



***Facultad
de
Ciencias***

**Diseño y despliegue de un framework de
verificación aplicado a una herramienta de
simulación de redes de interconexión
(Design and deployment of a verification
framework applied to an interconnection
networks simulation tool)**

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Álvaro Sainz Bárcena

Director: Enrique Vallejo Gutiérrez

Co-Director: Patricia López Martínez

Julio - 2017

Índice general

Índice de figuras	3
Índice de tablas	4
Agradecimientos	5
Resumen	6
Abstract	7
1. Introducción	8
1.1. Motivación del proyecto	8
1.2. Objetivos del proyecto	9
1.3. Estructura del documento	9
2. Trasfondo y herramientas empleadas	11
2.1. Red simulada	11
2.1.1. Topología	12
2.1.2. Router	13
2.1.3. Tipos de tráfico	13
2.1.4. Mecanismos de encaminamiento	14
2.2. Simulador FOGSim	16
2.3. Verificación y Validación	17
2.4. Herramientas empleadas	17
2.4.1. Eclipse	17
2.4.2. Google Test/Mock	18
2.4.3. Gnuplot	18
2.4.4. Git	18
2.4.5. GitLab	18
2.4.6. Jenkins	18
2.4.7. LaTeX	19
3. Desarrollo de pruebas y resultados	20
3.1. Pruebas unitarias	20
3.2. Pruebas de integración	23
3.2.1. Test tráfico uniforme	23
3.2.2. Test tráfico adverso	24
3.3. Prueba de sistema	27
4. Automatización con Jenkins	32
4.1. Jenkins y programación de tareas	32
4.2. Resultados	34
5. Conclusiones y trabajos futuros	37
5.1. Conclusiones	37

5.2. Trabajos futuros	38
Bibliografía	39
Anexos	40
Anexo A. Fichero de entrada tipo para el test de sistema	41
Anexo B. Fichero de salida ejemplo	42
Anexo C. Script test de sistema: Throughput y Latencia	46

Índice de figuras

2.1. Red <i>Dragonfly</i> con $h = p = 2$ y $a = 4$. Imagen tomada de [1].	12
2.2. Arquitectura general de un router con canales virtuales. Tomada de [2].	13
2.3. Ejemplos de encaminamiento Mínimo, Valiant, y PAR en una red Dragonfly. Figura cogida de [3].	14
2.4. Mecanismo UGAL. La congestión de los buffers globales (q0, q1) no está conectada directamente al router origen. Imagen tomada de [4].	15
2.5. Encaminamiento adaptativo progresivo. Tomada de [4].	16
3.1. Diagrama de clases simplificado de la dependencia existente entre el generador y el módulo del switch.	21
3.2. Test superado con tráfico uniforme en la red. Ciclos ejecutados=10.000.000, seed=1, probabilidad de inyección=90 %.	25
3.3. Test fallido con tráfico uniforme en la red. Ciclos ejecutados=1.000.000, seed=23, probabilidad de inyección=90 %.	26
3.4. Test superado con tráfico adverso no aleatorio en la red. Ciclos ejecutados=1.000.000, seed=57, probabilidad de inyección=90 %.	26
3.5. Test superado con tráfico adverso aleatorio en la red. Ciclos ejecutados=1.000.000, seed=18, probabilidad de inyección=90 %.	27
3.6. Valores del multiplicador de la desviación típica que hay que aplicar para que el test no detecte fallos en las diferentes pruebas concretas con las configuraciones indicadas. . .	31
4.1. Captura de pantalla de la configuración de la tarea encargada de ejecutar la prueba de sistema. Concretamente, la frecuencia con la que se ejecuta.	33
4.2. Captura de pantalla de la configuración de la tarea encargada de ejecutar la prueba de sistema. Concretamente, la ejecución de comandos mediante consola.	34
4.3. Captura de pantalla del <i>webhook</i> creado que avisa a la tarea de Jenkins después de aplicar un evento <i>push</i> (cambios en los ficheros del repositorio).	34
4.4. Captura de pantalla del panel de control de Jenkins una vez definidas la tarea de la prueba de sistema y la de las pruebas unitarias e integración.	35
4.5. Captura de pantalla del panel de control de Jenkins tras una ejecución fallida de la tarea.	35
4.6. Captura de pantalla de la salida por consola como resultado de una ejecución fallida de la tarea debido a la inexistencia del fichero del script de la prueba de sistema.	35
4.7. Captura de pantalla de la salida por consola como resultado de una ejecución correcta de la tarea de la prueba de sistema, el resultado exitoso del test de sistema.	36

Índice de tablas

3.1. Resultados de varias ejecuciones de latencia y throughput con tráfico uniforme y routing mínimo.	28
3.2. Resultados de varias ejecuciones de latencia y throughput con tráfico uniforme y routing piggybacking.	29
3.3. Resultados de varias ejecuciones de latencia y throughput con tráfico adverso aleatorio y routing piggybacking.	30
3.4. Resultados de varias ejecuciones de latencia y throughput con tráfico adverso aleatorio y routing valiant.	30

Agradecimientos

Quiero agradecer la realización de este proyecto a todos los buenos profesores que me han enseñado y formado hasta ahora, y que han sabido motivarme para llegar hasta donde estoy.

A Enrique y Patricia, directores de este trabajo por guiarme, aconsejarme y ayudarme, y que han conseguido que su realización haya sido mucho más amena y motivadora.

A mis padres por darme la oportunidad de poder estudiar el grado y depositar su confianza en mí. También a mis amigos con quienes me escapo de esta atmósfera informática y consiguen sacar siempre unas buenas risas.

Y, por último y no por ello menos importante, a Tam por aguantarme tantas horas tanto en la biblio como fuera de ella, por saber animarme y ser sin duda un apoyo fundamental.

Resumen

Debido a las tareas de investigación del grupo de Arquitectura y Tecnología de Computadores de la UC, el simulador de redes de interconexión sobre el que trabajan sufre modificaciones continuas de código, lo que en ocasiones ha provocado tener que invertir un tiempo excesivo en localizar y solventar ciertos errores generados a consecuencia de dichos cambios.

Con el objetivo de solucionar, o al menos minimizar el proceso de detección y localización de estos errores, se ha elaborado un framework de verificación compuesto por un conjunto de pruebas software para comprobar el correcto funcionamiento del simulador después de haber sido modificado. En concreto, utilizando la librería Google Test, se han realizado pruebas unitarias que verifican módulos aislados del simulador y tests de integración para comprobar el comportamiento en conjunto de un grupo de componentes interconectados. También se ha elaborado una prueba de sistema mediante un script en Python que verifica el software completo.

Además, se ha englobado el framework en una estrategia de integración continua con la herramienta Jenkins de modo que las pruebas se ejecuten tras aplicar cambios en el simulador o periódicamente, de manera automática y transparente al programador.

Esta buena práctica supone ventajas al proceso de desarrollo del software, ya que ayudará a los programadores a detectar la localización de los errores tras haber aplicado cambios en el código del simulador.

Palabras clave: framework de pruebas, verificación y validación, red de interconexión, Dragonfly, simulador.

Abstract

Due to the research tasks of the *Arquitectura y Tecnología de Computadores* group of the UC, the simulator of interconnection networks on which they work undergoes continuous code modifications, which has sometimes caused them to spend an excessive time in locating and solving certain errors generated by the changes.

In order to solve, or at least minimize this process, a verification framework has been developed, consisting of a set of software tests to verify the correct operation of the simulator after having been modified. In particular, using the Google Test library, unit tests have been performed to verify isolated modules of the simulator, integration tests to check the joint behavior of a group of interconnected components. A system test has also been developed using a Python script that verifies the complete software behaviour.

In addition, the framework has been embedded in a continuous integration strategy with the Jenkins tool so that the tests are executed after applying changes in the simulator or periodically, in an automatic and transparent manner for the programmer.

This good practice gives advantages to the software development process, as it will help programmers detect the location of errors after applying changes to the simulator code.

Keywords: test framework, verification and validation, interconnection network, Dragonfly, simulator.

Capítulo 1

Introducción

Todo buen desarrollo de proyectos software ha de seguir ciertas metodologías para garantizar que el producto final sea el correcto, esté libre de fallos y posea todas las funcionalidades requeridas de un modo eficiente. Durante el proceso de desarrollo, hay cambios continuos en el proyecto, realizados usualmente por más de un programador. Como consecuencia, es altamente probable que a medida que se van aplicando diferentes cambios, se introduzcan defectos que pasen inadvertidos y transparentes al programador, y afecten de un modo más grave y costoso al proyecto en un futuro. Además, debido a la participación de numerosas personas sobre un único proyecto, se pueden generar diversos conflictos entre diferentes versiones del código a pesar de trabajar en distintas partes del mismo. Para solventar o minimizar estos problemas, se han de seguir una serie de técnicas durante el desarrollo del software. Los miembros del equipo han de integrar periódicamente su trabajo a un repositorio compartido con un sistema de control de versiones [5] con el objetivo de informar de sus avances y cambios al resto del equipo. A su vez, esto permite ejecutar pruebas para detectar los posibles errores cada vez que se han integrado cambios en el proyecto de manera automática. Esta buena práctica de desarrollo software se denomina integración continua [6].

Los principales objetivos de la integración continua consisten en encontrar y arreglar defectos con mayor rapidez, mejorar la calidad del software y reducir el tiempo en que se tarda en validar y verificar nuevas actualizaciones del proyecto. Gracias a esta técnica se consigue mejorar la productividad del equipo, puesto que los programadores evitan realizar tareas manualmente como la ejecución de pruebas. Consiguientemente, se localizan y arreglan errores con mayor rapidez visualizando el resultado de las pruebas en cada integración realizada, evitando que el error se convierta en un problema mayor posteriormente.

En el presente trabajo se pretende diseñar y desplegar un framework de pruebas, de modo que se consigan las ventajas ya mencionadas a través de la elaboración de diversas pruebas y ejecución automática de éstas sobre un proyecto software, en concreto, sobre un simulador que emula el comportamiento de una red de interconexión.

1.1. Motivación del proyecto

Actualmente el grupo de investigación de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria utiliza un simulador de una red de interconexión de tipo *Dragonfly* [7], con el objetivo de investigar cómo mejorar el rendimiento de la red; disminuyendo la *latencia* (retardos temporales) y aumentando el *throughput* (datos transmitidos por unidad de tiempo).

Este simulador fue elaborado por ellos y con el único fin de investigación. Por lo tanto y a diferencia de un software comercial y dedicado a uso empresarial, lo utiliza un grupo muy reducido de personas y está destinado a tareas de investigación, en lugar de programarse una sola vez y que lo disfruten muchos clientes. Debido a la naturaleza de estas tareas, el código del simulador sufre cambios constantes para probar diferentes modificaciones en la red y determinar cómo afectan al rendimiento.

En ciertas ocasiones alguno de estos cambios ha dado lugar a resultados inesperados o extraños cuya causa se ha tardado en averiguar más tiempo del deseado. Debido a esto y para evitar tener que invertir tiempo en encontrar y solucionar los problemas, es necesaria una herramienta que permita detectarlos comprobando el correcto comportamiento de los diferentes componentes que forman el simulador. De esta manera surge el presente trabajo de fin de grado, el desarrollo e implantación de un framework que verifique el funcionamiento del simulador después de haber aplicado diferentes modificaciones en el código y que detecte si algún módulo se comporta de un modo distinto al esperado.

1.2. Objetivos del proyecto

La finalidad de este proyecto es elaborar un framework que realice las pruebas/tests necesarias que aseguren el correcto funcionamiento del simulador de redes de interconexión utilizado en el grupo de Arquitectura y Tecnología de Computadores de la Facultad de Ciencias. De este modo, se pretende ayudar a los investigadores que emplean el simulador evitándoles tener que invertir tiempo en identificar y localizar errores que causan resultados incoherentes. Para conseguirlo, se han propuesto una serie de objetivos:

- Estudio y comprensión del funcionamiento del simulador, del mismo modo que el entendimiento del funcionamiento de una red de interconexión de tipo *Dragonfly*.
- Elaboración de pruebas unitarias para testear el comportamiento de los módulos de un manera aislada, es decir, con independencia de otros módulos del simulador de los que dependa su comportamiento.
- Elaboración de pruebas de integración que comprueben que un grupo de módulos que hayan superado las pruebas unitarias, funcionen juntos correctamente.
- Elaboración de pruebas de sistema que verifiquen el comportamiento de todo el simulador, comprobando que los resultados más relevantes (*latencia* y *throughput*) son los esperados.
- Automatización del proceso de compilación, ejecución y presentación de los resultados del framework de verificación en el momento que se desee, ya sea cada cierto tiempo, cuando se realice algún *commit* (cambios parciales o finales en el código) o cuando el usuario lo necesite.

De este modo, se pretende abarcar principalmente dos campos. Por un lado y orientado a la mención de ingeniería de computadores de Ingeniería Informática, conocer el funcionamiento de un simulador de redes de interconexión utilizado en el ámbito de la investigación y, por consecuencia, estudiar la topología y los componentes que forman la red *Dragonfly* programada en el simulador. Por otro lado y enfocado a la mención de Ingeniería del Software, la realización de diferentes tipos de pruebas que garanticen el correcto funcionamiento del software, así como el estudio y elección de las herramientas más apropiadas para la elaboración del framework y su integración en un entorno de integración continua.

1.3. Estructura del documento

El presente informe está dividido en cuatro capítulos además del actual. Se estructuran de la siguiente manera:

- Capítulo 2: Trasfondo y herramientas empleadas. Se explica el funcionamiento de todos los componentes y configuraciones de la red que han sido probados en el framework, el simulador FOGSim y se introduce el concepto de verificación y validación. Además, se detallan las herramientas utilizadas para la elaboración del trabajo.

-
- Capítulo 3: Desarrollo de pruebas y resultados. En este capítulo se describe en detalle cómo se han elaborado las pruebas que forman el framework de verificación y los resultados obtenidos tras ejecutarlas sobre el simulador.
 - Capítulo 4: Automatización con Jenkins. Se indica el proceso seguido para conseguir que las pruebas se ejecuten de manera automática, ya sea periódicamente o cuando el programador aplique cambios en el simulador, a través del servidor de integración continua Jenkins.
 - Capítulo 5: Conclusiones y trabajos futuros. En este capítulo final se exponen las conclusiones obtenidas tras la realización del trabajo y futuras mejoras que pueden aplicarse sobre el framework de verificación desarrollado.

Capítulo 2

Trasfondo y herramientas empleadas

Cada vez es más frecuente escuchar el término *High Performance Computing*, refiriéndose a sistemas con mucha capacidad de cómputo destinados a resolver problemas complejos que son utilizados mayoritariamente en centros de investigación. La característica fundamental de estos computadores es su alta velocidad de procesamiento, lo que ha permitido un gran avance en investigaciones de diferentes ámbitos, logrando increíbles resultados operando con conjuntos de datos extremadamente grandes. Alguno de los ejemplos son el estudio del clima y la predicción de cambios climáticos para evitar tragedias, el estudio del universo mediante simulaciones como la evolución estelar, o incluso en el ámbito militar para simular explosiones nucleares y evitar realizar pruebas verdaderas. También se utilizan para el estudio del plegamiento de las proteínas y cómo afecta a las personas que sufren enfermedades como cáncer o Alzheimer.

Para conseguir este alto rendimiento, un supercomputador está compuesto por muchos nodos de cómputo (servidores) que operan sobre el mismo problema. Hoy en día, existen supercomputadores que tienen hasta decenas de miles de servidores [8]. El objetivo es que todos estos nodos de cómputo de la máquina trabajen de un modo paralelo, es decir, que realicen sus operaciones *simultáneamente* con respecto al resto de nodos y conseguir dividir el problema entre el número de servidores que posea el supercomputador.

