Capítulo 6

Simulación de esquemas de encolamiento

En este capítulo se presenta el proceso realizado para simular diferentes esquemas de encolamiento (FIFO, PQ y LLQ) de manera básica con el fin de ofrecer QoS en una red. También se presentan gráficas que permiten realizar el análisis posterior de los datos obtenidos en la simulación.

6.1. Entorno de desarrollo utilizado para la implementación

Como presenta Varga (2011), OMNeT++ es una herramienta que se puede ejecutar en diferentes sistemas operativos. Ha sido probada en los siguientes ambientes:

- Windows 7, Vista y XP;
- Mac OS Leopard (10.5) y Snow Leopard (10.6);
- Ubuntu 10.04 LTS y 11.04;
- Fedora Core 15 y Core 16;
- Red Hat Enterprise Linux Desktop Workstation 5.5; y
- Open SUSE 11.4.

OMNeT++ se ha probado en arquitecturas Intel de 32 y 64 bits, aunque *Red Hat* es el único sistema operativo en el que se ha probado con la arquitectura Intel-32 bits.

Para la implementación realizada en este capítulo se probó el simulador OMNeT++ 4.1 en Mac OS X Lion (10.7), con resultados exitosos. Para su implementación fue necesario instalar, *XCode 4.3.2* y *Java Runtime*, el cual no viene por defecto en la última versión de Mac OS X; en Windows 7, se adicionó el JDK en la carpeta pertinente del simulador, para su correcto funcionamiento.

Las simulaciones de OMNeT++, pueden correr prácticamente en cualquier entorno Unix que contenga un compilador C++ actualizado, ya que algunas características (i.e., *Tkenv*, *parallel simulation*, *XML support*, etc.) necesitan librerías externas (e.g., *Tcl/Tk, MPI, LibXML* y *Expat*)

El entorno de desarrollo integrado de OMNeT 4.x se basa en la plataforma *Eclipse* (Varga, 2011); utiliza además nuevos editores, asistentes y funcionalidades adicionales. Para lograr la plataforma visual y facilitar un entorno agradable para la herramienta, OMNeT++ integra nuevas funcionalidades que permiten crear y configurar modelos; estos archivos utilizan extensiones *.ned y .ini*. Incluye también otro tipo de funcionalidades para realizar ejecuciones por lotes y analizar resultados. *Eclipse*, por su parte, permite editar el funcionamiento de las partes adjuntas del código C++, para el correcto funcionamiento de la simulación.

Los archivos *.ned*, permiten al usuario generar módulos simples con su propio lenguaje (*Ned*), los cuales son componentes activos de los modelos (i.e., *routers*, generadores de tráfico, *switches*, etc.), y describir la estructura del modelo de simulación que se desea

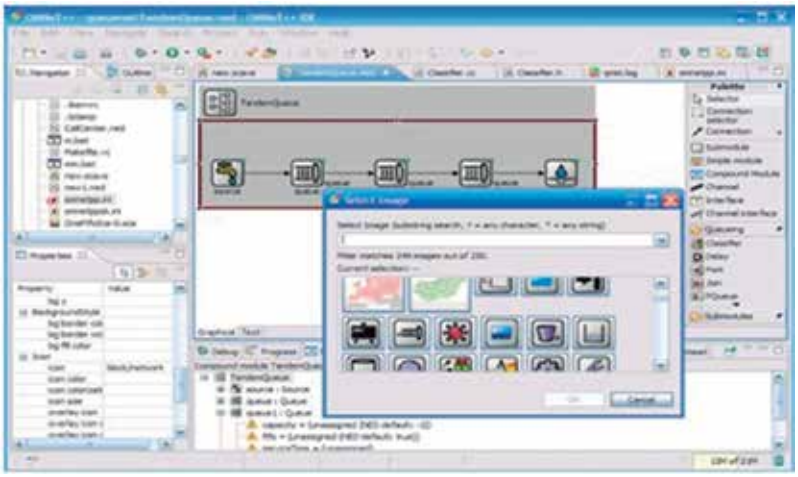


Figura 63. Presentación archivo *.ned* (OMNeT++ Community, 2012)

implementar. A su vez un archivo *.ned* puede utilizar módulos desarrollados en otros archivos *.ned* para generar una red con diferentes elementos. En la Figura 63 se puede ver una red generada por múltiples módulos, en el que la red es, a su vez, un módulo.

