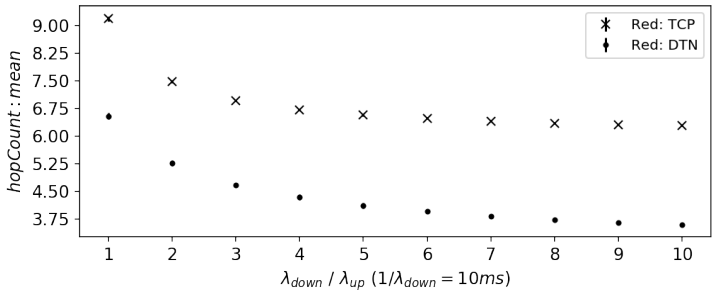


Figura 8.2: Saltos medios de extremo a extremo. ($1/\lambda_{down} = 10ms$)

Sin embargo, si los tiempos de actividad son entre uno y dos órdenes de magnitud mayores que los tiempos de inactividad, las métricas de rendimiento están cerca del caso ideal. Es muy importante señalar que en los casos estudiados, las redes DTN pueden ser hasta dos veces más rápidas que las redes TCP y tienen la propiedad deseada de degradación ("graceful degradation"). Se puede observar que la distribución (barras de error) de los resultados es mayor en la red DTN.
Estas conclusiones también alientan una mayor investigación para mejorar tanto la fiabilidad como el rendimiento en este tipo de sistemas. Además, muestran que la precisión de la predicción de tiempos de actividad y tiempos de inactividad debe mejorarse para poder utilizarlos de manera segura. El trabajo futuro mejorará estos resultados.
En cuanto a la cantidad de memoria necesaria en el buffer del bundle del nodo 1, como así también en el nodo 0, quienes son los protagonista de la reparación del sistema reenviando los mensajes al nodo 2 y al nodo 1 respectivamente, podemos observar de las figuras 7.11 y sobre todo de la figura 7.12 que la necesidad de memoria para ese buffer es a lo sumo $100 * 16bytes = 1600bytes$, ya que estamos suponiendo que cada paquete tiene un tamaño de 16 bytes (una linea de chache) por ejemplo. Lo cual nos hace pensar que es un sistema factible de implementar por el bajo requerimiento de recursos.

Como se comento oportunamente en el capitulo 2, decíamos que el diseñador de la red debe trabajar dentro de las limitaciones tecnológicas para implementar la topología, el enrutamiento y el control de flujo de la red. Y que una clave para la eficiencia de las redes de interconexión proviene del hecho de que los recursos de comunicación se comparten.
El patrón de conexión de estos nodos define la topología de la red. Una buena topología explota las propiedades de la tecnología de empaquetado de la red, para maximizar el ancho de banda de la red.
El framework usado permite explorar distintas topologías, como de anillo, de malla, etc, una vez que se ha elegido una topología, puede haber muchas rutas posibles (secuencias de nodos y canales) que un mensaje podría tomar a través de la red para llegar a su destino. El enrutamiento determina cuál de estas posibles rutas toma realmente un mensaje.
Para ello se deberían implementar algoritmos de enrutamiento bastantes diferentes del clásico de "los caminos mas cortos" usado en el presente trabajo, implicando un cambio desde luego en las métricas para caracterizar las redes de estudio, esto a su vez, generaría una extensión al trabajo que no se hallaba programada. Sin embargo esta problemática está en los planes inmediatos posteriores a la conclusión de este trabajo de tesis, más aun alentados por los resultados encontrados.