Sin embargo, los nodos tienen que compartir datos de la aplicación, por lo que el throughput y la latencia de las comunicaciones entre ellos son críticos para el rendimiento. Este inconveniente se genera principalmente porque un nodo puede necesitar esperar a que termine de operar otro nodo para poder ejecutar su parte ya que comparten recursos. Además, es necesario mantener la coherencia en el problema evitando que diferentes nodos accedan y modifiquen a la vez a los mismos datos.

Por lo tanto, que haya paralelismo implica que los distintos servidores del supercomputador tengan que intercambiar y compartir información, es decir, se necesitan comunicar. Esta comunicación se realiza mediante un medio, denominado *red de interconexión*, y puede ser una de las mayores limitaciones de rendimiento de los supercomputadores en determinadas aplicaciones debido a la frecuencia con la que se deben comunicar los nodos y al gran número de los mismos que poseen hoy en día estas máquinas. Debido a esto, las redes de interconexión han adquirido una gran importancia en el mundo de los supercomputadores.

2.1. Red simulada

La red implementada en el simulador es del tipo *Dragonfly* [7], una red de altas prestaciones destinada a alcanzar un alto rendimiento, elevada escalabilidad y bajo coste. Al igual que toda red de interconexión, está compuesta por los nodos de cómputo que envían y reciben los mensajes, routers que determinan el camino que deben seguir los mensajes para llegar a su destino, y los enlaces por los cuales circula el mensaje. Es importante indicar que las métricas más importantes para el rendimiento de la red son la *latencia*, que hace referencia al intervalo de tiempo desde el envío hasta la recepción

del mensaje, y el throughput, indicando la cantidad de paquetes que se transfieren por unidad de tiempo. Para conseguir un mayor rendimiento general de la red, el objetivo es disminuir la latencia y aumentar el throughput.

2.1.1. Topología

El modo en que están organizados los componentes mencionados anteriormente describen la topología de la red. Principalmente puede ser de dos tipos: directa e indirecta. La *Dragonfly* es una red directa porque todos los routers o encaminadores están conectados a algún nodo de cómputo. Por contraposición, una indirecta se caracteriza por la existencia de routers de tránsito que no están unidos a ningún nodo y únicamente se encargan de reenviar información.

La red *Dragonfly* está dividida en diferentes grupos compuestos por varios routers cada uno. De este modo, podemos diferenciar dos niveles en la red. Por un lado, los routers que componen un grupo y cuya comunicación se realiza mediante enlaces locales, y por otro lado los diferentes grupos que forman la red y que se comunican mediante enlaces globales. De esta manera, los enlaces locales (enlaces cortos) pueden ser cables eléctricos mientras que los globales (enlaces largos) son ópticos, lo que permite reducir el coste de la red.

Todos los routers de un mismo grupo están conectados directamente entre sí. Además, cualquier grupo se conecta con el resto de los grupos mediante los enlaces globales permitiendo que todos los routers de la red se puedan comunicar. Tanto el número de nodos de cómputo asociados a cada router, el número de routers que componen un grupo y el número de enlaces globales que posee cada router (que conectarán con routers de otros grupos) son parámetros de la red. Comúnmente están definidos como:

- p : El número de nodos asociados a cada router $\implies p = h = a/2$
- a : El número de routers por grupo $\implies a = 2 \cdot p = 2 \cdot h$
- h : El número de enlaces globales por router $\implies h = p = a/2$

La figura 2.1 nos muestra la topología una red compuesta por cuatros routers por grupo, dos enlaces globales que posee cada router que conectan con otros grupos, y dos nodos de cómputo asociados a cada router.

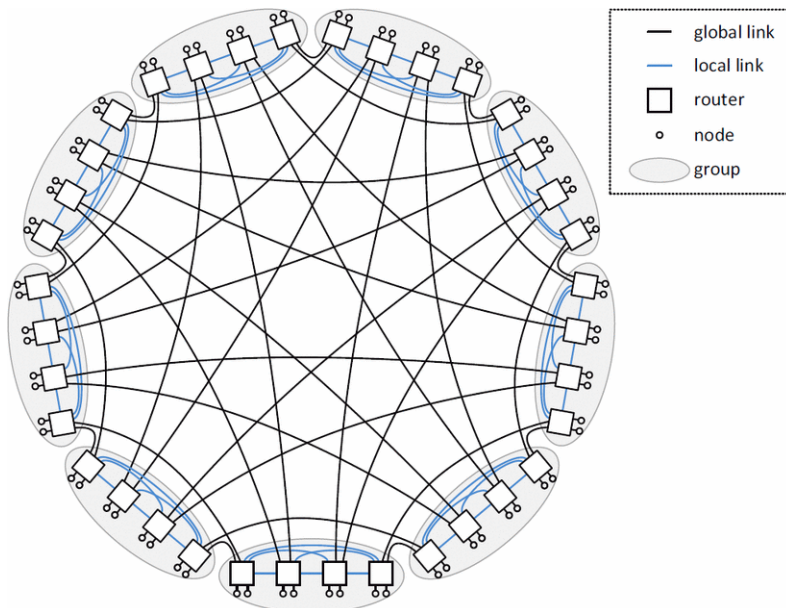


Figura 2.1: Red *Dragonfly* con $h = p = 2$ y $a = 4$. Imagen tomada de [1].

2.1.2. Router

Como ya mencionamos en el apartado anterior, los dispositivos encargados de decidir el camino que han de seguir los mensajes para llegar al destino correcto son llamados routers o encaminadores. Dependiendo del *mecanismo de routing* empleado, que se explicarán en la sección 2.1.4, el camino que sigue un mensaje por la red puede variar, pese a que el nodo destino sea el mismo. Al no pertenecer a un entorno IP, donde un switch actúa en Ethernet y un router a nivel IP, se puede referir a estos dispositivos como routers o switches.

El router/switch está formado por puertos de entrada (*input*) y de salida (*output*), los cuales se encargan de recibir y de enviar, respectivamente, los datos. Además, cada puerto puede tener buffers (pequeñas memorias) que almacenan el paquete temporalmente y pueden estar conectados a los de entrada, los de salida o ambos. Toda la memoria de un puerto se organiza en diferentes canales virtuales (cada uno con uno o varios buffers) con almacenamiento y control de flujo (técnica para sincronizar el envío de paquetes entre el emisor y el receptor) independientes, a pesar de emplear el mismo enlace físico (puerto). Esto ofrece varias ventajas como evitar el *deadlock* (bloqueo permanente de la red) o priorizar un tráfico frente a otro. Asimismo, los puertos de entrada y salida se interconectan mediante un *crossbar*, denominado en algunos entornos como un switch interno que conmuta entre ambos puertos permitiendo la unión entre las distintas entradas y salidas. La planificación de los paquetes que se inyectan en los buffers de entrada para atravesar el *crossbar* la realiza el *allocator*, permitiendo o denegando el acceso a cada buffer de entrada, de acuerdo a las necesidades y contención puntual. El componente dedicado a determinar el puerto de salida de un mensaje recibido a través de un puerto de entrada se denomina unidad de routing.

En la siguiente imagen (figura 2.2) se puede observar la estructura básica de un router con canales virtuales en cada buffer.

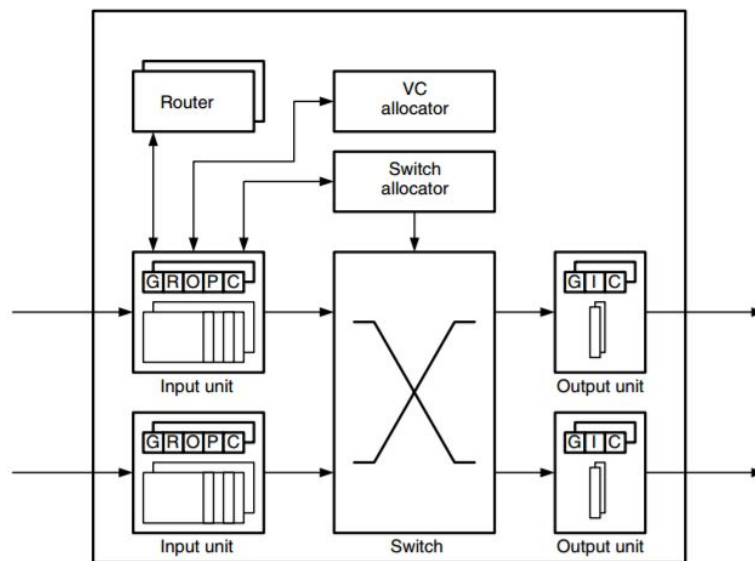


Figura 2.2: Arquitectura general de un router con canales virtuales. Tomada de [2].

2.1.3. Tipos de tráfico

Los mensajes o paquetes que circulan en la red se denomina tráfico. Este tráfico es generado por los nodos de cómputo que necesitan comunicarse con otros nodos, inyectando los paquetes en su router asociado. El tipo de tráfico generado afecta notablemente al rendimiento de la red ya que puede ocasionar congestión y saturación de los enlaces. A continuación, se explicarán los principales patrones de tráfico existentes que han sido probados en el *framework* y la manera en que se implementan en el simulador. Se utilizan patrones de tráfico sintético que aproximan el comportamiento de aplicaciones reales en algunos casos.

2.1.3.1. Tráfico uniforme

El tráfico uniforme consiste en elegir un nodo destino de manera aleatoria de entre todos los nodos que forman la red, a excepción del nodo fuente. De este modo se consigue dividir el tráfico entre todos los enlaces globales y locales de la red de una manera equitativa, por lo que no hay congestión ni saturación en los enlaces.

2.1.3.2. Tráfico adverso

En este apartado se analiza el patrón de tráfico que resulta más adverso para esta topología y que puede ocurrir en ciertas condiciones, por ejemplo, cuando se ejecuta una aplicación en sólo dos grupos de la red. El tráfico adverso se caracteriza por transmitir datos únicamente entre un grupo reducido de nodos, mientras que el resto no recibe ningún paquete. Este patrón puede configurarse de diferentes modos en cuanto a la elección de los nodos destinos. A continuación se definen dos de las más comunes.

Por un lado, el tráfico adverso no aleatorio consiste en enviar todos los mensajes generados en un grupo a un mismo router destino y repartirlos de manera aleatoria entre todos los nodos del router.

Por otro lado, también es frecuente el tráfico adverso aleatorio, en donde en lugar de elegir un router destino, se escoge un mismo grupo destino para todos los paquetes generados y se dividen de manera similar al tráfico uniforme entre los routers del grupo.

Como consecuencia de este patrón, el enlace global que conecta el grupo origen y el grupo destino se satura, ya que existe un único cable global que une cada par de grupos. De igual modo, los enlaces locales del grupo destino sufren las mismas consecuencias.

2.1.4. Mecanismos de encaminamiento

El mecanismo de routing o encaminamiento define el camino que ha de seguir un mensaje generado en un nodo origen para llegar al nodo destino en la red. Principalmente, se dividen en mecanismos *Oblivious* y adaptativos. Por un lado, los *Oblivious* fijan el trayecto que debe seguir el mensaje una vez es inyectado en la red. Por otro lado, el enrutamiento adaptativo es capaz de modificar el camino a seguir del paquete en función del estado de la red. Esto permite aumentar el rendimiento de la misma cuando el enlace que debería seguir el paquete está saturado, ya que posibilita la elección de otro enlace. Sin embargo, el encaminamiento adaptativo puede generar una situación en la que un paquete permanezca circulando por la red y nunca llegue a su destino, lo que se conoce como *livelock*.

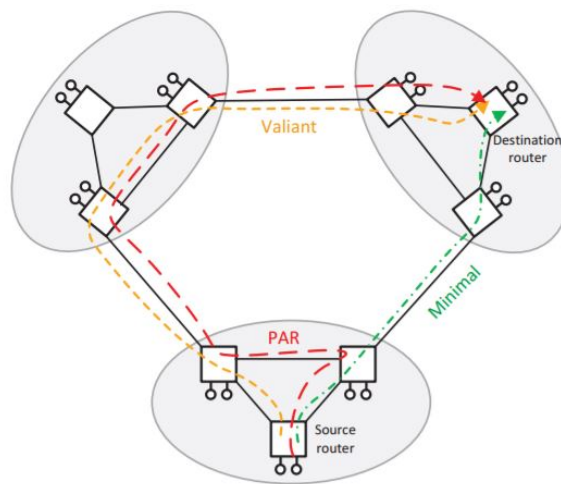


Figura 2.3: Ejemplos de encaminamiento Mínimo, Valiant, y PAR en una red Dragonfly. Figura cogida de [3].

2.1.4.1. Routing Mínimo

Uno de los mecanismos *Oblivious* más básicos es el routing mínimo. Consiste en comunicar el router asociado al nodo fuente con el router del nodo destino a través del camino con el menor número de saltos posible. Como se puede apreciar en la figura 2.3, requiere como máximo un total de tres saltos en la red: $l - g - l$, donde l indica el enlace local y g el enlace global. Este mecanismo consigue un alto throughput con tráfico uniforme porque todos los enlaces son usados equitativamente y ninguno está saturado.

2.1.4.2. Routing Valiant

Como se explicó en la sección 2.1.3.2, cuando hay tráfico adverso en la red, se saturan determinados enlaces globales y locales que implican una gran pérdida de rendimiento. Para solventar este problema, el encaminamiento Valiant [1] (no mínimo) tiene como objetivo repartir el tráfico entre todos los enlaces globales de la red. Para conseguirlo, se elige un nodo intermedio aleatoriamente de entre todos los routers de la red (a excepción del origen y destino), y seguidamente, se envía el mensaje al router destino mediante routing mínimo. Pueden ocurrir hasta 6 saltos en la red: $l - g - l - l - g - l$.

A pesar de que este mecanismo implique un mayor número de saltos para llegar al destino, se consigue evitar la congestión en la red y es muy eficiente en patrones adversos, obteniendo una latencia mucho menor en estos casos.

2.1.4.3. Routing Adaptativo

Como su propio nombre indica, este mecanismo se adapta al estado de la red para escoger el camino que debe seguir el paquete una vez es inyectado en el router. En concreto, el mecanismo Universal Globally-Adaptative Load-balanced (UGAL [9]), elige entre utilizar el encaminamiento mínimo o el Valiant en función de la ocupación de los buffers de los routers de manera que se obtenga la menor latencia posible. Esta latencia se calcula como el producto entre el número de saltos y la ocupación del buffer. Como se muestra en la figura 2.4, la información del estado de los buffers de los enlaces globales no está disponible para el router origen al no estar conectado directamente. En este caso, se utilizan los buffers locales para estimar la ocupación de los globales.

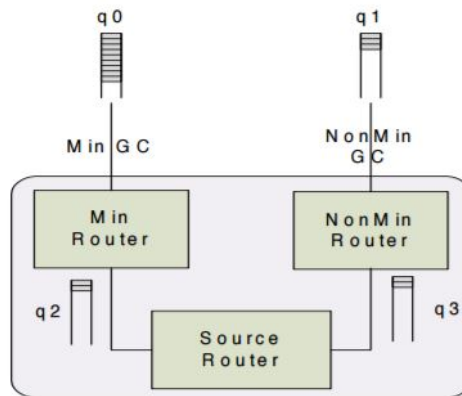


Figura 2.4: Mecanismo UGAL. La congestión de los buffers globales (q_0 , q_1) no está conectada directamente al router origen. Imagen tomada de [4].

Otro encaminamiento adaptativo es el Progressive Adaptive Routing (PAR), similar al UGAL, pero con la peculiaridad de que la decisión entre utilizar routing mínimo o Valiant (no mínimo), la evalúa en cada salto del grupo origen. Si en un momento se opta por un camino no mínimo, es definitivo y no se vuelve a evaluar. Por lo tanto, en el peor de los casos, si se determina un enrutamiento mínimo en el primer salto, y para el siguiente un Valiant, el camino más largo que calcula PAR estará compuesto por seis saltos: $l - l - g - l - g - l$, como se aprecia en la figura 2.3 pintado de rojo. En la figura 2.5 se refleja el trayecto que sigue el paquete.