6.2. Escenario de simulación

En términos de calidad de servicio (QoS), son muchos los obstáculos que hay que superar para lograr entregar los datos dentro de los parámetros ideales para las aplicaciones sensibles al retardo. Para evaluar el tipo de esquema en el cuál el funcionamiento es óptimo, es necesario diseñar una red que contenga los elementos primordiales para la evaluación de cada uno de los componentes que afectan la calidad de servicio.

Teniendo en cuenta lo anterior, cabe aclarar que los componentes que influyen en el retardo de extremo a extremo, como indica Tanenbaum (2003) son muchos, incluyendo el retardo del algoritmo de codificación, el tiempo de empaquetado, los tiempos de propagación (despreciable excepto cuando la información viaja grandes distancias), el tiempo de transmisión, los tiempos de espera en las colas de red (lo cual depende de la cantidad de paquetes que cada cola maneje), el tiempo de descompresión, etc.

Por ende, para lograr efectuar una simulación que sea lo más ajustada posible a la realidad, es necesario tener un escenario donde se encuentren todos estos componentes básicos y se pueda aislar el componente con el cual se desea trabajar. Obligatoriamente tienen que existir cuatro módulos principales: un equipo generador de tráfico; un módulo que direccione y maneje los paquetes, simulando el paso por otro u otros nodos de la red; un elemento que consuma los mensajes para terminar su recorrido y; los enlaces que se encargan de transportar los paquetes.

Para el escenario de simulación elegido, se cuenta con los cuatro componentes primordiales explicados. El primero, un generador de tráfico con la posibilidad

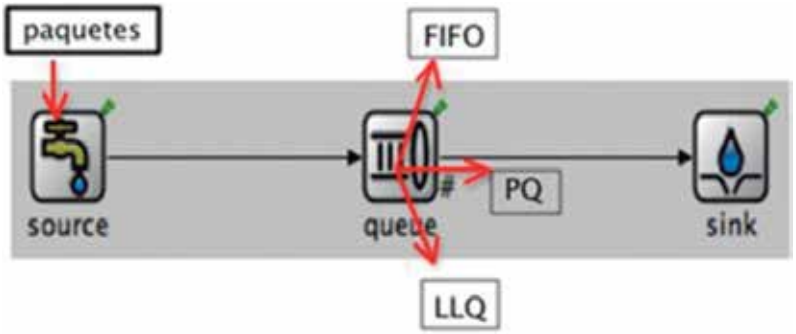


Figura 64. Escenario de simulación adaptado de simulador OMNeT++

de etiquetar los paquetes creados y emitirlos por el canal. Este elemento tiene configuradas, a su vez, las características necesarias para congestionar el siguiente componente, una cola, la que, en este caso en particular, simulará los elementos por los que tienen que pasar a través de la red, donde existe la posibilidad de congestión, seguido por un consumidor, quien se encargará de representar el elemento al cual llega el mensaje. La implementación realizada se presenta en la Figura 64.

Cada uno de estos elementos representa, como se ha mencionado, la estructura para que la información viaje, por lo que es importante explicar, cada uno de ellos.

6.2.1. Paquete

OMNeT++ es un simulador orientado al paso de mensajes para la comunicación de sus módulos. Se utilizan dos tipos de objetos para la transmisión de mensajes, los cuales permiten comunicar módulos simples y compuestos, los *cMessages* y los *cPackets*.

cPacket es una subclase de *cMessage* (OMNeT++ Community, 2012). La primera es utilizada para simulaciones que necesitan paquetes de red (i.e., tramas, datagramas, etc.); la segunda, para el resto de paso de mensajes que no son específicos.

En el contexto de la simulación a presentar, se seleccionaron, para el paso de información, mensajes de tipo *cPacket*, ya que presentan un constructor similar al de *cMessage*, por ser una subclase, pero aceptan el argumento de tamaño del mensaje (*bit length)*; este archivo es guardado en bits, pero puede ser almacenado, a su vez, en bytes. En caso que el campo guardado no sea múltiplo de ocho, automáticamente lo completará con una función techo para agregar los bits que hagan falta.