Este trabajo ha logrado consolidar una gran motivación en varias líneas de investigación y desarrollo asociadas a las WNOCs y WNBCs y las arquitecturas segmentadas. Una de ellas es el desarrollo a nivel micro electrónico de SOCs que implementen esta tecnología por medio de herramientas de software libre. Existen actualmente ecosistemas que permiten el desarrollo y hasta la fabricación de SOCs con herramientas de software libre, entre los que se destaca *"efabless"*[1]. El desarrollo de productos, herramientas y servicios dentro de la filosofía de trabajo *"efabless"* es sin lugar a dudas una oportunidad muy relevante para el crecimiento de la industria nacional pero con perspectivas de aplicación tanto nacionales como internacionales. Las posibilidades de generar productos y servicios "virtuales" en el área de la microelectrónica son prácticamente infinitas. Desde la perspectiva y dimensión académica, la realización de estas actividades le permitirá a la Universidad Nacional de Córdoba, interactuar más asiduamente con el medio tanto público como privado usando como herramientas a los proyectos de extensión. Desde la perspectiva industrial se puede asegurar que el nuevo modelo de negocio basado en la filosofía de trabajo "efabless" permitirá la existencia de una comunidad de desarrolladores locales ofrecer sus productos y servicios, de muy alto nivel tecnológico y por tanto de gran valor agregado tanto en el contexto nacional como internacional. Resulta muy atractivo pensar que esta comunidad de desarrolladores locales necesitan una muy alta calificación científica, tecnológica y académica pero sólo una computadora y herramientas de software libre para desarrollar sus actividades. Esta característica es ideal para una economía como la Argentina. El desarrollo de la tecnología micro electrónica, está normalmente asociado al uso de herramientas de software de un costo excesivamente elevado lo que desalienta rápidamente a los potenciales profesionales independientes e incluso a los laboratorios y otras organizaciones públicos o privados a ingresar en esta área. Sin embargo, el uso de las herramientas de software libre abre un panorama mucho más prometedor. Una vertiente de trabajos similar a la que fue en su momento la de los sistemas operativos de código abierto, se está queriendo empezar a formar en el área de la microelectrónica. El éxito de esta nueva corriente, descansa en la generación de una masa crítica de desarrolladores altamente capacitados para moverse con solvencia y comodidad en ecosistemas como "efabless".
Este trabajo final es sin lugar a dudas una parte fundamental de los cimientos necesarios para llevar adelante esta transformación.

[1] https://www.efabless.com/

Bibliografía

[1] Koren and Su, "Reliability Analysis of N-Modular Redundancy Systems with Intermittent and Permanent Faults," in IEEE Transactions on Computers, vol. C-28, no. 7, pp. 514-520, July 1979, doi: 10.1109/TC.1979.1675397.

[2] E. P. Kim and N. R. Shanbhag, "Soft N-Modular Redundancy," in IEEE Transactions on Computers, vol. 61, no. 3, pp. 323-336, March 2012, doi: 10.1109/TC.2010.253.

[3] T. J. Dysart and P. M. Kogge, "Reliability Impact of N-Modular Redundancy in QCA," in IEEE Transactions on Nanotechnology, vol. 10, no. 5, pp. 1015-1022, Sept. 2011, doi: 10.1109/TNANO.2010.2099131.

[4] R. Johansson, "Two error-detecting and correcting circuits for space applications," Proceedings of Annual Symposium on Fault Tolerant Computing, Sendai, Japan, 1996, pp. 436-439, doi: 10.1109/FTCS.1996.534630.

[5] E. Khan, S. Lehmann, H. Gunji and M. Ghanbari, "Iterative error detection and correction of H.263 coded video for wireless networks," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 14, no. 12, pp. 1294-1307, Dec. 2004, doi: 10.1109/TCSVT.2004.837018.

[6] A. Veneris and I. N. Hajj, "A hybrid approach to design error detection and correction [VLSI digital circuits]," ICECS'99. Proceedings of ICECS '99. 6th IEEE International Conference on Electronics, Circuits and Systems (Cat. No.99EX357), Pafos, Cyprus, 1999, pp. 347-350 vol.1, doi: 10.1109/ICECS.1999.812294.

[7] G. de M. Borges, L. F. Gonçalves, T. R. Balen and M. S. Lubaszewski, "Increasing reliability of programmable mixed-signal systems by applying design diversity redundancy," 2010 15th IEEE European Test Symposium, Praha, 2010, pp. 261-261, doi: 10.1109/ETSYM.2010.5512730.

[8] H. Chang and J. He, "Structure diversity design of analog circuits by evolutionary computation for fault-tolerance," 2012 International Conference on Systems and Informatics (ICSAI2012), Yantai, 2012, pp. 828-832, doi: 10.1109/ICSAI.2012.6223137.