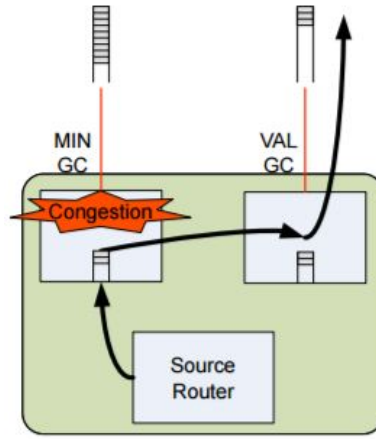


Figura 2.5: Encaminamiento adaptativo progresivo. Tomada de [4].

2.1.4.4. Routing Piggybacking

El encaminamiento Piggybacking se basa en el routing adaptativo para conocer el estado de los enlaces locales, pero a diferencia de éste, utiliza un mecanismo para averiguar la saturación de los globales. Para conocerlo, cada router determina si su enlace global está o no saturado, y envía un *broadcast* a todos los routers de su grupo. De este modo, conseguimos evitar realizar más saltos cuando el enlace global que queremos atravesar está saturado. Hay que tener en cuenta que este encaminamiento consume un ancho de banda extra para enviar esta información, y que puede llegar con cierto retraso.

2.2. Simulador FOGSim

FOGSim es el nombre que recibe el simulador elaborado y utilizado por el grupo de investigación ATC (Arquitectura y Tecnología de computadores) de la Universidad de Cantabria que emula el comportamiento de la red explicada con anterioridad. Está escrito en el lenguaje de programación C++ y cuenta con más de 10.000 líneas de código y más de 50 clases con fuertes dependencias entre ellas. Este simulador es capaz de emular el comportamiento de una red de interconexión Dragonfly con diferentes configuraciones y tráfico, indicando considerables estadísticas y resultados que reflejan el comportamiento y rendimiento de la red ante diferentes situaciones. Entre otros, los principales parámetros que FOGSim admite para configurar la red son la duración de la simulación medida en ciclos, la topología (parámetros h , p y a), el tipo de tráfico (único para toda la red), la cantidad de carga generada, el mecanismo para evitar *deadlock*, el tamaño del mensaje y el mecanismo de routing. Los resultados más importantes para medir el rendimiento de la red devueltos tras una ejecución son el throughput que indica la cantidad de mensajes transmitidos por unidad de tiempo y la latencia media reflejando el tiempo de propagación y transmisión del mensaje.

El simulador recibe la configuración de la red mediante un fichero de entrada (ver Anexo A) donde se especifican los valores de todos los parámetros necesarios. Además, se ha de indicar por línea de comandos el nombre del fichero de salida, la carga ofrecida (definida con la probabilidad con la que se decide si un paquete es generado o no), y la semilla con la que los números aleatorios son creados para no obtener los mismos resultados para una misma probabilidad en diferentes ejecuciones, puesto que algunos aspectos dependen de esta secuencia pseudoaleatoria como la generación del tráfico o la elección de destinos. En el Anexo B se muestra un ejemplo de fichero de salida, con las estadísticas y resultados generados tras una simulación.

En cuanto a la compilación del código, se utiliza el comando `make` que lee el archivo *Makefile* donde se especifican todos los flags, opciones y librerías necesarias para compilar, así como las clases que componen el simulador. Además, genera el archivo ejecutable para el lanzamiento de FOGSim.

El código del simulador en su versión operativa reside en una rama pública del controlador de versiones Git, concretamente en Bitbucket. Asimismo existen varias ramas privadas con diferentes versiones de FOGSim pertenecientes a los desarrolladores donde aplican cambios en el simulador con el objetivo de investigar cómo mejorar el rendimiento de la red en diferentes situaciones.

2.3. Verificación y Validación

Al conjunto de procesos de comprobación y análisis que aseguran que el software que se desarrolla está acorde a su especificación y cumple las necesidades requeridas se denomina Verificación y Validación (V&V). Según definió Bohem [10], la verificación responde a la pregunta, ¿estamos construyendo el producto correctamente? Mientras que la validación contesta a ¿estamos construyendo el producto correcto? De este modo, se pueden diferenciar ambos conceptos, siendo la verificación la comprobación de que el software cumple con su especificación, y la validación un proceso general probando que el software hace lo que el usuario espera.

Las técnicas de V&V se dividen en técnicas estáticas, las revisiones, y técnicas dinámicas, las pruebas. Ambas permiten detectar y corregir los defectos que hacen desviar al proyecto del resultado esperado. Dentro de las pruebas de software se distinguen diferentes niveles:

- Las pruebas unitarias consisten en verificar que cada módulo que forma el sistema se comporta de la manera adecuada de un modo aislado, es decir, de manera independiente y sin interactuar con ninguna otra parte del producto. Hay que tener en cuenta que para lograr que un módulo opere aisladamente puede ser necesario simular o controlar el comportamiento de otros módulos del sistema en caso de que el módulo testeado tenga dependencias con éstos. Además, los errores están más acotados y son más fáciles de localizar.
- El siguiente nivel de pruebas son los tests de integración. Una vez que todos los módulos del sistema han sido probados independientemente, se agrupan en subsistemas interconectados y se prueba el comportamiento en conjunto a fin de detectar fallos resultantes de la interacción entre ellos. A diferencia de las pruebas unitarias, es más costoso localizar la fuente de los fallos debido a la cantidad de módulos pertenecientes al conjunto probado y su compleja interacción.
- En el próximo nivel se encuentran las de sistema. Estos tests se limitan a verificar el correcto comportamiento del software completo, es decir, con todos sus componentes operativos. Se basan en comprobar si resultado obtenido del software satisface los requisitos establecidos, tanto funcionales como no funcionales.
- Finalmente, el último nivel de pruebas son las de aceptación. En este caso, es el cliente quien realiza las pruebas para decidir si se queda con el producto. Estas pruebas no se van a abordar en el framework puesto que el software no está destinado a un cliente en el sentido estricto de la palabra.

2.4. Herramientas empleadas

En esta sección se explicará el principal software empleado en la realización del trabajo.

2.4.1. Eclipse

Como plataforma para la realización del código de diferentes tests se ha utilizado Eclipse. Es un entorno de desarrollo integrado de código abierto muy usado en programación, con una inmensa cantidad de herramientas, plugins y librerías actualizadas disponibles para la mayoría de tecnologías y lenguajes de programación. Además, cuenta con varios mecanismos (atajos de teclado, búsquedas, compilación en tiempo real...) que facilitan y agilizan tanto la programación como la tarea de depuración.

2.4.2. Google Test/Mock

Para la realización de las pruebas unitarias y de integración, se ha empleado la librería Google Test [11], destinada al testing en C++. Este framework es gratuito, de código abierto, está actualizado y dispone de todas las aserciones necesarias para las pruebas requeridas en este trabajo. Se ha elegido Google Test ya que permite la integración con Jenkins a la hora de automatizar el proceso de ejecución de los tests.

Además dispone de la librería Google Mock, mediante la cual se crean objetos Mock para eliminar las dependencias con otras clases. Esta librería opera en conjunto con Google Test y admite variadas opciones a la hora de controlar el comportamiento del objeto Mock, como definir qué métodos serán llamados, en qué orden, cuántas veces, qué retornarán o indicar un orden para retornar diferentes valores de un mismo método, entre otras.

2.4.3. Gnuplot

Las gráficas mostradas en este trabajo se han realizado mediante Gnuplot, un paquete gratuito que, a través de línea de comandos, crea gráficas de muchos y variados tipos, tanto en 2D como en 3D. Además, permite generar ficheros de diferentes formatos de salida y admite scripts para agilizar el proceso a la hora de crear y modificar las gráficas. En este trabajo, se han empleado diferentes scripts para la creación de las distintas gráficas.

2.4.4. Git

Git es un software de control de versiones para gestionar los cambios que se aplican a un proyecto. Su uso está principalmente orientado a proyectos con una gran cantidad de código fuente y/o que están siendo desarrollados o mantenidos de un modo distribuido entre varias personas. De este modo se consigue evitar conflictos entre diferentes versiones a la vez que informar acerca de los cambios realizados en el código fuente. También cuenta con varias ramas en las que se alojan copias de distintas versiones del proyecto en relación a su estado (funcional, desarrollando, con bugs...) permitiendo una estructura de almacenamiento más cómoda del proyecto.

2.4.5. GitLab

Basado en Git, GitLab es un software de código libre que proporciona un servicio web orientado a la gestión de repositorios y desarrollo de software colaborativo, con la ventaja de ofrecer también un sistema de seguimiento de errores. Además, del mismo modo que Bitbucket donde se aloja actualmente el código del simulador, es un hospedaje web basado en el software previo. Se ha empleado GitLab porque Bitbucket no permite la conexión con Jenkins debido a ciertos recursos requeridos, según se detalla en la sección 4.1. A medida que se aplican cambios en el código de un proyecto alojado en GitLab y se suben al repositorio, se pueden indicar las modificaciones realizadas. También, existe la posibilidad de añadir *Webhooks*, los cuales permiten establecer comunicaciones con otras aplicaciones software externas como Jenkins.

2.4.6. Jenkins

Jenkins es un servidor de integración continua de uso muy extendido, gratuito y de código libre. Es capaz de monitorizar el repositorio de control de versiones y actuar de determinada manera ante cualquier cambio realizado en el repositorio. Este software se basa en definir diferentes tareas para cada *build* (proceso de compilación del código fuente y creación del ejecutable) e indicar qué hacer para cada una de ellas. Por ejemplo, en una tarea se puede programar un *build* para que se realice periódicamente, y en caso de que haya habido algún error, notificar al usuario del mismo, o en caso de éxito ejecutar determinadas pruebas. Además, esta herramienta puede ayudar en el proceso de desarrollo indicando que se lancen métricas de calidad, visualizando el resultado de diferentes tests o generando la documentación del proyecto. Jenkins es capaz de extender su funcionalidad a través de

multitud de *plugins* disponibles, modificando o añadiendo diferentes funciones. Todo esto ayuda en gran medida a los desarrolladores a agilizar el proceso de integración continua de un modo completo y así detectar errores y mejoras en el proyecto.

2.4.7. LaTeX

En lo referente a la redacción del presente informe, se ha empleado el sistema de composición de textos \LaTeX , mayoritariamente utilizado en artículos, libros o trabajos académicos científicos o técnicos. La principal ventaja de este software es que, a diferencia de editores de texto tipo *WYSIWYG* (*what you see is what you get*) como Word, permite al escritor centrarse en el contenido y despreocuparse del formato. Además se basa en instrucciones y macros que agilizan determinadas tareas, como por ejemplo a la hora de citar referencias, enumerar figuras, escribir ecuaciones matemáticas, etc. El editor de texto utilizado ha sido Texmaker, ya que integra muchas herramientas para desarrollar documentos con \LaTeX ayudando en la redacción, como el modo de visualización continuo, auto-completado o corrección ortográfica.

En cuanto a la escritura de la bibliografía, se ha usado la herramienta BibTeX. Admite muchos estilos para la misma y se basa en la lectura de un fichero de información bibliográfica, donde se indica el tipo de documento (libro, artículo, tesis...) e información acerca del mismo para ser representada conforme el estilo escogido.

Capítulo 3

Desarrollo de pruebas y resultados

En este capítulo se explicará detalladamente el desarrollo del *framework*, así como las diferentes pruebas elaboradas que lo componen.

Para la creación de las pruebas se han barajado diferentes posibles frameworks destinados al testing en C++, como UnitTest++, CppUnit o Google Test. Se ha optado por la última opción debido a que se puede integrar fácilmente la librería Google Mock para poder crear objetos Mock (explicados en la sección 3.1) fácilmente y que serán muy necesarios. Tanto UnitTest++ como CppUnit carecen de ésta librería y la tarea de crear los Mocks sería bastante más costosa. Además, Google Test genera un documento XML con los resultados de la ejecución de las pruebas que Jenkins puede representar gráficamente. Por otra parte, Google Test, a diferencia de UnitTest++ y CppUnit, admite la posibilidad de elegir qué tests ejecutar y dispone de aserciones no fatales que admiten seguir ejecutando el test pese a haber resultado fallidas.

Como se ha mencionado anteriormente, el proyecto sobre el que se van a programar las pruebas, tiene una gran cantidad de código y todos los módulos (clases) están muy interrelacionados entre sí. Además, la continua modificación del código fuente, falta de refactorización y escasa documentación dificulta en gran medida tanto la realización de las pruebas como la comprensión del simulador.

Para comenzar a realizar las pruebas, es necesario tener un conocimiento general del funcionamiento de *FOGSim*, así como los módulos que lo componen y las dependencias existentes entre ellos. Con este último objetivo, se ha realizado un diagrama de clases mediante ingeniería inversa con la herramienta *MagicDraw*, gracias al cual se observa la compleja y gran estructura del sistema visualizando las clases (módulos) y las dependencias entre ellas. Este diagrama no se incluye en el documento debido a que por su complejidad no aporta nada a simple vista, sin contar con una herramienta de modelado que permita navegar por él. Dada la complejidad del simulador, se han elegido un conjunto pequeño de módulos para las pruebas unitarias y de integración. Una vez terminado el proyecto, el objetivo sería que cada desarrollador realizase las pruebas unitarias del código nuevo según se fuera desarrollando.

Una vez escogido *Google Test* como framework que se usará para programar las pruebas, se ha creado un proyecto con las librerías necesarias para los tests (*gtest* y *gmock*) y el software del simulador en *Eclipse* para una mayor comodidad en el manejo del código. Con todo esto, ya se dispone de todo lo necesario para comenzar a elaborar las pruebas.

3.1. Pruebas unitarias

En esta prueba se pretende comprobar el correcto funcionamiento de los módulos del simulador por separado sin ninguna dependencia con otras clases. Por ello, la mejor opción es buscar los módulos más aislados posibles (ninguno está aislado totalmente) para facilitar esta tarea. En concreto, una vez examinado el diagrama de clases, se observa que el generador es el módulo que menos dependencias

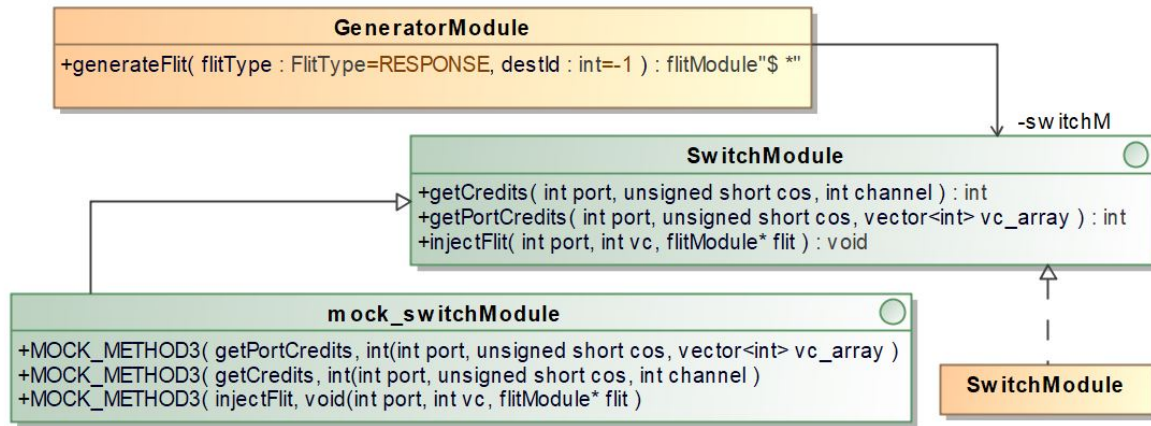


Figura 3.1: Diagrama de clases simplificado de la dependencia existente entre el generador y el módulo del switch.

posee, además de ser de vital importancia para el correcto funcionamiento del simulador. La principal función de este módulo es generar un cierto tráfico en los nodos, determinar el destino de los paquetes e inyectarlos en el switch o router.