Para la creación del paquete utilizado en este trabajo, fue necesario generar, en lenguaje *Ned*, un paquete que tiene un entero (*int*) llamado *priority*. Una vez se compiló el proyecto, gracias al paquete de extensión *.msg*, se generaron otras dos clases, una representando la clase desarrollada en C++ (*MyPacket_m.cc*) y otra en representación de la cabecera (*MyPacket_m.h*).

En la clase *MyPacket_m.cc* quedaron plasmados todos los métodos que contiene la clase *cPacket* (*get* y *set*) incluyendo los de la variable *priority* adicionada por los desarrolladores del proyecto.

Los paquetes, aunque poseen la propiedad de definir un tamaño variable, se tomarán con un tamaño constante, ya que, para los contextos en los que se evaluará la práctica, no se tendrá en cuenta ese tamaño.

6.2.2. Generador de tráfico

La Figura 65 muestra la imagen con la que se identificará el generador en la red. Se plantea el generador de tráfico como la representación de un nodo en la red que espera transmitir información, lo que puede ser visto desde diferentes perspectivas. Por esta razón, no se especifica como un nodo en particular, ya que puede estar representando, tanto a un equipo que genera paquetes, como a la transmisión de paquetes de varios equipos que tienen llegada a la cola a presentar.

Figura 65. Generador de tráfico (OMNeT++ Community, 2011a)

El generador de tráfico tiene un puerto de salida, por donde encuentra la posibilidad de colocar los paquetes en el canal. A su vez, posee una variable *volatile double interArrival Time*, expresada en unidades de segundos, que representa el tiempo de generación de los paquetes. Como la idea se centra en iniciar la simulación con una cola congestionada para analizar el comportamiento de los algoritmos de encolamiento, se están enviando todos los paquetes con un tiempo de llegada a la cola de cero (0) segundos.

Las clases de OMNeT++ implementadas por el generador desarrollado son las siguientes:

Initialize(). Encargada de asignar el tiempo de inicio a la variable *startTime* y auto-enviar un mensaje al módulo, para que este sea manejado por el método *handleMessage()*; el método que utiliza para realizar lo explicado tiene como nombre *scheduleAt* y recibe como parámetros, el tiempo de simulación dado por la variable *startTime* y el mensaje a enviar.

textbfhandleMessage(). Este método recibe como parámetro un *cMessage*, el que, para este caso, proviene del método *Initialize()*. Es necesario aclarar que para la práctica presentada, se recibe un paquete de tipo *MyPacket*, desarrollado por los autores del proyecto, por lo que es necesario hacer un *cast* para lograr editar el paquete y sus prioridades.

6.2.3. Colas

Para entender la importancia de la implementación de las colas en las redes de datos es necesario primero entender que las aplicaciones son de gran importancia para

la implementación de etiquetas en los paquetes, ya que de acuerdo con la etiqueta, se define el ancho de banda que necesitan y la prioridad que puedan requerir. Para entender esto más a fondo, se puede reunir en dos grupos las diferentes aplicaciones: sensibles y no sensibles al retardo.

Las aplicaciones sensibles al retardo se centran en los diferentes programas ejecutados en la red, que requieren llegar con el mínimo retraso posible, desde su fuente hasta su destino final. Generalmente, este tipo de aplicaciones suelen representarse con comunicaciones de VoIP y video, pero no son necesariamente estos los ejemplos que pueden definir este tipo de información de una forma totalmente objetiva.

Por lo anterior, se debe tener en cuenta el desarrollo exponencial de las aplicaciones que generan las grandes compañías y las aplicaciones de salud, las cuales tienden a tener el comportamiento de una aplicación sensible al retardo, ya que en ellas se encuentra información valiosa que debe llegar con tiempos efectivos a su destino final. En ambientes industriales, las medidas de ciertos sensores son críticas, como medidas de químicos, operaciones de precisión, detección de bombas, entre otros aspectos que afectan, incluso, la vida de las personas involucradas.

Por otro lado, existen las aplicaciones que no son sensibles al retardo, aquellas cuyo momento de llegada no es crítico, como es el caso del correo electrónico, los mensajes de texto, los chats, las actualizaciones en páginas Web, la descarga de archivos y los elementos de las redes sociales, entre otros.

El tráfico de estas aplicaciones podría en determinado momento esperar por el uso del canal; pero el hecho de que espere no indica que se quede sin utilizarlo, ya que en determinado momento –y según las características del algoritmo de encolamiento–, este tráfico debe utilizar el canal.

