Caso 3: Simulación de equipos y protocolos con OMNET++

Ane Sanz Rekalde

Rendimiento en Redes de Telecomunicación

Curso 2020-2021

Índice

1. Introducción

El objetivo de esta práctica es realizar una simulación de equipos de transmisión en redes con enlaces bajo protocolo ARQ utilizando la herramienta software Omnet++. Para ello, se va a realizar simulación de los protocolos, de la multiplexación de enlaces y de la función de conmutación.

La red que se va a simular consta de 5 nodos, donde los paquetes llegan de forma externa a 3 de ellos y los otros 2 nodos son considerados nodos finales. La siguiente figura muestra el esquema de la red.

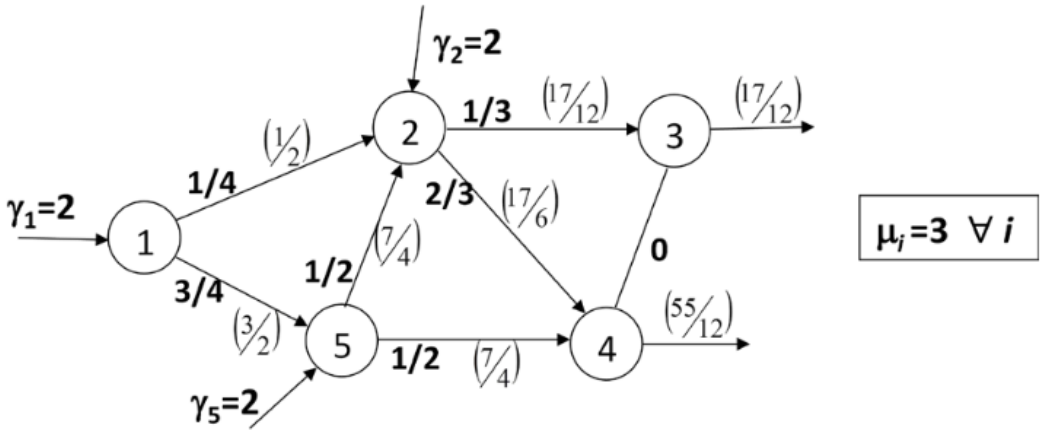


Figura 1: *Esquema de red a simular*

2. Simulación de la red – Stop and Wait

2.1. Creación de la red – fichero *.ned*

Para simular la red, en el fichero *.ned* se han definido los diferentes tipos de módulos o nodos que se necesitan. Para cada uno de los módulos, se han definido las entradas y salidas que tiene y los posibles parámetros que pudieran tener. En este caso, se han definido 3 módulos principales:

- **Módulo *source*:** Módulo encargado de generar los paquetes externos con tiempos que siguen una distribución exponencial. Este módulo solamente tiene un puerto de salida.
- **Módulo *node:*** Módulo intermedio encargado de recibir paquetes por varias entradas posibles y rutarlos por los dos posibles enlaces de salida que tiene. Por lo tanto, cada módulo nodo puede tener varios puertos de entrada y salida, que se han definido como *array*. Además, en este módulo se ha definido el parámetro *prob1* para definir la probabilidad de que un paquete se rute por el primer enlace de salida de los dos posibles.

- **Módulo *endNode:*** Módulo final que recibe los paquetes de alguno de los nodos intermedios y envía los ACK o NAK. Este módulo también puede tener varios puertos de entrada y salida que se han definido como *array.*

El siguiente cuadro muestra la definición de estos módulos:

```
simple source
{
    parameters:
        @display("i=block/source");
    gates:
        output out;
}
simple node
{
    parameters:
        @display("i=block/routing");
        double prob1;
    gates:
        input in[];
        output out[];
}
simple endNode
{
    parameters:
        @display("i=block/arrival");
    gates:
        input in[];
        output out[];
}
```

Después, se ha definido la red completa, donde se tiene que especificar cuántos módulos de cada tipo definidos va a haber y las conexiones entre los puertos de entrada y salida de los módulos, para crear las conexiones entre módulos. En este caso se han definido 3 fuentes, 3 nodos intermedios y 2 nodos finales. Además, también se ha definido un canal del tipo *DataRateChannel* para poder simular los enlaces. Este canal admite varios parámetros, y en esta práctica se han utilizado *datarate, packet error rate* y *delay.* Los valores que se muestran en el siguiente bloque son como ejemplo, ya que se han ido modificando en la realización de la práctica para ver su efecto.

```
network StopAndWait
{
    @display("bgb=500,500");
    types:
        channel C extends DatarateChannel
        {
            datarate = 1000bps;
            per = 0.05;
            delay = 0.1ms;
        }
    submodules:
        src[3]: source {
            @display("p=50,50,c,200");
        }
        node[3]: node;
        endNode[2]: endNode {
            @display("p=450,50,c,200");
        }
```

Por último, como ya se ha mencionado, se han definido todas las conexiones de entrada y salida para conectar los módulos mediante el canal C que se ha creado, como se muestra a continuación:

```
connections:
        endNode[0].out++ --> C --> node[0].in++;
        node[0].out++ --> C --> endNode[0].in++;

        endNode[1].out++ --> C --> node[0].in++;
        node[0].out++ --> C --> endNode[1].in++;

        node[0].out++ --> C --> node[2].in++;
        node[2].out++ --> C --> node[0].in++;

        node[0].out++ --> C --> node[1].in++;
        node[1].out++ --> C --> node[0].in++;

        endNode[1].out++ --> C --> node[2].in++;
        node[2].out++ --> C --> endNode[1].in++;

        node[1].out++ --> C --> node[2].in++;
        node[2].out++ --> C --> node[1].in++;

        src[0].out --> node[0].in++;
        src[1].out --> node[1].in++;
        src[2].out --> node[2].in++;
```