El objetivo de este test es comprobar que la carga aplicada (tráfico generado) mediante el método `generateFlit()` sea la apropiada, teniendo en cuenta la probabilidad de inyección, parámetro del sistema. Primeramente, hay que evitar las dependencias con otras clases que posee `generatorModule.cc`, la clase principal del generador que será verificada. De este modo se ha de asegurar que se está probando única y exclusivamente la clase deseada, y evitar el comportamiento inesperado que puedan causar los módulos de los que depende.

Como se observa en la figura 3.1, la clase `generatorModule.cc` tiene una fuerte dependencia con la clase principal del switch, `switchModule.cc`, puesto que a la hora de generar un paquete, se ha de comprobar si el switch asociado al nodo generador tiene espacio disponible en los buffers para aceptar el dato, medido con créditos. Este switch se recibe por parámetro en el constructor del generador. Además, en caso de que se haya podido generar el paquete, a la hora de inyectarlo en el switch se ha de llamar a la función `injectFlit` de `switchModule.cc`.

```

#include "switch/switchModule.h"
#include "flit/flitModule.h"
#include "gmock/gmock/gmock.h"
class mock_switchModule : public switchModule {
public:
    mock_switchModule(string name, int label, int aPos, int hPos,
        int ports, int vcCount): switchModule(name, label, aPos,
        hPos, ports, vcCount){}
    MOCK_METHOD3(getPortCredits, int(int port, unsigned short cos,
        vector<int> vc_array));
    MOCK_METHOD3(getCredits, int(int port, unsigned short cos, int
        channel));
    MOCK_METHOD3(injectFlit, void(int port, int vc, flitModule*
        flit));
};
  
```

Para solventar este problema se han empleado objetos *Mock* [12], que permiten simular el comportamiento de clases y métodos, utilizando la librería *googleMock*. El objetivo de usar este tipo de

objetos es conseguir que la clase `switchModule.cc` actúe de un modo controlado, concretamente sus métodos `getCredits`, `getPortCredits` e `injectFlit`, que son los que llama el generador. Dicho esto, se crea la clase `mock_switchModule.h`, que define las funciones anteriores y se puede visualizar en el cuadro anterior.

Como se muestra en la figura 3.1, la clase `Mock` debe implementar la interfaz `switchModule.h`, sus métodos han de llamarse igual y tener el mismo tipo de parámetros que la clase `switchModule.cc`. De esta manera, será posible crear el objeto `Mock` en el test y *engañar* al generador haciéndole creer que es la clase `switchModule.cc`. Así, cuando se crea el generador recibe el objeto de tipo `mock_switchModule.h` heredado de `switchModule.h` y a la hora de llamar a las funciones definidas, se ejecutarán las del `Mock`. No obstante, hay que tener en cuenta que las funciones han de estar definidas como virtuales en la interfaz del switch para permitir polimorfismo [13], y posibilitar la herencia en los métodos entre la clase base del switch y sus subclases. Es decir, conseguir que se ejecuten los métodos de la subclase `Mock` en lugar de los de la clase base del switch. Tanto `getCredits` como `getPortCredits` son virtuales porque se llaman desde otras subclases, pero `injectFlit` se ha tenido que redefinir en el código fuente como virtual. De este modo, como se observa en el diagrama de la figura 3.1 la clase `mock_switchModule.h` será *hermana* de `switchModule.cc` e *hija* de `switchModule.h`.

GoogleMock permite a los métodos del objeto `Mock` comportarse de diferentes maneras según se requiera. Por ejemplo, pueden retornar un valor específico indicado, devolverlo sólo un número definido de veces, elegir el orden en que se retornen los valores y otras variadas opciones. En este caso, el comportamiento del método `generateFlit` depende de si hay espacio o no en los buffers del switch, por lo que, aplicando técnicas de caja negra, los casos a probar serán dos: cuando hay espacio y cuando no hay espacio. Para ello, en el primer caso se necesita que los métodos `getCredits` y `getPortCredits` devuelvan un valor mayor que 8, ya que son los créditos disponibles necesarios para que quepa un paquete en el buffer del switch. En cuanto al método `injectFlit`, al ser tipo *void* simplemente no hará nada, puesto que es suficiente con que no se ejecute el original. En la prueba se devuelven siempre que se llamen a `getCredits` y `getPortCredits` 96 créditos, espacio que tienen inicializado por defecto los buffers. En el segundo caso, cuando no hay créditos disponibles, los métodos anteriores deberán devolver 0 y se verificará que no se ha generado ningún paquete. Gracias a esta técnica se consigue aislar el módulo del generador para ser probado por sí solo. La definición para este comportamiento de los métodos mencionados es sencilla:

```
using ::testing::Return;
EXPECT_CALL(*switchM, getCredits(pPos,0,channel)).WillRepeatedly(
    Return(96));
EXPECT_CALL(*switchM, getPortCredits(pPos,0,vct)).WillRepeatedly(
    Return(96));
EXPECT_CALL(*switchM, injectFlit(pPos,0,flit));
```

La probabilidad con la que se inyecta un paquete hay que dividirla entre el tamaño del flit, medido en phits (unidad de información mínima de un paquete), porque cada paquete tarda en generarse n ciclos, siendo n el tamaño del flit, inyectándose un phit por ciclo. El generador se ejecuta en cada ciclo, determinando aleatoriamente si debe generar un paquete, usualmente compuesto por 8 phits, de modo que con una probabilidad del 80 %, cada 100 ciclos se generarían 10 paquetes idealmente. Para probar que la carga aplicada es la adecuada, en este test se verifica que el número de flits (paquetes) generados no difiere en más de un 2 % de la carga idónea con respecto a la probabilidad de inyección. Es necesario establecer un margen debido a la naturaleza aleatoria del código, puesto que diferentes semillas para generar los números aleatorios dan lugar a diferentes ejecuciones y resultados, todos ellos válidos. Además, como en este test se simula la red durante 100.000 ciclos, se ha considerado que añadir un 2 % de margen respecto de la media es suficiente para asegurar que no se ha generado una carga excesivamente mayor o menor de la esperada. Asimismo, fijar un margen menor podría causar falsos negativos donde el test debería pasar sin detectar que ha habido un error en la carga generada, y sin embargo, el resultado del test mostraría un fallo en la carga.

A continuación se explicará la programación de este test. Primeramente, se inicializan las variables globales requeridas por `generatorModule.cc` a un valor coherente, y se ejecuta el generador durante un número determinado de ciclos. Una vez conocido el número de flits que se han generado, almacenados en una variable global, se realiza la comprobación estableciendo un margen, superior e inferior del 2 % (1.02 y 0.98 respectivamente). De este modo, se calcula el número de paquetes máximo y mínimo que no se puede exceder con las siguientes ecuaciones:

$$Flits_Max_Permitted = ((p/100)/packet_size) \cdot cycles \cdot margenSup \quad (3.1)$$

$$Flits_Min_Permitted = ((p/100)/packet_size) \cdot cycles \cdot margenInf \quad (3.2)$$

Siendo p la probabilidad de inyección de un paquete, $packet_size$ el tamaño del paquete (en phits), $cycles$ el número de ciclos totales ejecutados y $margen\{Sup, Inf\}$ los límites resultantes de aplicar un porcentaje a la media.

Calculados los límites que no se han de sobrepasar, se comprueba usando los *Assertions EXPECT_LT* de *Google Test*, superándose el test si el primer parámetro es menor (*Expect Less Than*) que el segundo, y fallando en caso contrario.

```
EXPECT_LT(g_tx_packet_counter, Flits_Max_Permitted);
EXPECT_LT(Flits_Min_Permitted, g_tx_packet_counter);
//g_tx_packet_counter es la variable global cuyo valor es la cantidad
  de paquetes que han sido generados.
```

3.2. Pruebas de integración

Como se mencionó anteriormente, el test de integración tiene como objetivo comprobar el funcionamiento de un módulo formado por diferentes componentes para verificar su correcto comportamiento en conjunto. En este caso, se ha probado que el destino de los paquetes generados sea el correcto, para lo cual entran en contacto las clases `generatorModule.cc` y `steady.cc`. Esta última se encarga de calcular el destino del paquete en función del tipo de tráfico empleado (sección 2.1.3). Se han elaborado tres pruebas de integración para diferentes patrones: un test utilizando tráfico uniforme y dos para tráfico adverso (aleatorio y no aleatorio).

En la elaboración de las pruebas se ha seguido la misma estructura para los tres tests. Primero, se inicializan las variables globales requeridas a un valor coherente para la prueba y se crea un array de los nodos conectados a la red cuyo valor, inicialmente 0, indica el número de paquetes destinados al mismo. Posteriormente, se generan de acuerdo a la carga indicada paquetes ejecutando el método `generateFlit` durante un número determinado de ciclos y se actualiza el array según los destinos determinados por el simulador. Finalmente, se calcula una media del número de paquetes destinados a cada nodo, y de manera similar al test unitario, se establece un margen que ningún nodo del array debe exceder con respecto a la media.

3.2.1. Test tráfico uniforme

Utilizando un patrón de tráfico uniforme, todos los nodos de la red deben tener un número equilibrado de paquetes asociados como destino. Este test es el más básico de los tres, ya que es suficiente con hacer la media de los valores del array de nodos y calcular un margen superior e inferior en función de la media, de manera similar al test anterior. En este caso, se ha aplicado un margen del 10 % ya que el test se efectuará con 1056 nodos en la red, un tamaño usado con frecuencia en verdaderas simulaciones, y debido a esta gran cantidad de nodos el número máximo y mínimo de paquetes asociados a los destinos puede diferir en rango amplio, necesitándose un margen con más holgura.


```

int max=0;
int min=INT_MAX;
int packagesNode=0;
int totalPackages=0;
for(int i=0;i<totalNodes;i++){
    packagesNode=nodes[i];
    if (packagesNode>max){
        max=packagesNode;
    }else if(packagesNode<min){
        min=packagesNode;
    }
    totalPackages=totalPackages+packagesNode;
}
//totalFlitsGenerated indica la cantidad de paquetes que han sido
//generados durante la prueba
double average=totalFlitsGenerated/totalNodes;
double minPermitted=average*0.90;
double maxPermitted=average*1.10;

```

Utilizando los *Assertions* que se muestran a continuación, la prueba fallará si hay algún nodo con más paquetes asociados que el máximo permitido, alguno con menos del mínimo permitido (ecuaciones 3.1 y 3.2) o si el número de paquetes que se han generado es diferente de los que tienen asociados todos los nodos. Esto último garantiza que no se ha generado ningún paquete sin destino o con uno fuera de la red.

```

EXPECT_LT(minPermitted,min);
EXPECT_GT(maxPermitted,max);
EXPECT_EQ(totalFlitsGenerated,totalPackages);

```

Para comprobar detalladamente el comportamiento de este test, en la figura 3.2 se observan los resultados de ejecutar la prueba durante durante 10.000.000 de ciclos con un resultado positivo. En este caso, la red tiene los parámetros $h = p = 4$ y $a = 8$, por lo que hay un total de 1056 nodos, resultantes de la fórmula que determina el número de nodos de cómputo de la red: $(a \cdot h + 1) \cdot a \cdot p = 1056$. Se utiliza una probabilidad de inyección del 90 % y una semilla con valor 1 para generar los números aleatorios. Las líneas rojas marcan el número máximo y mínimo de paquetes que ningún nodo debe exceder (maxPermitted y minPermitted).

En otra ejecución durante menos ciclos, esta vez de 1.000.000, el test no es superado como puede apreciarse en la figura 3.3. Este resultado es debido a que no ha transcurrido un número necesario de ciclos y el tráfico no está lo suficientemente equilibrado en la red porque hay nodos que tienen más paquetes asociados que el máximo permitido y menos que el mínimo. Si se requiere probar la red durante menos ciclos, se debe aplicar un margen menos restrictivo, puesto que con el actual (10 %) falla habitualmente con menos de 10.000.000 de ciclos ejecutados.

Puede comprobarse en ambas gráficas que todos los nodos de la red reciben tráfico. En principio, esto debería ser imposible porque el nodo generador de tráfico nunca se envía mensajes a sí mismo (carece de sentido), por lo que para este test se ha escogido una etiqueta que identifica a un nodo inexistente de la red como generador. Por ejemplo, si la red dispone de 1056 nodos (generadores), se escoge un identificador fuera del rango, como el 1200.

3.2.2. Test tráfico adverso

La prueba de integración del generador utilizando un tráfico adverso (sección 2.1.3.2) en la red es semejante a la anterior, con la variante de que ahora el tráfico solo se genera hacia un único switch destino y se distribuye aleatoriamente entre sus nodos asociados. Se ha de verificar que el switch

destino sea el correcto y que el tráfico sea uniforme entre ellos. Por lo tanto, hay que tener en cuenta que la media a la que aplicaremos el margen superior e inferior se calcula en función de únicamente los nodos destino que tienen paquetes asociados (parámetro de la red p). Para calcular el destino de los mensajes con este patrón, entra en juego una nueva variable, la distancia al grupo destino cuyo valor define el número de grupos que salta el paquete respecto del grupo origen. Esta distancia puede ser mayor que el número de grupos que forman la red, así que para averiguar el grupo destino del flit se hace de la siguiente manera:

$$\text{grupos totales} = h \cdot a + 1$$

$$\text{grupo destino} = h_{Pos_origen} + \text{distancia mod grupos totales}$$

Conocido el grupo, el switch destino se corresponde con la misma posición que el switch origen dentro del grupo asociado al nodo generador:

$$\text{switch destino} = \text{grupodestino} \cdot a + a_{Pos_origen}$$

De manera análoga al test de tráfico uniforme, para realizar esta prueba se generan paquetes durante un número determinado de ciclos y además se comprueba, en cada ciclo que se ha generado un flit, que el grupo y switch destinos sean los correctos. A diferencia del tráfico uniforme, todos los paquetes son dirigidos a un número reducido de nodos $= p$, por lo que cada uno de ellos tiene muchos más paquetes asociados ejecutando los mismos ciclos con la misma probabilidad de inyección, con respecto al tráfico uniforme. Consiguientemente, el número de paquetes asociados a cada destino tiende a aproximarse más al promedio con respecto a la probabilidad. Debido a esto, se reduce el margen que admite el test con respecto de la media obtenida del número paquetes asociados a los nodos, consiguiendo unos límites más razonables. En la figura 3.4 se muestra el resultado de ejecutar el test durante 1.000.000 de ciclos con los mismos parámetros de la red que la prueba anterior ($2h = 2p = a = 8$), pero esta vez aplicando un margen superior e inferior de un 2 % de la media (ecuaciones 3.1 y 3.2). Este margen es menor con respecto al test anterior porque hay muchos menos nodos que reciben los mensajes, y como se simula durante los mismos ciclos, cada uno de ellos recibe un mayor número de paquetes y tiende a aproximarse más a la media ideal. En este caso, el nodo generador

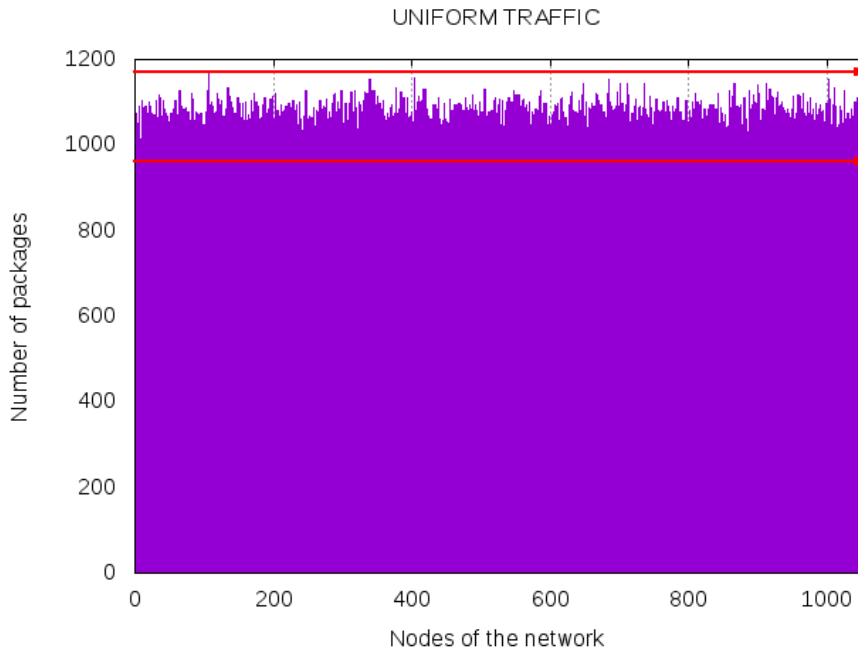


Figura 3.2: Test superado con tráfico uniforme en la red. Ciclos ejecutados=10.000.000, seed=1, probabilidad de inyección=90 %.