Otra característica de la mayoría de aplicaciones no sensibles al retardo (Tanenbaum, 2003) es el alto nivel de confiabilidad que deben tener, ya que, aunque son aplicaciones que permiten tener tiempo de retardo extenso, es necesario que sus paquetes lleguen completos y sin errores, lo que implica, hacer cálculos de suma, para comprobar la llegada correcta de todos los bits y retransmisiones, en caso de que los paquetes lleguen con errores.

Para efectos del esquema de simulación a realizar en este capítulo, se definieron dos tipos de prioridad de tráfico, alta y baja. Prioridad alta para tráfico sensible a retardo, como VoIP o video, y prioridad baja para tráfico transaccional, tipo ftp o correo electrónico.

En la comparación de los comportamientos de las colas, además de definir las prioridades, es necesario seleccionar las colas a evaluar; para este caso, se valoraron tres tipos de cola, para analizar, en efectos prácticos, como OMNeT++ permite apreciar la diferencia entre los comportamientos de estas colas y lo eficientes que ellas pueden ser. Las colas elegidas para evaluar la diferencia en su comportamiento son FIFO, PQ y LLQ.

FIFO (*First In First Out*) es un tipo de encolamiento que se utiliza generalmente por defecto. Es implementado también cuando no se maneja tráfico sensible al retardo, ya que al no tener ningún algoritmo de encolamiento complejo para el almacenamiento y la extracción en la cola, minimiza el tiempo de procesamiento. Su comportamiento, como se muestra en la Figura 66**,** define que el primer paquete en entrar a la cola es el primero en salir y así sucesivamente.

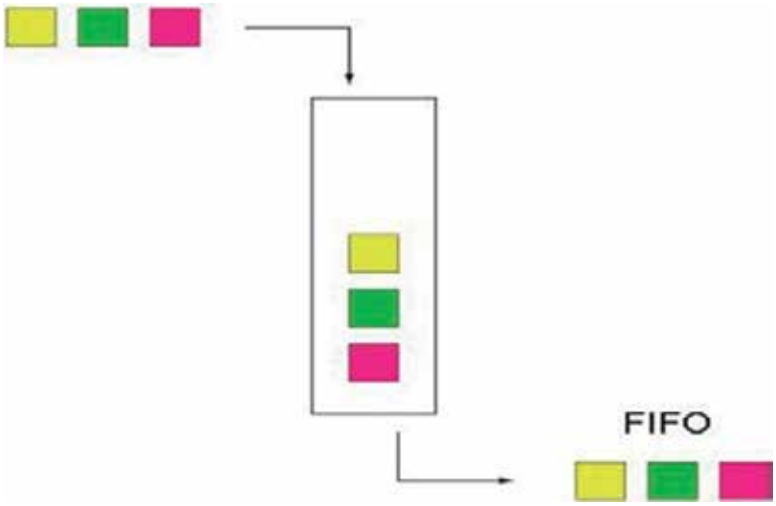


Figura 66. Encolamiento FIFO (Frabs, 2008)

PQ (*Priority Queuing*) es un tipo de encolamiento orientado a la prioridad, que organiza cada paquete en proceso de inserción a la cola, según la prioridad que cada paquete tenga, como se presenta en la Figura 67.

Priority Queuing conserva varias colas, las cuales comprenden, a su vez, diferentes tipos de prioridades. Cada cola es una cola FIFO en sí misma, y guarda los elementos según su etiqueta lo defina. Su implementación es ideal para atender aplicaciones de alto nivel de sensibilidad, cuando el tráfico que ellas generan no es demasiado. Es necesario aclarar que la cola no se debe implementar cuando el tráfico de alta prioridad generado es constante, porque el tráfico de baja prioridad se demoraría en