[9] I. Gashi, P. Popov and L. Strigini, "Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers," in IEEE Transactions on Dependable and Secure Computing, vol. 4, no. 4, pp. 280-294, Oct.-Dec. 2007, doi: 10.1109/TDSC.2007.70208.

[10] Molette, P.; Cougnet, C.; Saint-Aubert, PH.; Young, R.W.; Helas, D. ; "Technical and Economical Comparison Between a Modular Geostationary Space Platform and a Cluster of Satellites". Acta Astronautica (Pergamon Press Ltd.) 12 (11): 771–784.doi:10.1016/0094-5765(84)90097-3; 1984.

[11] Brown, Owen ; Eremenko, Paul; "Fractionated Space Architectures: A Vision for Responsive Space"; 4th Responsive Space Conference. Los Angeles, CA: American Institute of Aeronautics and Astronautics. pp. Paper No. AIAA–RS4–2006–1002; (2006).

[12] Brown, Owen; Eremenko, Paul; "The Value Proposition for Fractionated Space Architectures"; AIAA Space 2006. San Jose, CA: American Institute of Aeronautics and Astronautics. pp. Paper No. AIAA–2006–7506; (2006).

[13] K. Fall, "A delay-tolerant network architecture for challenged internets," in Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ser. SIGCOMM '03. New York, NY, USA: ACM, 2003, pp. 27–34.

[14] J. A. Fraire and J. M. Finochietto, "Design challenges in contact plans for disruption-tolerant satellite networks," IEEE Communications Magazine, vol. 53, no. 5, pp. 163–169, May 2015.

[15] J. Fraire and J. Finochietto, "Routing-aware fair contact plan design for predictable delay tolerant networks," Ad Hoc Networks, vol. 25, pp. 303 – 313, 2015, new Research Challenges in Mobile, Opportunistic and Delay-Tolerant Networks Energy-Aware Data Centers: Architecture, Infrastructure, and Communication.

[16] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss, "Delay-tolerant networking architecture," In-

ternet Requests for Comments, RFC Editor, RFC4838, April 2007, http://www.rfc-editor.org/rfc/rfc4838.txt.

[17] C. Caini, H. Cruickshank, S. Farrell, and M. Marchese, "Delay- and disruption-tolerant networking (dtn): An alternative solution for future satellite networking applications," Proceedings of the IEEE, vol. 99, no. 11, pp. 1980–1997, Nov 2011.

[18] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss, "Delay-tolerant networking: An approach to interplanetary internet," Comm. Mag., vol. 41, no. 6, pp. 128–136, Jun. 2003. [Online]. Available: http://dx.doi.org/10.1109/MCOM.2003. 1204759

[19] C. Caini and R. Firrincieli, DTN for LEO Satellite Communications. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 186–198.

[20] J. A. Fraire and P. A. Ferreyra, "Assessing dtn architecture reliability for distributed satellite constellations: Preliminary results from a case study," in Biennial Congress of Argentina (ARGENCON), 2014 IEEE, June 2014, pp. 564–569.

[21] J. A. Fraire, P. Madoery, S. Burleigh, M. Feldmann, J. Finochietto, A. Charif, N. Zergainoh, and R. Velazco, "Assessing contact graph routing performance and reliability in distributed satellite constellations," Journal of Computer Networks and Communications, vol. 2017, p. 18, 2017, article ID 2830542.

[22] C. Barrientos, A. Ferral, L. Cara, J. A. Fraire, R. Velazco, P. Madoery, and P. Ferreyra, "A segmented architecture approach to provide a continuous, long-term, adaptive and cost-effective glaciers monitoring system based on dtn communications and cubesat platforms," in 1st IAA Latin American Symposium on Small Satellites, Buenos Aires, Argentina, Feb. 2017.

[23] Pablo A. Ferreyra, Carlos A. Marqués, Ricardo T. Ferreyra, and Javier P. Gaspar; "Failure Map Functions and Accelerated Mean Time to Failure Tests: New Approaches for Improving the Reliability Estimation in Systems Exposed to Single Event Upsets"; IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 52, NO. 1, FEBRUARY 2005.

[24] P. A. Ferreyra, G. Viganotti, C. A. Marques, R. Velazco, and R. T. Ferreyra, "Failure and coverage factors based markoff models: A new approach for improving the dependability estimation in complex fault

tolerant systems exposed to seus," IEEE Transactions on Nuclear Science, vol. 54, no. 4, pp. 912–919, Aug 2007.