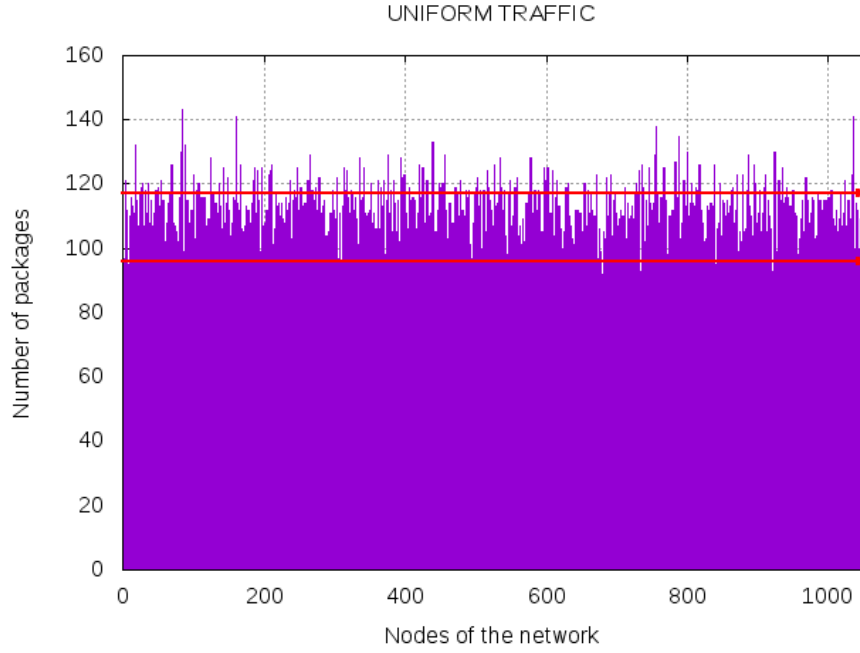


Figura 3.3: Test fallido con tráfico uniforme en la red. Ciclos ejecutados=1.000.000, seed=23, probabilidad de inyección=90 %.

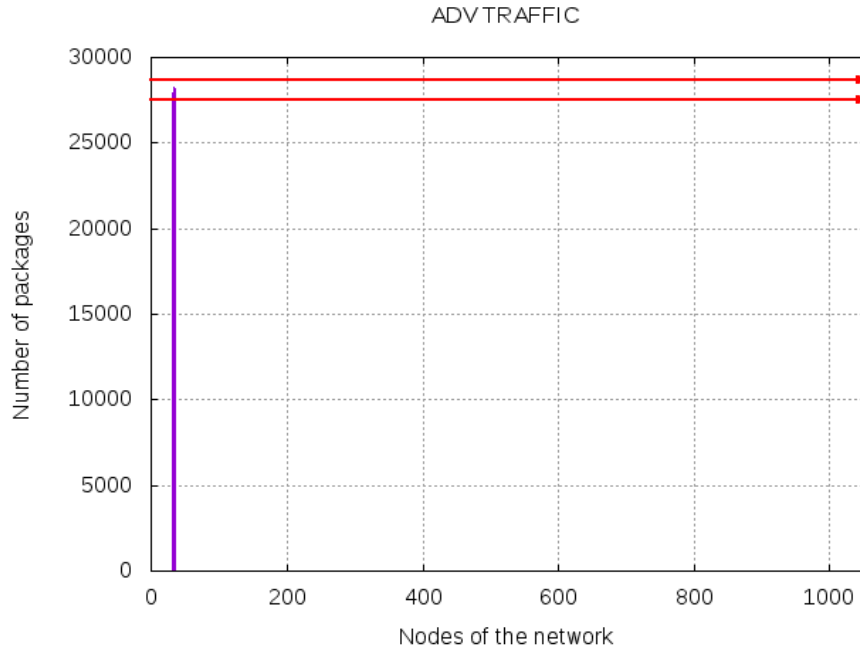


Figura 3.4: Test superado con tráfico adverso no aleatorio en la red. Ciclos ejecutados=1.000.000, seed=57, probabilidad de inyección=90 %.

está conectado al primer switch del primer grupo (ambos identificados con valor 0) de la red y se ha utilizado una distancia con valor 1, por lo que el switch destino es el número $1 \cdot a + a_{Pos} = 8$. Consiguientemente, los nodos destinos serán $8 \cdot p = 32$, hasta el 35 ($p = 4$).

El último test de integración elaborado se ha realizado para un tráfico adverso aleatorio en la red (sección 2.1.3.2), por lo que ahora los paquetes se dirigen hacia todos los switches de un único grupo, repartidos de manera uniforme entre ellos. El switch destino de cada paquete se determina de manera aleatoria. Asimismo, el nodo final se escoge de modo aleatorio de entre todos los nodos asociados al

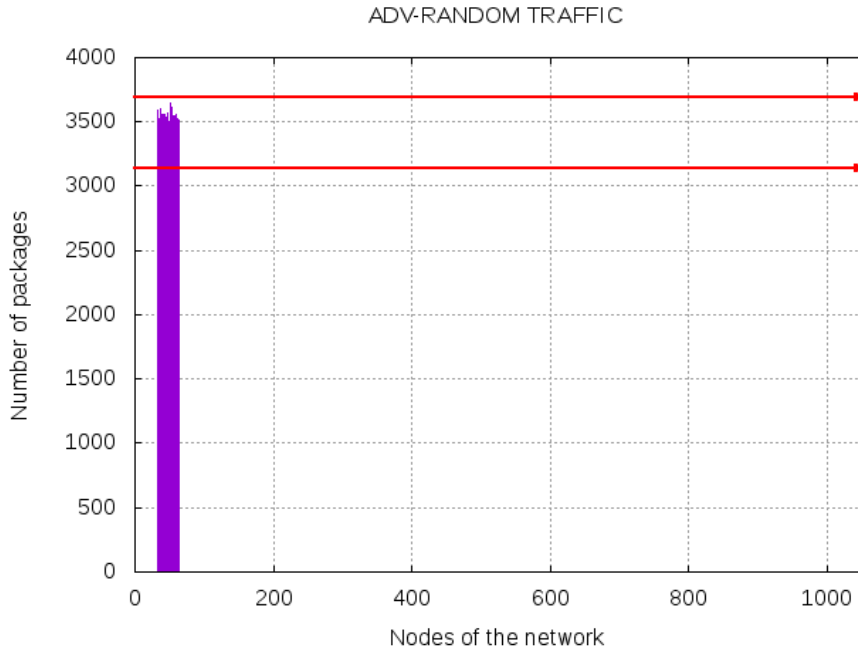


Figura 3.5: Test superado con tráfico adverso aleatorio en la red. Ciclos ejecutados=1.000.000, seed=18, probabilidad de inyección=90 %.

switch destino. En este caso, la media de paquetes asociados a cada nodo a la que aplicaremos los márgenes se calcula en función del número de grupos que componen la red, ya que todos los flits generados se dividen aleatoria y uniformemente entre todos los nodos de un único grupo. En el test se realizan las mismas comprobaciones a excepción del switch destino porque es aleatorio, y se han establecido los límites teniendo en cuenta un 8 % de la media resultante de la ejecución (ecuaciones 3.1 y 3.2). Este porcentaje es razonable porque el número de nodos destino es menor que con tráfico uniforme y mayor que con tráfico adverso. Para comprobar el resultado del test, se han ejecutado 1.000.000 de ciclos con una semilla de 18 y probabilidad de inyección igual a 90 % (figura 3.5). Los parámetros de la red son $h = p = 4$ y $a = 8$, por lo que la cantidad de nodos destino es $a \cdot p = 32$ (número de nodos por grupo). El generador, al igual que anteriormente, se encuentra en el primer grupo, así que con una distancia = 1, el grupo destino es el 1 (segundo grupo de la red). Concretamente, los nodos que reciben los paquetes son desde el $a \cdot p \cdot distancia = 32$ (primer nodo del segundo grupo) hasta el 63 (32 nodos/grupo). Las aserciones empleadas para determinar el resultado de la prueba son similares a las del test de tráfico uniforme.

3.3. Prueba de sistema

En esta prueba se pretende verificar el simulador como un sistema completo utilizando todos sus módulos (ejecuciones reales). Para ello, se ha elegido la latencia y el throughput como parámetros a comprobar porque son los resultados más elementales e importantes en la medida del rendimiento de la red que devuelve FOGSim. Consiguientemente, se ha elaborado un script en lenguaje Python (Anexo C) que lee la configuración de la red (probabilidad, tipo de tráfico y mecanismo de routing) del archivo de salida (véase un ejemplo en Anexo B) del simulador y en base a ello escoge una latencia y un throughput de referencia para compararlos con los resultados obtenidos en la ejecución.

Para hallar estos valores de referencia de ambas medidas en las diferentes configuraciones (y debido a la imposibilidad de calcularse de manera analítica), se ha ejecutado el simulador (versión libre de errores) varias veces con cada una de ellas y se han guardado los resultados para poder calcular la media en cada configuración. En cuanto al intervalo de confianza admitido, en esta ocasión y a

diferencia de las pruebas anteriores en las que se establecía un margen fijo, se ha optado por hacer la prueba más exhaustiva ya que ahora se tiene en cuenta el comportamiento del sistema completo, lo que puede ocasionar mayores diferencias en los resultados, además de obtener distintos resultados en cada configuración diferente. Por ello, se ha elegido la desviación típica (raíz cuadrada de la varianza) como medida de dispersión [14] para aplicárselo a la media con el objetivo de conseguir que las medidas de referencia sean más precisas. Las ecuaciones 3.3, 3.4, y 3.5 han sido empleadas en el script y determinan la media m , varianza σ^2 y desviación típica σ respectivamente. Se denomina n al número de ejecuciones y x_i al valor de la latencia o throughput de cada ejecución.

$$m = \frac{1}{n} \sum_{i=1}^n x_i \quad (3.3)$$

$$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - m)^2 \quad (3.4)$$

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - m)^2} \quad (3.5)$$

Una vez se ha realizado la ejecución del simulador, se lee la configuración que ha sido empleada del fichero de salida, y la latencia y throughput resultantes. En el script están almacenados los resultados de throughput y latencia en listas, obtenidos en varias ejecuciones (una por cada configuración y medida diferente), permitiendo añadir más resultados o modificarlos en un futuro. Los valores de la media y la desviación típica se calculan en base a las listas (tanto la de throughputs como la de latencias) correspondientes a la configuración empleada en la ejecución que se quiera probar. Las diferentes configuraciones empleadas se muestran en las tablas 3.1, 3.2, 3.3, y 3.4 así como sus resultados obtenidos (almacenados en las listas del script). También se muestra la media y desviación típica resultantes. Se han realizado un total de cinco ejecuciones por cada configuración empleando diferentes semillas (1, 10, 15, 20 y 25). Los parámetros de red $h = p = 4$ y $a = 8$ durante 100.000 ciclos, de los cuales 50.000 han sido empleados para el calentamiento de la red.

Tráfico uniforme en la red utilizando routing mínimo.

Probabilidad de inyección	Latencia (ciclos)	Media	Desviación típica	Throughput (phits/(nodo*ciclo))	Media	Desviación típica
10 %	136.436	136.449	0.0325	0.100117	0.0999	0.0000885
	136.494			0.0999462		
	136.471			0.0998797		
	136.413			0.0999786		
	136.433			0.0999389		
40 %	146.008	146.0002	0.0308	0.400066	0.40005	0.00021
	146.023			0.400228		
	145.968			0.399989		
	146.034			0.400261		
	145.968			0.399732		
100 %	1053.87	1053.708	1.5602	0.729039	0.7282	0.00061
	1055.3			0.72789		
	1054.94			0.72742		
	1052.99			0.728484		
	1051.44			0.728293		

Tabla 3.1: Resultados de varias ejecuciones de latencia y throughput con tráfico uniforme y routing mínimo.

Tráfico uniforme en la red utilizando routing piggybacking.

Probabilidad de inyección	Latencia (ciclos)	Media	Desviación típica	Throughput (phits/(nodo*ciclo))	Media	Desviación típica
10 %	146.37	146.327	0.03447	0.10012	0.09995	0.000103
	146.303			0.0999364		
	146.307			0.0998342		
	146.36			0.099937		
	146.298			0.0999588		
40 %	169.912	169.893	0.04380	0.400019	0.4001	0.000273
	169.918			0.400331		
	169.824			0.399974		
	169.935			0.400532		
	169.88			0.39988		
100 %	1337.6	1338.161	1.2484	0.673067	0.6729	0.000534
	1339.8			0.672443		
	1338.94			0.672338		
	1337.91			0.673573		
	1336.56			0.673284		

Tabla 3.2: Resultados de varias ejecuciones de latencia y throughput con tráfico uniforme y routing piggybacking.

Como parámetro del script, además del fichero de salida del simulador, se ha de indicar un número real que será un multiplicador a la desviación típica. Este producto es sumado y restado a la media obtenida de las ejecuciones para determinar unos límites máximo y mínimo, respectivamente. El valor que se debe establecer a este multiplicador varía en función de la mayor o menor amplitud que se desee aplicar a los límites, por lo que tiene un fuerte impacto en el resultado del test. Para conseguir un resultado estadísticamente significativo, es decir, que sea improbable que haya sido obtenido debido al azar, y evitar falsos negativos (el test falla cuando el caso teórico es cierto), un valor razonable es de 3, ya que un desvío de 3σ respecto de la media, implica, atendiendo a la distribución gaussiana, que la probabilidad de obtener falso negativo es del 0,27 % (100 % - 99,47 %)[15].

A modo de ejemplo y con el objetivo de corroborarlo, la gráfica 3.6 muestra para qué valor del multiplicador (número de σ) el test comienza a resultar fallido con las configuraciones indicadas. En concreto, se ha ejecutado el simulador con routing mínimo y tráfico uniforme, routing piggybacking y tráfico uniforme y adverso aleatorio, y routing valiant con tráfico adverso aleatorio, con diferentes probabilidades en cada caso. Para las ejecuciones cuyo resultado se refleja en la figura 3.6 se han empleado semillas distintas en cada caso para crear distintos números aleatorios.

Finalmente, el script comprueba que tanto la latencia como el throughput obtenidos como resultado de la ejecución, no superan el límite máximo ni sobrepasan el mínimo, en cuyo caso se da por aprobado el test; de otro modo, el test falla indicando el valor o los valores incoherentes. La configuración base empleada para la obtención de los resultados (fichero de entrada que necesita FOGSim) se muestra en el Anexo A. Asimismo, esta configuración es la que se debe utilizar en futuras pruebas del simulador, eso sí, para los diferentes tipos de tráfico, mecanismos de routing y probabilidades de inyección que han sido testeados.