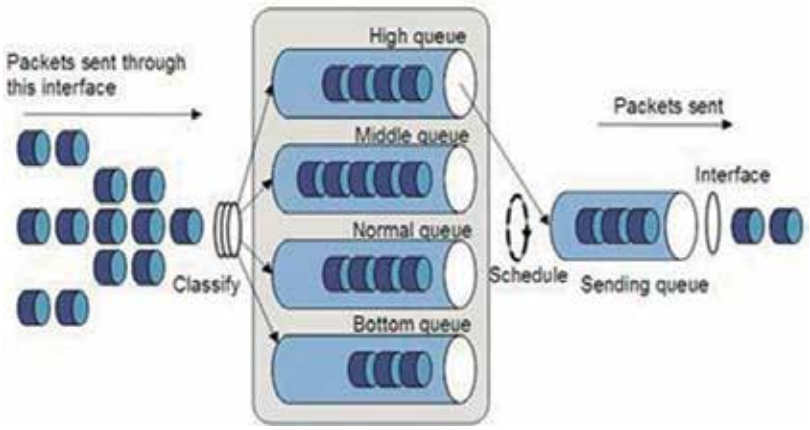


Figura 67. Encolamiento PQ (Moorey, 2008)

ser atendido o incluso, en el peor de los casos, podría no llegar a ser despachado, ya que el algoritmo está diseñado para atender primero los paquetes de mayor prioridad, siempre.

LLQ (*Low Latency Queuing*) es un algoritmo que presenta aspectos importantes en el manejo de tráfico, ya que le da mayor prioridad a las aplicaciones sensibles al retardo, sin olvidar la cola de baja prioridad, como se muestra en la Figura 68. Para lograrlo, conmuta entre las diferentes colas, asignando más ancho de banda a las colas con mayor prioridad, logrando que estas lleguen en tiempo ideal a su destino final.

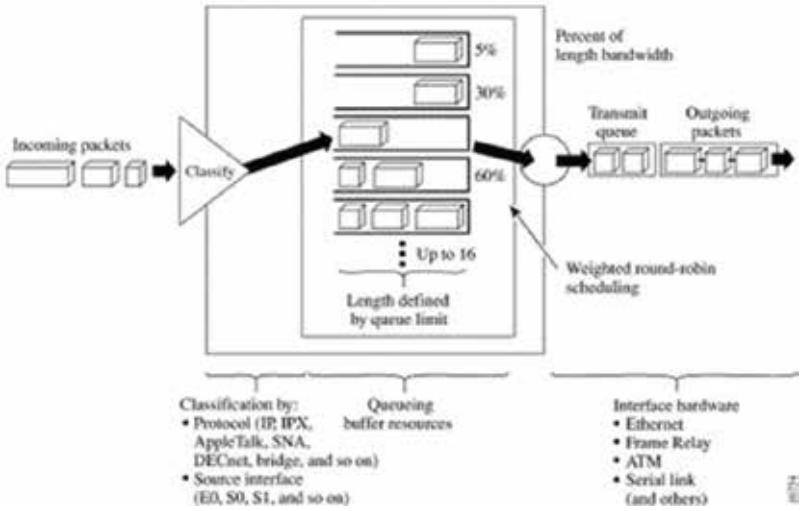


Figura 68. Encolamiento LLQ (http://www.ccietalk.com/)

Según las colas utilizadas, se tendrán diferentes tiempos de llegada; se espera analizar qué tipo de encolamiento presenta mejor desempeño frente a los demás.

La cola, al igual que los otros módulos desarrollados, cuenta con tres métodos principales que permite utilizar OMNeT++ y con una variable *Volatile Double Service Time* que permite medir los tiempos de llegada de los mensajes a este módulo.

Initialize(). Este método se encarga de asignar el nombre a la cola y generar un paquete *end-Service*, para su uso posterior.

handleMessage().Recibe el mensaje, lee la prioridad según la información obtenida y la cola implementada, guarda el paquete en la ubicación que le corresponda y, en el momento indicado, saca los paquetes de la cola para su posterior envío.

startService(). Define el tiempo en el cual llega el mensaje.

6.2.4. Consumidor

El consumidor es un módulo desarrollado para simular el usuario final de la aplicación. En este punto, el paquete recibido será eliminado, permitiendo que el módulo reciba otros paquetes.

Ya que este módulo no debe hacer ningún tipo de algoritmo de procesamiento alto, solamente se utilizará un método para manejar todo el tráfico, para que reciba y haga

el procesamiento indicado, *handleMessage()*, el cual como se mencionó, se encargará de tomar el mensaje y eliminarlo. Es un módulo que queda diseñado para poder ser editado y posteriormente implementado, para la recepción y manejo del diverso tráfico que puede recibir.