[25] Randal E. Bryant and David R. O'Hallaron, "Computer system a programmer's perspective", THIRD EDITION, ISBN 978-0-13-409266-9-ISBN 0-13-409266-X, Pearson.

[26] James F. Kurose, Keith W. Ross; "COMPUTER NETWORKING A Top-Down Approach"; SIXTH EDITION, ISBN-13: 978-0-13-285620-1, Pearson Education, Inc., publishing as Addison-Wesley.

[27] W. J. Dally and B. Towles, "Principles and Practices of Interconnection NetWorks", Morgan Kaufmann, San Francisco, Calif, USA, 2003.

[28] Jhon Eduardo Bedoya Camacho, Tesis de Grado de Master, "PROPUESTA Y SIMULACIÓN DE UNA SOLUCIÓN BASADA EN REDES TOLERANTES AL RETARDO PARA PROPORCIONAR COMUNICACIONES EN ENTORNOS REMOTOS AISLADOS", (2013), Universidad Politécnica de Madrid, Escuela Técnica Superior de Ingenieros de Telecomunicación.

[29] K. Scott and S. Burleigh, "Bundle Protocol Specification," Internet Requests for Comments,RFC 5050, 2007, issn 2070-1721, http://www.rfc-editor.org/rfc/rfc5050.txt

[30] CCSDS, "Schedule-Aware Bundle Routing (SABR), White Book", CCSDS Secretariat, Consultative Committee for Space Data Systems (CCSDS), 2016, CCSDS 232.0-B-2, Feb, Proposed Recommendation for Space Data System Standards.,

[31] P. Molette and C. Cougnet and Ph. Saint-Aubert and R.W. Young and D. Helas, "Technical and economical comparison between a modular geostationary space platform and a cluster of satellites", Acta Astronautica, vol11, num12, p771 - 784, 1984, issn 0094-5765, http://www.sciencedirect.com/science/article/pii/0094576584900973.

[32] Brown, O. and Eremenko, P., "The Value Proposition for Fractionated Space Architectures", AIAA-2006-7506, AIAA Space 2006, 2006, San Jose, CA,

[33] Mohsen Mosleh and Kia Dalili and Babak Heydari, "Optimal Modularity for Fractionated Spacecraft: The Case of System F6", Procedia Computer Science, vol28, p164 - 170, 2014, 2014 CSER Conference, issn 1877-0509, http://dx.doi.org/10.1016/j.procs.2014.03.021.

[34] Jain, S., Fall, K., and Patra, R., "Routing in a Delay Tolerant Network", SIGCOMM Comput. Commun. Rev. Vol.34, Iss.4, p145-158, August 2004.

[35] András Varga, OMNeT++ - Simulation Manual, 2016.

[36] S. Wang and T. Jin, " Wireless network-on-chip: a survey", The Journal of Engineering, 2014, vol2014, num3, p98-104, ISSN 2051-3305.

[37] Pablo Ferreyra, Juan A. Fraire, Fabián Gomez, Raoul Velazco, Daniel Sánchez, Dardo Viñas Viscardi; "Delay-Tolerant Wireless Networks on Chip: Preliminary Analysis and Results"; March 2019, DOI: 10.1109/LATW.2019.8704623, Conference: 2019 IEEE Latin American Test Symposium (LATS).

[38] L. Dailey Paulson, Computer, "Computer system, heal thyself", 2002, vol35, num8,p20-22, ISSN 0018-9162.

[39] C. Pahl, "Containerization and the PaaS Cloud", IEEE Cloud Computing, 2015, vol2, num3, p24-31,ISSN 2325-6095, May.

[40] R. Dua and A. R. Raja and D. Kakadia, "Virtualization vs Containerization to Support PaaS", 2014 IEEE International Conference on Cloud Engineering, 2014, p610-614, March.

[41] Petersen, E., "Single Event Effects in Aerospace", isbn 9780-4707-6749-8, lccn 2011-0021-91, https://books.google.com.ar/books?id=Er5_rzW0q3MC, 2011, John Wiley & Sons.