Tráfico adverso aleatorio en la red utilizando routing piggybacking.

Probabilidad de inyección	Latencia (ciclos)	Media	Desviación típica	Throughput (phits/(nodo*ciclo))	Media	Desviación típica
10 %	255.306	255.308	0.02976	0.100102	0.09997	0.00009825
	255.337			0.099963		
	255.272			0.0998305		
	255.339			0.0999702		
	255.287			0.100013		
40 %	370.536	371.898	0.91654	0.399936	0.39984	0.00030858
	372.089			0.399985		
	371.481			0.399761		
	372.817			0.400192		
	372.569			0.39937		
100 %	1031.9	1032.027	0.19929	0.41931	0.4191	0.00030856
	1032.02			0.419335		
	1031.8			0.418573		
	1032.1			0.41915		
	1032.32			0.419136		

Tabla 3.3: Resultados de varias ejecuciones de latencia y throughput con tráfico adverso aleatorio y routing piggybacking.

Tráfico adverso aleatorio en la red utilizando routing valiant

Probabilidad de inyección	Latencia (ciclos)	Media	Desviación típica	Throughput (phits/(nodo*ciclo))	Media	Desviación típica
10 %	265.576	265.598	0.040530	0.100118	0.09996	0.0001134
	265.572			0.0999523		
	265.645			0.0998011		
	265.559			0.0999473		
	265.639			0.099993		
40 %	387.495	387.462	1.36883	0.39959	0.39951	0.0004218
	388.663			0.399818		
	385.75			0.399974		
	396.475			0.39923		
	388.927			0.398947		
100 %	1084.45	1083.688	0.95127	0.395699	0.39566	0.0007459
	1083.05			0.396097		
	1083.83			0.396617		
	1084.69			0.395186		
	1082.42			0.394715		

Tabla 3.4: Resultados de varias ejecuciones de latencia y throughput con tráfico adverso aleatorio y routing valiant.

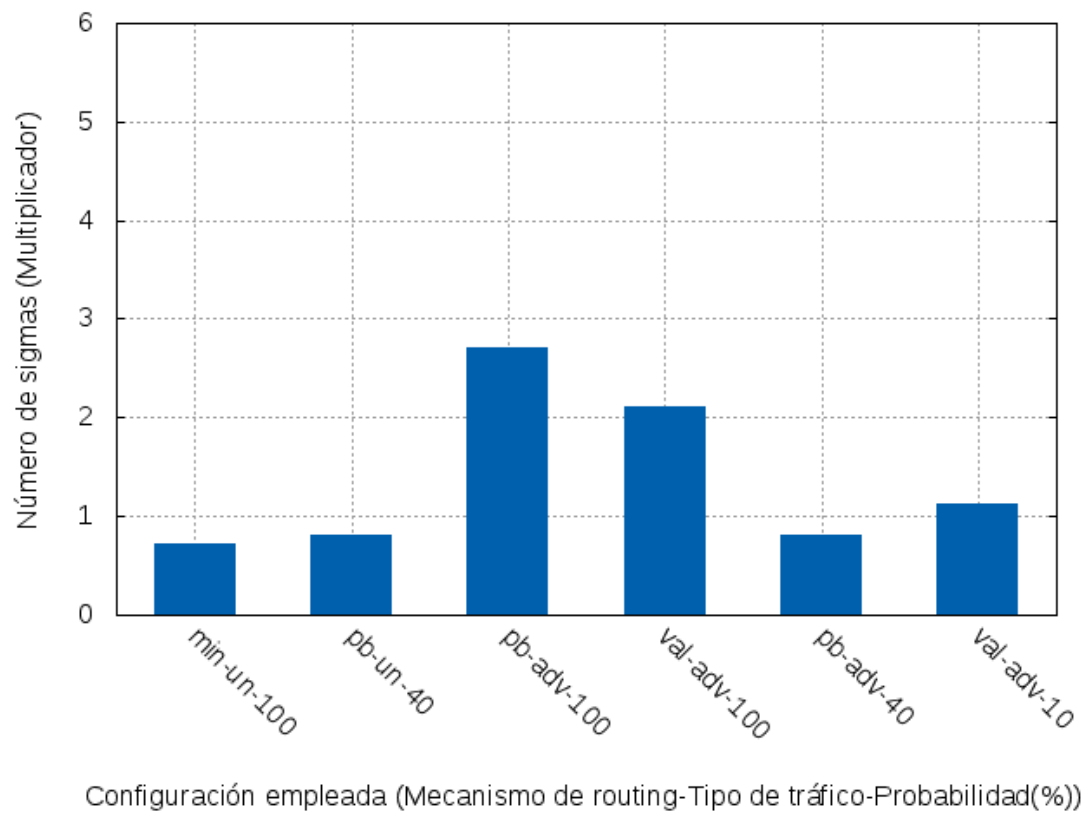


Figura 3.6: Valores del multiplicador de la desviación típica que hay que aplicar para que el test no detecte fallos en las diferentes pruebas concretas con las configuraciones indicadas.

Capítulo 4

Automatización con Jenkins

En el presente capítulo se detallará el procedimiento seguido para conseguir aplicar integración continua al simulador, es decir, cómo se ha logrado ejecutar las pruebas que verifican el software de manera automática y transparente al programador. Para conseguir este objetivo, es indispensable que el código del simulador se aloje en un repositorio de control de versiones donde se integren los cambios realizados por los programadores y existan diferentes versiones del software. Como se ha mencionado anteriormente, el código se aloja en el servicio de almacenamiento Bitbucket basado en Git. Para lograr esta automatización del proceso se ha empleado la herramienta Jenkins, destinada a la integración continua.

4.1. Jenkins y programación de tareas

Como se ha mencionado en la sección 2.4.6, Jenkins se basa en la programación y ejecución de diferentes tareas independientes. Para cada tarea hay que definir dónde reside el código fuente sobre el que se trabajará, en qué momentos se ha de ejecutar la tarea y qué hacer en la misma, con una amplia variedad de opciones como compilar, ejecutar programas, comandos o scripts, enviar e-mails, etc.

Primeramente, se ha instalado el servidor local Jenkins en Ubuntu [16]. Tras la instalación, por defecto el servicio escucha en el puerto 8080, por lo que para acceder a Jenkins es suficiente con escribir la dirección `http://localhost:8080` en un navegador desde el equipo local. Si se necesita acceder desde otra máquina se puede establecer una conexión mediante ssh al equipo donde se aloja Jenkins.

El siguiente paso es crear y programar las tareas. Debido al tiempo de ejecución de los diferentes tests elaborados, se ha elegido dividir el proceso de ejecución de las pruebas en dos tareas de Jenkins. Por un lado, la primera consistirá en ejecutar los tests unitarios y de integración. Puesto que su tiempo de ejecución no es muy largo, es interesante realizarlos cada vez que alguien aplica un cambio en el repositorio. Para ello, el controlador de versiones deberá notificar a Jenkins cuando ocurra algún cambio en el repositorio.

Por otro lado, la segunda tarea radicará en efectuar la prueba de sistema. Ya que su tiempo de ejecución es más pesado que los otros tests, se ha considerado suficiente con ejecutarla una vez al día en caso de que se haya habido cambios en el simulador. Para poder programarlo, Jenkins utiliza una estrategia de *polling*, de manera que con cierta periodicidad (configurable) se comprueba si ha habido cambios en el repositorio y en ese caso, se ejecuta la tarea correspondiente. En caso contrario no se hará nada porque no se ha modificado el código del simulador y carece de sentido ejecutar las pruebas.

A continuación se explicará la configuración de ambas tareas. Jenkins posee una interfaz intuitiva mediante la cual se han creado las dos tareas y definido las opciones básicas (nombre, descripción...). Para que Jenkins sepa dónde tiene que acceder y así obtener el código sobre el que operar, se ha de indicar el origen del código fuente (en este caso, Git) y la URL del repositorio (.git). Jenkins también

permite autenticarse por parte del usuario para acceder al repositorio en caso de que el repositorio sea privado y se necesiten credenciales.

La tarea de la prueba de sistema requiere ser programada para que se ejecute una vez al día en caso de que haya habido modificaciones. Para configurar cuándo se debe ejecutar la tarea, se observa en la figura 4.1 que se ha seleccionado la opción “Consultar repositorio (SCM)”, que consiste en hacer *polling* con una frecuencia determinada. Esta frecuencia se indica en el cuadro de texto mostrado en la figura, regida por una sintaxis concreta, la cual consta de cinco campos separados por un espacio haciendo referencia a minutos, horas, días del mes, meses, y días de la semana respectivamente. El signo “*” indica todos los valores posibles, y “H” significa cualquier valor único de ese campo. Por ejemplo, con una configuración de “H * * * *” se ejecutaría la tarea una vez cada hora, durante todas las horas de todos los días y meses. En la presente tarea, se ha empleado una frecuencia de “H H * * *” de modo que se realice en un minuto y hora concreta durante todos los días y meses. El símbolo “H” puede considerarse como un valor aleatorio en un rango, pero en realidad es un hash (correspondencia) del nombre de la tarea, no una función aleatoria, de modo que el valor permanece estable para cualquier tarea.

Disparadores de ejecuciones

- ☐ Lanzar ejecuciones remotas (ejem: desde 'scripts')
- ☐ Build after other projects are built
- ☐ Build when a change is pushed to BitBucket
- ☐ Build when a change is pushed to GitLab. GitLab CI Service URL: <http://127.0.0.1:8080/project/Prueba%20sistema%20FOGSim>
- ☒ Consultar repositorio (SCM)

Programador:

Would last have run at jueves 22 de junio de 2017 12H35' CEST; would next run at viernes 23 de junio de 2017 12H35' CEST.

☐ Ignore post-commit hooks

- ☐ Ejecutar periódicamente
- ☐ GitHub hook trigger for GITScm polling
- ☐ Perforce triggered build.

Figura 4.1: Captura de pantalla de la configuración de la tarea encargada de ejecutar la prueba de sistema. Concretamente, la frecuencia con la que se ejecuta.

En el caso que nos ocupa, la tarea deberá compilar el código del simulador, realizar una simulación con el fichero de parámetros de entrada apropiado y ejecutar el script con los argumentos requeridos del test de sistema. Para ello, se ha de indicar que se ejecuten una serie de comandos por consola, como se muestra en la figura 4.2, donde se compila el código, se ejecuta la simulación y finalmente el test de sistema.

La tarea correspondiente a la prueba de sistema se ha programado para que Jenkins conozca dónde se aloja el código fuente y pueda acceder a él. Sin embargo, como la tarea de las pruebas unitarias y de integración se debe ejecutar cada vez que ocurra alguna modificación, es necesario configurar el servicio web donde se hospeda el código para avisar a Jenkins en el momento que hayan ocurrido cambios en el repositorio. Para conseguirlo, se ha de añadir un *webhook* en el servicio web de controlador de versiones, donde se indica la dirección URL del proyecto/tarea de Jenkins. Gracias a esto, el gestor de repositorios web enviará una petición POST mediante el protocolo *http* a la URL especificada en el *webhook* cuando haya ocurrido algún evento definido en el mismo. Como interesa avisar a Jenkins cuando haya habido algún cambio en el código, se debe indicar en el *webhook* el evento Push, es decir, cuando ocurra un *push* en el repositorio (actualización de ficheros), existiendo otros como añadir comentarios, creación o actualización de la wiki, etc.



Figura 4.2: Captura de pantalla de la configuración de la tarea encargada de ejecutar la prueba de sistema. Concretamente, la ejecución de comandos mediante consola.

Para conseguir establecer una comunicación entre Jenkins y el servicio web de hospedaje (actualmente alojado en Bitbucket), es necesario disponer de una dirección IP pública donde instalar y alojar las tareas de Jenkins la cual sería especificada en el *webhook* del proyecto del gestor de repositorios. Debido a la imposibilidad de disponer de una IP pública, se ha decidido instalar el servidor GitLab en local pensando en una futura migración del código a éste (y dada la imposibilidad de instalar Bitbucket en local), para permitir la comunicación con Jenkins (también en local) y verificar así su funcionamiento en conjunto. Si en algún momento se contase con una IP pública, simplemente habría que modificar la URL indicada en el *webhook*. Para instalar el servidor GitLab local es suficiente con ejecutar el comando `$ apt-get install gitlab`. Por defecto, el proceso escucha en el puerto 8080, el mismo que utiliza Jenkins, por lo que se ha modificado al 8081 a través del fichero de configuración `/etc/gitlab/gitlab.rb`. Una vez instalado, se ha clonado el código del simulador al servidor local para verificar la comunicación después de aplicar un *push* en el repositorio. Además, se ha añadido el *webhook* al proyecto en GitLab (*Settings* → *Integrations*), mostrado en la figura 4.3. La dirección IP 127.0.0.1 es la resolución de localhost, por lo que es equivalente.

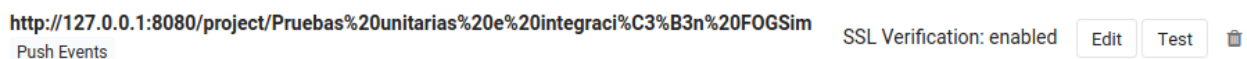


Figura 4.3: Captura de pantalla del *webhook* creado que avisa a la tarea de Jenkins después de aplicar un evento *push* (cambios en los ficheros del repositorio).

Con el objetivo de comprobar su correcto funcionamiento, se ha modificado el fichero de entrada del simulador y se ha actualizado el repositorio para comprobar que se ejecutan las pruebas. Para ello se han aplicado cambios en el repositorio de GitLab a través de la consola. Mediante la acción *pull* se sincroniza los archivos del repositorio con los locales (directorio actual), *add* y *commit* indican los ficheros que se han modificado y permite añadir una descripción del cambio, y finalmente con el comando *push* se suben los cambios al repositorio. De este modo se comprueba que Jenkins detecta los cambios efectuados en GitLab y, en consecuencia, ejecuta las pruebas.

4.2. Resultados

Después de haber definido, configurado y ejecutado las tareas adecuadamente, el panel de control de Jenkins resultante se muestra en la figura 4.4.

A continuación, se mostrará el resultado obtenido después de ejecutar la tarea de la prueba de sistema después de haber aplicado algún cambio en el simulador. En el primer caso (figuras 4.5 y 4.6) la tarea de Jenkins resulta fallida debido a que el script de la prueba de sistema no está subido en





<div> <div>Todo</div> <div>+</div> </div>			
S	W	Nombre ↓	Último Éxito
		Prueba sistema FOGSim ▼	19 Hor - #7
		Pruebas unitarias e integración FOGSim	20 Hor - #101

Figura 4.4: Captura de pantalla del panel de control de Jenkins una vez definidas la tarea de la prueba de sistema y la de las pruebas unitarias e integración.

el repositorio del proyecto (GitLab). La ejecución correcta de la tarea se muestra en la figura 4.7, obteniendo además un resultado en el que no se detectan defectos tras la ejecución del test de sistema.