6.2.5. Canal

El canal que se implementó en la simulación es el predeterminado por OMNeT++, el cual se asume como infinito, por lo que no influirá en el tiempo manejado en la implementación, lo que permite asegurar que el tiempo expuesto es total y únicamente el resultado de lo sucedido en la cola.

6.3. Resultados experimentales

6.3.1. Datos obtenidos en las simulaciones

A continuación se presenta la interfaz que utiliza OMNeT++ para evidenciar la ocurrencia de cada evento en el simulador. Gracias a este mecanismo de simulación, se puede realizar un análisis comparativo de la situación ocurrida en la simulación. En la Figura 69, se puede apreciar, en color azul, la descripción de los eventos arrojados por OMNeT++, y en verde, la descripción de los eventos realizada por los programadores de la simulación.

En la Figura 69 es posible notar como al inicio de la simulación, se empiezan a describir los eventos por un número consecutivo, el tiempo de simulación, el nombre de la red y el módulo que genera el evento separados por un punto, la información del mensaje y el ID del mensaje.

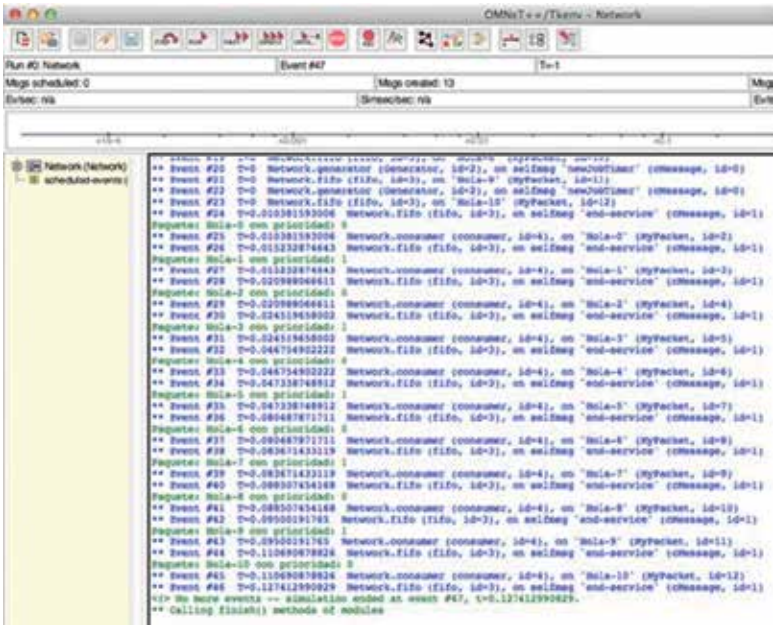


Figura 69. Resultados de la simulación FIFO

Adicional a los eventos, se muestran las prioridades de los paquetes en color verde, para facilitar la visualización del orden de llegada de los paquetes.

Para las siguientes dos simulaciones se presenta la Tabla 5, como resumen, detallando los datos más relevantes de la simulación para el desarrollo del estudio realizado. La Tabla 5 presenta los valores principales para los tres tipos de cola evaluados en las simulaciones, FIFO, PQ y LLQ. De cada tipo de cola, se evaluó el tiempo de salida del primer paquete de alta prioridad y el último paquete de alta prioridad, para poder comprobar, en comparación con las otras colas, si tiene mucho retardo en salir el primer y el último paquete. Por otra parte, se espera poder comprobar si los paquetes de baja prioridad están saliendo en tiempos pertinentes y si está quedando por tiempo indefinido en la cola, lo cual implica la necesidad de evaluar el tiempo del primero y el último de los paquetes de baja prioridad.

Tabla 5. Resumen de los resultados de simulación

COLAS	Salida primer paquete con prioridad alta	Salida último paquete con prioridad alta	Salida primer paquete con prioridad baja	Salida último paquete con prioridad baja
FIFO	0.01523	0.09500	0.01038	0.11069
PQ	0.01038	0.04675	0.04733	0.11069
LLQ	0.01038	0.04733	0.02451	0.11069

De lo anterior, se puede asumir la necesidad de implementación de colas para el manejo de las redes teniendo en cuenta los siguientes enunciados:

La cola que permitió a la aplicación con más prioridad finalizar primero, fue *Priority Queuing*.

La cola en la que ambas aplicaciones llegaron en tiempos similares fue la cola *FIFO*.