S	W	Nombre ↓
		Prueba sistema FOGSim

Figura 4.5: Captura de pantalla del panel de control de Jenkins tras una ejecución fallida de la tarea.

cycle: 98900	Messages sent: 1305124	Messages received: 1303444
cycle: 99000	Messages sent: 1306453	Messages received: 1304748
cycle: 99100	Messages sent: 1307827	Messages received: 1306091
cycle: 99200	Messages sent: 1309117	Messages received: 1307416
cycle: 99300	Messages sent: 1310387	Messages received: 1308727
cycle: 99400	Messages sent: 1311694	Messages received: 1310029
cycle: 99500	Messages sent: 1313014	Messages received: 1311332
cycle: 99600	Messages sent: 1314359	Messages received: 1312646
cycle: 99700	Messages sent: 1315672	Messages received: 1314033
cycle: 99800	Messages sent: 1316999	Messages received: 1315318
cycle: 99900	Messages sent: 1318345	Messages received: 1316598
Write output		
Freed memory		
Simulation finished		
+ ./testSystem.py output 4		
/tmp/hudson8589260930747899150.sh: 4: /tmp/hudson8589260930747899150.sh: ./testSystem.py: not found		
Build step 'Ejecutar línea de comandos (shell)' marked build as failure		
Finished: FAILURE		

Figura 4.6: Captura de pantalla de la salida por consola como resultado de una ejecución fallida de la tarea debido a la inexistencia del fichero del script de la prueba de sistema.

Como curiosidad, Jenkins valora el estado de las últimas ejecuciones de las tareas con el estado del clima, y en base al resultado de la ejecución indica con un sol si han acabado con éxito últimamente o con tormenta en caso contrario.

```
cycle: 99500 Messages sent: 1313014 Messages received: 1311332
cycle: 99600 Messages sent: 1314359 Messages received: 1312646
cycle: 99700 Messages sent: 1315672 Messages received: 1314033
cycle: 99800 Messages sent: 1316999 Messages received: 1315318
cycle: 99900 Messages sent: 1318345 Messages received: 1316598
Write output
Freed memory
Simulation finished
+ ./testSystem.py output 4
The traffic is uniform
The routing is minimal
The probability is 10
Latency reference (average latency): 136.4494 --- Output latency: 136.413

Throughput reference (average throughput): 0.09997208 --- Output throughput: 0.0999786
The Latency Standard Deviation is 0.0325161498336
The Throughput Standard Deviation is 8.85283400951e-05
MAX Latency permitted: 136.579464599
MIN Latency permitted: 136.319335401
MAX Throughput permitted: 0.10032619336
MIN Throughput permitted: 0.0996179666396
Latency is correct!
Throughput is correct!
TEST PASSED --> Latency and Throughput are correct. Well done! :)
Finished: SUCCESS
```

Figura 4.7: Captura de pantalla de la salida por consola como resultado de una ejecución correcta de la tarea de la prueba de sistema, el resultado exitoso del test de sistema.

Capítulo 5

Conclusiones y trabajos futuros

En este breve y último capítulo del informe se mostrarán las conclusiones obtenidas tras la realización del presente trabajo, tanto a nivel técnico como a nivel personal. Además, se expondrán las posibles mejoras futuras que pueden aplicarse al framework elaborado para conseguir testear el simulador de un modo más completo.

5.1. Conclusiones

El objetivo de este proyecto es realizar un framework que verifique el funcionamiento de un simulador de redes de interconexión y asegurar al programador que los cambios aplicados para mejorar el rendimiento de la red no desencadenen resultados inesperados. De este modo y como ha ocurrido en ciertas ocasiones, se pretende evitar que el usuario invierta un tiempo excesivo en buscar y encontrar posibles errores. Gracias al framework construido y las pruebas elaboradas que lo componen, se ha conseguido cumplir dicha finalidad sobre un subconjunto de módulos del simulador debido a la excesiva cantidad de componentes que lo forman. Además, para evitar al programador que deba ejecutar los tests manualmente se ha logrado automatizar el proceso de verificación del código después de haber aplicado cambios sobre el simulador de modo que se pruebe de un modo transparente al usuario. De esta manera, se ha introducido el concepto de integración continua sobre un software que carecía de esta buena práctica.

En lo referente al apartado técnico, este trabajo me ha proporcionado nuevos conocimientos relacionados con dos especialidades de la ingeniería informática. Por un lado y relativo a las redes de computadores, he aprendido de un modo básico el funcionamiento de una red de interconexión de tipo Dragonfly, lo que ha requerido ciertas lecturas de documentos científicos. Asimismo, he aprendido el funcionamiento básico de un simulador empleado por un grupo de investigación en esta universidad. Por otro lado y en relación a la ingeniería software, he obtenido conocimientos más profundos a cerca del testing software y diferentes técnicas utilizadas en el mismo, así como el aprendizaje del lenguaje de programación C++. También he aprendido una buena práctica que ha de ser utilizada en todo buen desarrollo software, la integración continua y cómo aplicarla a un software para conseguir que el desarrollo sea menos costoso y evadir futuros problemas.

En cuanto a nivel personal, he disfrutado con la realización del proyecto y estoy satisfecho con el resultado conseguido y conocimientos aprendidos durante este período. Además, resulta gratificante que el framework elaborado pueda sea de utilidad para los usuarios del simulador.

También, he conocido el *modus operandi* de los investigadores pertenecientes al grupo ATC y me he visto involucrado en su ámbito de trabajo de investigación.

5.2. Trabajos futuros

Como se ha mencionado anteriormente, el simulador está formado por una gran cantidad de módulos, y ha sido inviable abarcar todos sus componentes en el framework de pruebas. Por lo tanto, se puede añadir más funcionalidad añadiendo más pruebas unitarias para probar el resto de módulos de FOGSim de manera aislada, y más pruebas de integración sobre diferentes subconjuntos de sus componentes. Sería suficiente con programar nuevos tests en el proyecto del framework escogiendo los módulos deseados y comparando los resultados obtenidos en la ejecución con los esperados. Así se conseguiría verificar el simulador de un modo más completo y detectar otros posibles errores ubicados en distintas zonas. Por ejemplo, otro módulo interesante a probar que tiene gran importancia en el comportamiento de la red sería el de los buffers, de forma que se verifique que los buffers de los switches funcionan de la manera apropiada ante diferentes patrones de tráfico. Además, para la puesta en marcha del framework, se debe migrar el repositorio de Bitbucket a GitLab, donde se ha verificado el correcto funcionamiento en local, y disponer de una dirección IP pública donde instalar Jenkins para permitir la comunicación entre ellos.

Por otro lado, debido a las dificultades encontradas por las interrelaciones entre módulos, se ha descubierto que se podrían aplicar algunas refactorizaciones para mejorar el código, de forma que fuera más mantenible en el futuro y de cara a la mejora y extensión del framework de verificación.

Bibliografía

- [1] M. Garcia, E. Vallejo, R. Beivide, M. Odriozola, C. Camarero, M. Valero, J. Labarta, C. Minckenberg, *et al.*, “On-the-fly adaptive routing in high-radix hierarchical networks,” in *Parallel Processing (ICPP), 2012 41st International Conference on*, pp. 279–288, IEEE, 2012.
- [2] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.
- [3] M. García González, *Mecanismos de routing para redes de interconexión dragonfly*. PhD thesis, Universidad de Cantabria, 2014.
- [4] N. Jiang, J. Kim, and W. J. Dally, “Indirect adaptive routing on large scale interconnection networks,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09*, (New York, NY, USA), pp. 220–231, ACM, 2009.
- [5] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. O’Reilly Media, Inc., 2012.
- [6] M. Fowler, “Continuous integration.” <https://www.martinfowler.com/articles/continuousIntegration.html>, 2006. Accessed: 2017-06-29.
- [7] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-driven, highly-scalable dragonfly topology,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08*, (Washington, DC, USA), pp. 77–88, IEEE Computer Society, 2008.
- [8] “Top 500 june 2017.” <https://www.top500.org/lists/2017/06/>. Accessed: 2017-06-27.
- [9] A. Singh, *Load-balanced routing in interconnection networks*. PhD thesis, Stanford University, 2005.
- [10] B. W. Boehm, “Verifying and validating software requirements and design specifications,” *IEEE software*, vol. 1, no. 1, p. 75, 1984.
- [11] “Google test.” <https://github.com/google/googletest>. Accessed: 2017-06-29.
- [12] T. Mackinnon, S. Freeman, and P. Craig, “Endo-testing: unit testing with mock objects,” *Extreme programming examined*, pp. 287–301, 2000.
- [13] P. Stevens, R. Pooley, and L. J. Aguilar, *Utilización de UML en Ingeniería del Software con Objetos y Componentes*, vol. 14. Addison Wesley, 2002.
- [14] M. C. Rodríguez, *Resumen Estadística*. Universidad de Cantabria, 2014.
- [15] S. S. Wilks, *Elementary statistical analysis*, pp. 144–147. Princeton University Press, 2015.
- [16] K. Kawaguchi, “Installing jenkins on ubuntu.” <https://wiki.jenkins.io/display/JENKINS/Installing+Jenkins+on+Ubuntu>. Accessed: 2017-06-29.

Anexos

Anexo A

Fichero de entrada tipo para el test de sistema

```
[CONFIG]
P=4
A=8
H=4
MaxCycles=50000
WarmupCycles=50000
Switch=IOQ_SW
XbarDelay=3
InternalSpeedup=2
ArbiterIterations=2
InjectionQueueLength=256
LocalQueueLength=32
GlobalQueueLength=256
OutQueueLength=32
LocalLinkTransmissionDelay=10
GlobalLinkTransmissionDelay=100
InjectionDelay=1
PacketSize=8
FlitSize=8
PalmTreeConfiguration=1
LocalLinkChannels=2
InjectionChannels=3
GlobalChannels=1
InputArbiter=RR
OutputArbiter=RR

#TRAFFIC PATTERN
Traffic=UN

#ROUTING
routing=MIN
deadlock_avoidance=DALLY
```

Anexo B

Fichero de salida ejemplo

PARAMETERS
Total Number Of Cycles: 100000
Max Cycles: 50000
H: 4
P: 4
A: 8
Switch: IOQ
Xbar Delay: 3
Out Queue Length: 32
Internal SpeedUp: 2
Number Segregated Traffic Flows: 1
InputArbiter: RoundRobin
OutputArbiter: RoundRobin
Arbiter Iterations: 2
Injection Delay: 1
Local Link Delay: 10
Global Link Delay: 100
Packet Size: 8 phits
Flit Size: 8 phits
Packet Flit Size: 1 flits
Generator Queue Length: 256
Buffer: SEPARATED
Local Queue Length: 32 (phits per vc)
Global Queue Length: 256 (phits per vc)
Total Local Buffer: 64 (phits per port)
Total Global Buffer: 256 (phits per port)
VCs: 3
Injection VCs: 3
Local Link VCs: 2
Global Link VCs: 1
Seed: 1
Palm Tree Configuration: 1

Latency Histogram Max Lat: 193
Hops Histogram Max Hops: 200
Input Speedup: 1
Parallel Req Issuing: 1
Injection Probability: 10
Traffic: UN

Deadlock Avoidance Mechanism: Dally
Routing: MIN
VC Usage: BASE
Misrouting Trigger: CGA
Local Threshold Percent: 100
Global Threshold Percent: 100
Threshold Min: 0
Congestion Misrouting Restriction: 0
Restriction Coef Percent: 150
Restriction Threshold: 5
Force Misrouting: 0

STATISTICS
Phits Sent: 5285744 (warmup: 5276464)
Flits Sent: 660718 (warmup: 659558)
Flits Received: 660772 (warmup: 657807)
Packets Received: 660772
Non Minimally Routed Packets Received: 0

InjectionMisrouted Packets Received: 0
 SrcGroupMisrouted Packets Received: 0
 IntGroupMisrouted Packets Received: 0
 Flits Misrouted Iter 0: 0 (0)
 Flits Local Misrouted Iter 0: 0 (0)
 Flits Global Misrouted Iter 0: 0 (0)
 Flits Global Mandatory Misrouted Iter 0: 0 (0)
 Flits MIN Routed Iter 0: 2435501 (99.7401)
 Flits Misrouted Iter 1: 0 (0)
 Flits Local Misrouted Iter 1: 0 (0)
 Flits Global Misrouted Iter 1: 0 (0)
 Flits Global Mandatory Misrouted Iter 1: 0 (0)
 Flits MIN Routed Iter 1: 6346 (0.259885)

Applied Load: 0.100109 phits/(node-cycle)
 Accepted Load: 0.100117 phits/(node-cycle)
 NonMinimal Accepted Load: 0 phits/(node-cycle)
 InjectionMisrouted Accepted Load: 0 phits/(node-cycle)
 SrcGroupMisrouted Accepted Load: 0 phits/(node-cycle)
 IntGroupMisrouted Accepted Load: 0 phits/(node-cycle)
 Cycles: 100000 (warmup: 50000)
 Total Hops: 1781203
 Local Hops: 1139994
 Global Hops: 641209
 Local Contention: 4.01691e+06
 Global Contention: 2.19405e+06
 Average Distance: 2.69564
 Average Local Distance: 1.72525
 Average Global Distance: 0.970394
 Average Local Contention: 6.07912
 Average Global Contention: 3.32044

Latency: 9.01533e+07 (warmup: 8.97607e+07)
 Packet latency: 9.01533e+07 (warmup: 8.97607e+07)
 Inj Latency: 3.13531e+06 (warmup: 3.11936e+06)
 Base Latency: 8.0807e+07
 Average Total Latency: 136.436
 Min Total Latency: 12
 Max Total Latency: 192
 Average Total Packet Latency: 136.436
 Average Inj Latency: 4.74492
 Average Latency: 131.691
 Average Base Latency: 122.292
 Max Hops: 3
 Max Local Hops: 2
 Max Global Hops: 1

PORT COUNTERS

Port0 Count: 164948 (9.99685 %)
 Port1 Count: 165409 (10.0248 %)
 Port2 Count: 165178 (10.0108 %)
 Port3 Count: 165237 (10.0144 %)
 Port4 Count: 162315 (9.83727 %)
 Port4 Contention Count: 1136229 (1.07597 %)
 Port5 Count: 162442 (9.84497 %)
 Port5 Contention Count: 1137086 (1.07679 %)
 Port6 Count: 162832 (9.86861 %)
 Port6 Contention Count: 1139827 (1.07938 %)
 Port7 Count: 163052 (9.88194 %)
 Port7 Contention Count: 1141338 (1.08081 %)
 Port8 Count: 163568 (9.91321 %)
 Port8 Contention Count: 1144990 (1.08427 %)
 Port9 Count: 162871 (9.87097 %)
 Port9 Contention Count: 1140084 (1.07963 %)
 Port10 Count: 162850 (9.8697 %)
 Port10 Contention Count: 1139940 (1.07949 %)
 Port11 Count: 160907 (9.75194 %)
 Port11 Contention Count: 1126325 (1.0666 %)
 Port12 Count: 160487 (9.72649 %)
 Port12 Contention Count: 1123394 (1.06382 %)
 Port13 Count: 159642 (9.67527 %)
 Port13 Contention Count: 1117496 (1.05823 %)
 Port14 Count: 160137 (9.70527 %)
 Port14 Contention Count: 1120943 (1.0615 %)

VC/PORT COUNTERS

LOCAL LINKS

VC0, Port 0 Count: 82648 (5.00897 %)
VC1, Port 0 Count: 79667 (4.8283 %)
VC0, Port 1 Count: 82324 (4.98933 %)
VC1, Port 1 Count: 80118 (4.85564 %)
VC0, Port 2 Count: 82412 (4.99467 %)
VC1, Port 2 Count: 80420 (4.87394 %)
VC0, Port 3 Count: 82525 (5.00152 %)
VC1, Port 3 Count: 80527 (4.88042 %)
VC0, Port 4 Count: 83538 (5.06291 %)
VC1, Port 4 Count: 80030 (4.8503 %)
VC0, Port 5 Count: 82561 (5.0037 %)
VC1, Port 5 Count: 80310 (4.86727 %)
VC0, Port 6 Count: 82649 (5.00903 %)
VC1, Port 6 Count: 80201 (4.86067 %)

GLOBAL LINKS

VC0, Port 7 Count: 321096 (19.4604 %)
VC0, Port 8 Count: 320554 (19.4275 %)
VC0, Port 9 Count: 319345 (19.3542 %)
VC0, Port 10 Count: 320141 (19.4025 %)

VC0:

Injection Queues Occupancy: 0.0482244 (0.0188377 %)
Local Queues Occupancy: 0.531164 (1.65989 %)
Global Queues Occupancy: 9.77552 (3.81856 %)

VC1:

Injection Queues Occupancy: 0.048147 (0.0188074 %)
Local Queues Occupancy: 0.527845 (1.64952 %)
Global Queues Occupancy: 0 (0 %)

VC2:

Injection Queues Occupancy: 0.0483373 (0.0188817 %)
Local Queues Occupancy: 0 (0 %)
Global Queues Occupancy: 0 (0 %)

Consumption Output Queues Occupancy: 0.330068 (1.03146 %)

Petitions Done: 2929024

Petitions Served: 2441847(83.3673 %)

Injection Petitions Done: 666090

Injection Petitions Served: 660718(99.1935 %)

Max Injections: 2640

Switch With Max Injections: 154

Min Injections: 2370

Switch With Min Injections: 212

Unbalance Quotient: 1.11392

Unfairness Quotient (stdev/avg): 0.0195393

GROUP 0

Group0 SW0 Flits: 2502

Group0 SW0 AvgTotalLatency: 129.531

Group0 SW0 Node0 MIN: 663

Group0 SW0 Node0 NON-MIN: 0

Group0 SW0 Node1 MIN: 602

Group0 SW0 Node1 NON-MIN: 0

Group0 SW0 Node2 MIN: 614

Group0 SW0 Node2 NON-MIN: 0

Group0 SW0 Node3 MIN: 623

Group0 SW0 Node3 NON-MIN: 0

Group0 SW1 Flits: 2392

Group0 SW1 AvgTotalLatency: 128.015

Group0 SW1 Node0 MIN: 636

Group0 SW1 Node0 NON-MIN: 0

Group0 SW1 Node1 MIN: 565

Group0 SW1 Node1 NON-MIN: 0

Group0 SW1 Node2 MIN: 580

Group0 SW1 Node2 NON-MIN: 0

Group0 SW1 Node3 MIN: 611

Group0 SW1 Node3 NON-MIN: 0

Group0 SW2 Flits: 2519

Group0 SW2 AvgTotalLatency: 128.722

Group0 SW2 Node0 MIN: 601

Group0 SW2 Node0 NON-MIN: 0

Group0 SW2 Node1 MIN: 660

Group0 SW2 Node1 NON-MIN: 0

Group0 SW2 Node2 MIN: 628

Group0 SW2 Node2 NON-MIN: 0
Group0 SW2 Node3 MIN: 630
Group0 SW2 Node3 NON-MIN: 0
Group0 SW3 Flits: 2569
Group0 SW3 AvgTotalLatency: 127.85
Group0 SW3 Node0 MIN: 654
Group0 SW3 Node0 NON-MIN: 0
Group0 SW3 Node1 MIN: 634
Group0 SW3 Node1 NON-MIN: 0
Group0 SW3 Node2 MIN: 656
Group0 SW3 Node2 NON-MIN: 0
Group0 SW3 Node3 MIN: 625
Group0 SW3 Node3 NON-MIN: 0
Group0 SW4 Flits: 2442
Group0 SW4 AvgTotalLatency: 127.992
Group0 SW4 Node0 MIN: 613
Group0 SW4 Node0 NON-MIN: 0
Group0 SW4 Node1 MIN: 626
Group0 SW4 Node1 NON-MIN: 0
Group0 SW4 Node2 MIN: 602
Group0 SW4 Node2 NON-MIN: 0
Group0 SW4 Node3 MIN: 601
Group0 SW4 Node3 NON-MIN: 0
Group0 SW5 Flits: 2557
Group0 SW5 AvgTotalLatency: 128.184
Group0 SW5 Node0 MIN: 645
Group0 SW5 Node0 NON-MIN: 0
Group0 SW5 Node1 MIN: 643
Group0 SW5 Node1 NON-MIN: 0
Group0 SW5 Node2 MIN: 651
Group0 SW5 Node2 NON-MIN: 0
Group0 SW5 Node3 MIN: 618
Group0 SW5 Node3 NON-MIN: 0
Group0 SW6 Flits: 2432
Group0 SW6 AvgTotalLatency: 128.764
Group0 SW6 Node0 MIN: 592
Group0 SW6 Node0 NON-MIN: 0
Group0 SW6 Node1 MIN: 612
Group0 SW6 Node1 NON-MIN: 0
Group0 SW6 Node2 MIN: 621
Group0 SW6 Node2 NON-MIN: 0
Group0 SW6 Node3 MIN: 607
Group0 SW6 Node3 NON-MIN: 0
Group0 SW7 Flits: 2462
Group0 SW7 AvgTotalLatency: 127.719
Group0 SW7 Node0 MIN: 597
Group0 SW7 Node0 NON-MIN: 0
Group0 SW7 Node1 MIN: 625
Group0 SW7 Node1 NON-MIN: 0
Group0 SW7 Node2 MIN: 621
Group0 SW7 Node2 NON-MIN: 0
Group0 SW7 Node3 MIN: 619
Group0 SW7 Node3 NON-MIN: 0

Anexo C

Script test de sistema: Throughput y Latencia

```
#!/usr/bin/python

import sys
import math

## Returns the average of the list passed as argument ##
def getAverage(list1):

    n=len(list1)
    average=sum(list1)/n
    return average

## Returns the Standard deviation of the list ##
def getStandardDeviation(list1, average):

    n=len(list1)
    suma=0

    for i in range(0, n):
        temp=pow((list1[i]-average),2)
        suma=suma+temp

    variance=(1/float((n-1)))*suma
    standardDeviation=math.sqrt(variance)

    return standardDeviation

if(len(sys.argv)<3 or (len(sys.argv))>3):
    print("ERROR! --> You must enter the name of the output_
        execution_file_and the multiplicator!")
    sys.exit(0)

##### Latencies result of different executions #####

lat_min_un_10=[136.436,136.494,136.471,136.413,136.433]
lat_min_un_40=[146.008,146.023,145.968,146.034,145.968]
lat_min_un_100=[1053.87,1055.3,1054.94,1052.99,1051.44]
```

```
lat_pb_adv_10=[255.306,255.337,255.272,255.339,255.287]
lat_pb_adv_40=[370.536,372.089,371.481,372.817,372.569]
lat_pb_adv_100=[1031.9,1032.02,1031.8,1032.1,1032.32]

lat_pb_un_10=[146.37,146.303,146.307,146.36,146.298]
lat_pb_un_40=[169.912,169.918,169.824,169.935,169.88]
lat_pb_un_100=[1337.6,1339.8,1338.94,1337.91,1336.56]

lat_val_adv_10=[265.576,265.572,265.645,265.559,265.639]
lat_val_adv_40=[387.495,388.663,385.75,396.475,388.927]
lat_val_adv_100=[1084.45,1083.05,1083.83,1084.69,1082.42]

#### Throughput result of different executions ####

thr_min_un_10=[0.100117,0.0999462,0.0998797,0.0999786,0.0999389]
thr_min_un_40=[0.400066,0.400228,0.399989,0.400261,0.399732]
thr_min_un_100=[0.729039,0.72789,0.72742,0.728484,0.728293]

thr_pb_adv_10=[0.100102,0.099963,0.0998305,0.0999702,0.100013]
thr_pb_adv_40=[0.399936,0.399985,0.399761,0.400192,0.39937]
thr_pb_adv_100=[0.41931,0.419335,0.418573,0.41915,0.419136]

thr_pb_un_10=[0.10012,0.0999364,0.0998342,0.099937,0.0999588]
thr_pb_un_40=[0.400019,0.400331,0.399974,0.400532,0.39988]
thr_pb_un_100=[0.673067,0.672443,0.672338,0.673573,0.673284]

thr_val_adv_10=[0.100118,0.0999523,0.0998011,0.0999473,0.099993]
thr_val_adv_40=[0.39959,0.399818,0.399974,0.39923,0.398947]
thr_val_adv_100=[0.395699,0.396097,0.396617,0.395186,0.394715]

# open the output file (result of an execution) in order to read it #
infile = open(sys.argv[1], 'r')

# Read the configuration in the output file in order to know the
  correct latencies #

flag=0 # This flag is a lock in order to get the correct value of
  throughput (first line "Accepted load: " in the output file) #
for line in infile:

    if("Average_Total_Latency:" in line):
        line=line.split(":")
        latency=line[1]

    if("Accepted_Load:" in line and flag==0):
        line=line.split("\n")
        throughput=line[2]
        flag=1

    if("Traffic:UN" in line): #UNIFORM TRAFFIC
        traffic="uniform"
```

```

if("Traffic:␣ADV_RANDOM_NODE" in line): #UNIFORM TRAFFIC
    traffic="adverse␣random"

if("Routing:␣MIN" in line):# MININAL ROUTING
    routing="minimal"

if("Routing:␣OBL" in line):# MININAL ROUTING
    routing="OBLvaliant"

if("Piggybacking␣Coef:" in line):# Piggybacking with
    ADAPTATIVE routing
    routing="Piggybacking␣ADP"

if("Injection␣Probability:␣10" in line):# prob=10
    prob=10

if("Injection␣Probability:␣40" in line):# prob=40
    prob=40

if("Injection␣Probability:␣100" in line):# prob=100
    prob=100

# We close the file #
infile.close()

print('The␣traffic␣is␣'+traffic)
print('The␣routing␣is␣'+routing)
print('The␣probability␣is␣'+str(prob))

##### latencyRef is the average of the latency result in n
        executions #####

if(traffic=="uniform"):# UNIFORM TRAFFIC
    if(routing=="minimal"):
        if(prob==10):
            latencyRef=getAverage(lat_min_un_10)
            standardDeviationLat=getStandardDeviation(
                lat_min_un_10,latencyRef)

            throughputRef=getAverage(thr_min_un_10)
            standardDeviationThr=getStandardDeviation(
                thr_min_un_10,throughputRef)
        elif(prob==40):
            latencyRef=getAverage(lat_min_un_40)
            standardDeviationLat=getStandardDeviation(
                lat_min_un_40,latencyRef)

            throughputRef=getAverage(thr_min_un_40)
            standardDeviationThr=getStandardDeviation(
                thr_min_un_40,throughputRef)
        elif(prob==100):
            latencyRef=getAverage(lat_min_un_100)
            standardDeviationLat=getStandardDeviation(

```

```
        lat_min_un_100 , latencyRef)

        throughputRef=getAverage(thr_min_un_100)
        standardDeviationThr=getStandardDeviation(
            thr_min_un_100 , throughputRef)
    else:
        latencyRef=-1

elif(routing=="Piggybacking_ADP"): #routing adaptative with
    PIGGYBACKING
    if(prob==10):
        latencyRef=getAverage(lat_pb_un_10)
        standardDeviationLat=getStandardDeviation(
            lat_pb_un_10 , latencyRef)

        throughputRef=getAverage(thr_pb_un_10)
        standardDeviationThr=getStandardDeviation(
            thr_pb_un_10 , throughputRef)
    elif(prob==40):
        latencyRef=getAverage(lat_pb_un_40)
        standardDeviationLat=getStandardDeviation(
            lat_pb_un_40 , latencyRef)

        throughputRef=getAverage(thr_pb_un_40)
        standardDeviationThr=getStandardDeviation(
            thr_pb_un_40 , throughputRef)
    elif(prob==100):
        latencyRef=getAverage(lat_pb_un_100)
        standardDeviationLat=getStandardDeviation(
            lat_pb_un_100 , latencyRef)

        throughputRef=getAverage(thr_pb_un_100)
        standardDeviationThr=getStandardDeviation(
            thr_pb_un_100 , throughputRef)
    else:
        latencyRef=-1
else:
    latencyRef=-1

elif(traffic=="adverse_random"): #ADV TRAFFIC
    if(routing=="Piggybacking_ADP"): #adaptative with PIGGYBACKING
        if(prob==10):
            latencyRef=getAverage(lat_pb_adv_10)
            standardDeviationLat=getStandardDeviation(
                lat_pb_adv_10 , latencyRef)

            throughputRef=getAverage(thr_pb_adv_10)
            standardDeviationThr=getStandardDeviation(
                thr_pb_adv_10 , throughputRef)
        elif(prob==40):
            latencyRef=getAverage(lat_pb_adv_40)
```

```

        standardDeviationLat=getStandardDeviation(
            lat_pb_adv_40,latencyRef)

        throughputRef=getAverage(thr_pb_adv_40)
        standardDeviationThr=getStandardDeviation(
            thr_pb_adv_40,throughputRef)
    elif(prob==100):
        latencyRef=getAverage(lat_pb_adv_100)
        standardDeviationLat=getStandardDeviation(
            lat_pb_adv_100,latencyRef)

        throughputRef=getAverage(thr_pb_adv_100)
        standardDeviationThr=getStandardDeviation(
            thr_pb_adv_100,throughputRef)
    else:
        latencyRef=-1

elif(routing=="OBLvaliant"): #VALIANT ROUTING
    if(prob==10):
        latencyRef=getAverage(lat_val_adv_10)
        standardDeviationLat=getStandardDeviation(
            lat_val_adv_10,latencyRef)

        throughputRef=getAverage(thr_val_adv_10)
        standardDeviationThr=getStandardDeviation(
            thr_val_adv_10,throughputRef)
    elif(prob==40):
        latencyRef=getAverage(lat_val_adv_40)
        standardDeviationLat=getStandardDeviation(
            lat_val_adv_40,latencyRef)

        throughputRef=getAverage(thr_val_adv_40)
        standardDeviationThr=getStandardDeviation(
            thr_val_adv_40,throughputRef)
    elif(prob==100):
        latencyRef=getAverage(lat_val_adv_100)
        standardDeviationLat=getStandardDeviation(
            lat_val_adv_100,latencyRef)

        throughputRef=getAverage(thr_val_adv_100)
        standardDeviationThr=getStandardDeviation(
            thr_val_adv_100,throughputRef)
    else:
        latencyRef=-1
else:
    latencyRef=-1

else:
    latencyRef=-1

print("Latency␣reference␣(average␣latency):␣"+str(latencyRef)+"␣---␣
      Output␣latency:␣"+latency)
print("Throughput␣reference␣(average␣throughput):␣"+str(throughputRef)
      +"␣---␣Output␣throughput:␣"+throughput)

```

```
print('The_Latency_Standard_Deviation_is_'+str(standardDeviationLat))
print('The_Throughput_Standard_Deviation_is_'+str(standardDeviationThr
))

if(latencyRef==-1):
    print("ERROR-->_Parameters_of_the_network_are_not_appropriate_
        for_the_test!")
else:

    multiplicator=float(sys.argv[2])

    latencyMax=latencyRef+multiplicator*standardDeviationLat
    latencyMin=latencyRef-multiplicator*standardDeviationLat

    throughputMax=throughputRef+multiplicator*standardDeviationThr
    throughputMin=throughputRef-multiplicator*standardDeviationThr

    print("MAX_Latency_permitted:_"+str(latencyMax))
    print("MIN_Latency_permitted:_"+str(latencyMin))

    print("MAX_Throughput_permitted:_"+str(throughputMax))
    print("MIN_Throughput_permitted:_"+str(throughputMin))

    flag=0
    if(float(latency)>latencyMax or float(latency)<latencyMin):
        flag=1
        print("TEST_Failed-->_Average_Latency_is_not_what_
            expected.")
    else:
        print("Latency_is_correct!")

    if(float(throughput)>throughputMax or float(throughput)<
        throughputMin):
        flag=1
        print("TEST_Failed-->_Average_Throughput_is_not_what_
            expected.")
    else:
        print("Throughput_is_correct!")

    if(flag==0):
        print("TEST_PASSED-->_Latency_and_Throughput_are_
            correct._Well_done!:_)")
```