La cola que presentó más eficiencia respecto a la llegada de los paquetes tanto prioritarios como no prioritarios fue la *LLQ*.

La cola de prioridades (*Priority queuing*), no permitió brindar un servicio óptimo, ya que deja pasar solo paquetes de alta prioridad antes que los de baja prioridad, lo que deja los paquetes de baja prioridad, esperando durante mucho tiempo. Como se puede apreciar en la Tabla 5, el primer paquete de baja prioridad en FIFO, es retirado de la cola para su envío a los 0.01038 s, en LLQ a los 0.02451 s, y en PQ 0.04734s; este último presenta tiempo ineficiente para los paquetes de baja prioridad, con aproximadamente el doble de la retirada de paquetes de baja prioridad de LLQ y cuatro veces la de FIFO.

El tiempo de procesamiento que toman los módulos en las tres diferentes colas es despreciable, por lo que es importante recalcar, que siendo esta una característica a favor de la cola FIFO, se puede objetar que es la menos ideal para implementar.

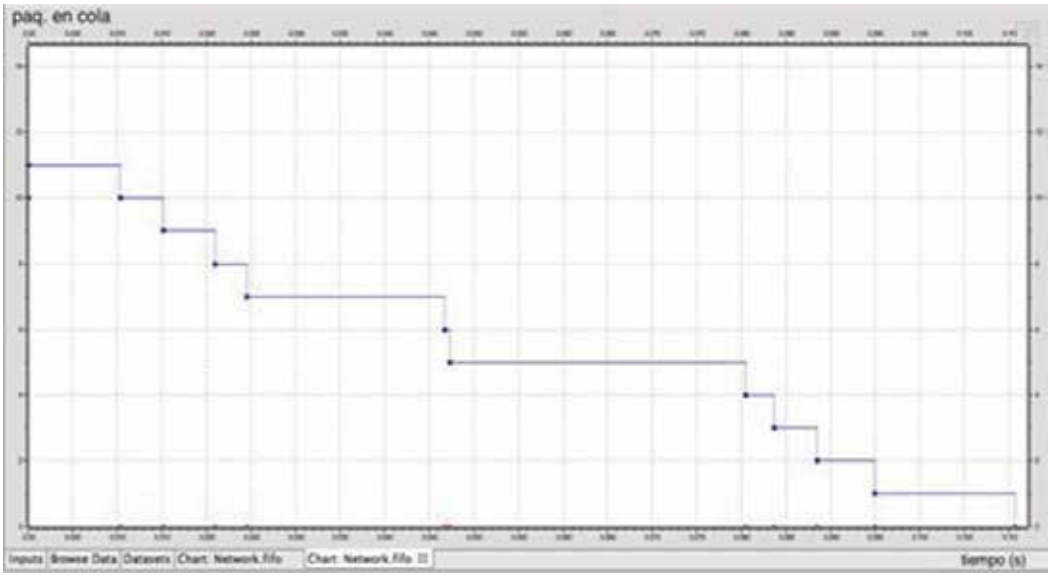
En la cola LLQ el tiempo de llegada del último paquete es 0.047338 s, y en PQ es 0.046754 s, la diferencia es relativamente pequeña, teniendo en cuenta que la cola LLQ, permitió dar paso a otro tipo de paquetes.

6.4. Gráficas del comportamiento de las colas

6.4.1. Gráfica FIFO

Además de lo presentado, OMNeT++ tiene una forma fácil para entender el comportamiento de los eventos y los paquetes. En la Figura 70 se pueden apreciar los puntos rojos inferiores, que resaltan los tiempos en que los paquetes son retirados de la cola, y la línea azul, que muestra el comportamiento de la cola; se puede ver claramente que la cola se encuentra congestionada al inicio de la simulación, lo que ocurre porque se establece que todos los paquetes llegan en el segundo cero (0), para generar una congestión; se puede ver cómo se va liberando la cola hasta quedar totalmente vacía.

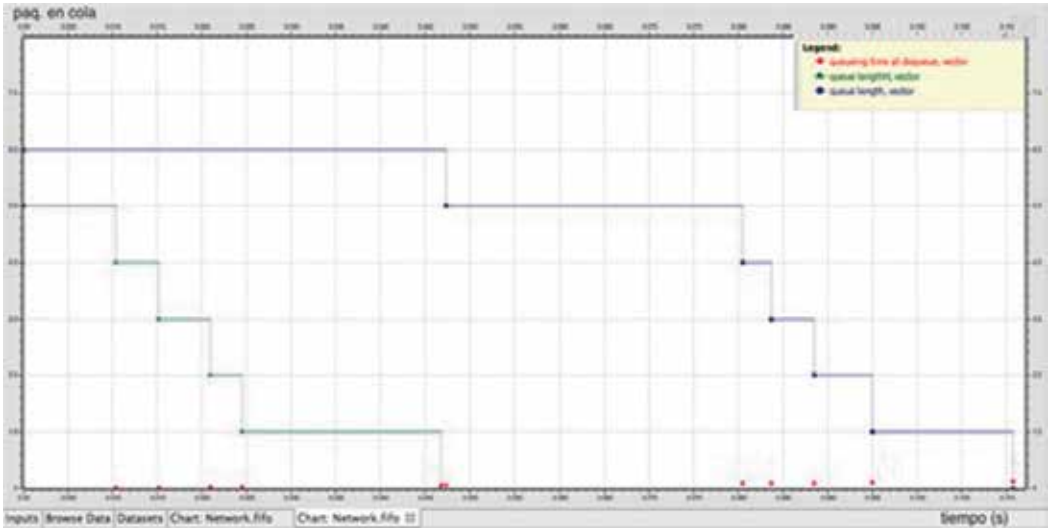
Figura 70. Comportamiento de la cola FIFO



6.4.2. Gráfica PQ

En el encolamiento PQ se puede apreciar que la cola de baja prioridad permanece cogestionada, hasta que la cola de alta prioridad queda completamente vacía; se aprecia como el primer paquete de baja prioridad, tiene mucho retardo al salir (ver Figura 71).

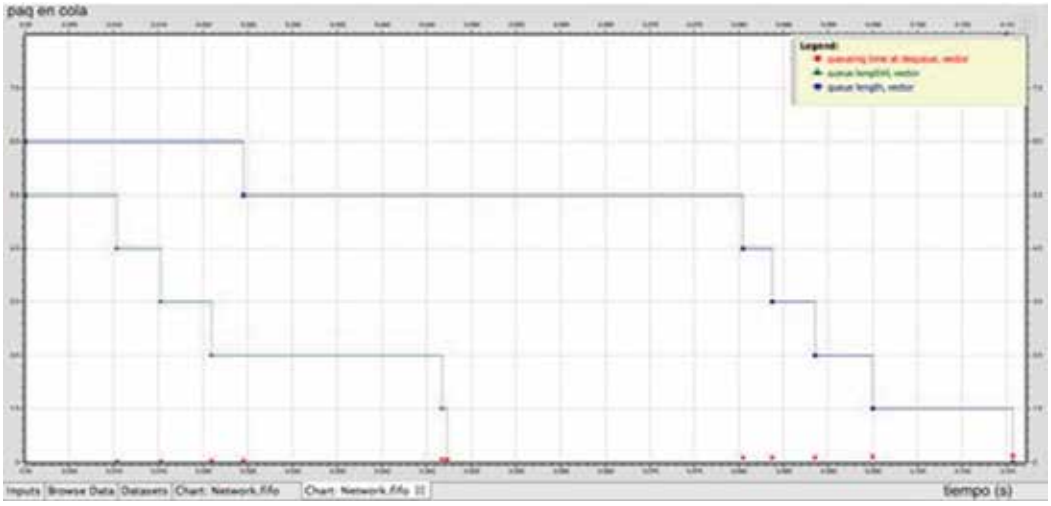
Figura 71. Comportamiento del encolamiento PQ



6.4.3. Gráfica LLQ

Para el algoritmo de encolamiento LLQ, es visible que hay extracción de paquetes en paralelo para ambas colas, aunque una tiene mayor prioridad. La Figura 72 presenta un solo paquete que hace esto, ya que para mayor comprensión de la simulación, esta se realizó con pocos paquetes.

Figura 72. Comportamiento del encolamiento LLQ



6.5. Comentarios finales

OMNeT++ ostenta ser un simulador de gran potencial. Al ser libre, permite que los usuarios desarrollen nuevas ideas y nuevos servicios. Aunque el manual presenta información bastante favorable, se tiene poca información sobre desarrollo, lo