Una vez definidos los módulos y sus interconexiones, la red creada tiene el aspecto de la siguiente figura:

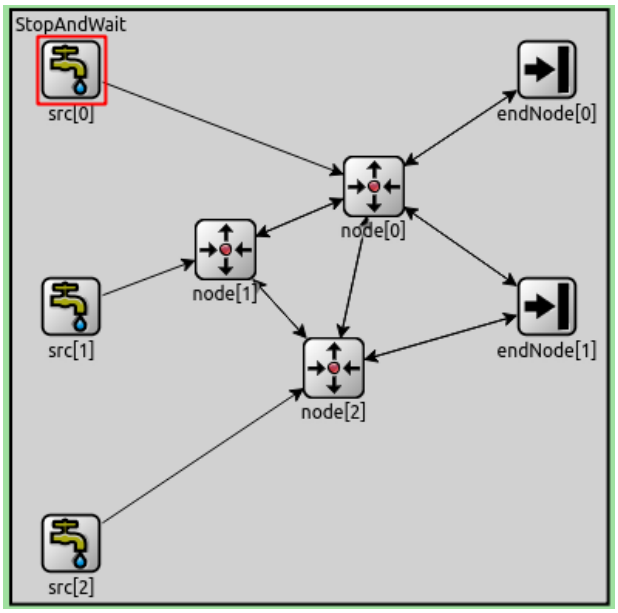


Figura 2: *esquema de red simulado*

2.2. Objeto personalizado de paquete

Para simular la red, se ha decidido crear un objeto personalizado (*myPacket*) para simular los paquetes que se envían, en lugar de utilizar los *cPacket* o *cMessage* que son parte de Omnet. Al principio, en este paquete solamente se han creado 3 parámetros que contienen el número de secuencia del paquete, el ID del nodo que ha creado el paquete y el tipo.

```
packet myPacket{

       unsigned int seq;

       unsigned int source;

       unsigned short type;    }
```

Para el campo *type,* se ha considerado que los paquetes con *type=0* son paquetes normales, los de *type=1* paquetes de tipo ACK y los de *type=2* paquetes de tipo NAK.

Una vez que la red está en marcha y la simulación se realiza correctamente de acuerdo al protocolo Stop and Wait, para poder visualizar diferentes datos estadísticos se añadirán otros campos al paquete.

2.3. Módulo nodo fuente

La programación del nodo fuente es bastante simple, ya que solamente tiene que generar y enviar paquetes en tiempos que siguen una distribución exponencial con media de 0.5 segundos. Para ello, en clase *source* se han implementado varios métodos. Primero se han definido 4 parámetros, por un lado, *meanLength* y *meanTime* para definir la media de la distribución exponencial que se va a seguir tanto para los tiempos en los que se envían los paquetes como para la longitud de los mismos. Por otro lado, *seq* funciona como contador para añadir el número de secuencia a los paquetes que se generan. Por último, se ha creado un objeto de tipo *cMessage* para utilizarlo como un automensaje que sirva para llamar cada cierto tiempo a la función que gestiona los mensajes.

```
class source : public cSimpleModule{
    private:
        double meanLength=1000;
        double meanTime=0.5;
        int seq=0;
        cMessage *event;
    protected:
        virtual void initialize() override;
        virtual void handleMessage(cMessage *msg) override;
        virtual myPacket* generarPaquete();
};
```

La primera función que se ha programado es la de inicialización, donde se crea un nuevo automensaje como se acaba de mencionar, y se calcula un tiempo aleatorio que sigue una distribución exponencial con media *meanTime.* Después, utilizando la función *scheduleAt* y añadiendo el momento de la simulación + el tiempo calculado, se llamará a la función *handleMessage* en el tiempo que se ha calculado.

```
void source::initialize(){
    event=new cMessage("event");

    double t=exponential(meanTime, 0);
    EV << "\nInitial time: " << t;
    scheduleAt(simTime()+t, event);
}
```

La función handleMessage llama a la función que genera un paquete del tipo *myPacket* y lo envía por el enlace de salida. Después, vuelve a calcular un tiempo aleatorio y lo utiliza en la función scheduleAt para volver a llamar a esta función y generar otro paquete.

```
void source::handleMessage(cMessage *msg){
    EV << "\nSending packet from Source " << getIndex() << " At time " <<
simTime();
    myPacket* paquete=generarPaquete();

    send(paquete,"out");
    double t=exponential(0.5, 0);
    scheduleAt(simTime()+t, event);
}
```

La función para generar los paquetes primero genera un nombre único que identifica la fuente que está generándolo y el número de secuencia, y crea el paquete con ese nombre. Después, añade los campos de número de secuencia (comenzando en 0 e incrementándose), origen (índice de fuente) y tipo (0 al tratarse de paquetes normales).

Además, para determinar el tamaño del paquete, calcula una longitud aleatoria que sigue una distribución exponencial con media *meanLength* y lo asigna.

```
myPacket* source::generarPaquete(){
    char packetName[20];
    sprintf(packetName,"Source%dPacket%d",getIndex(),seq);

    myPacket* p=new myPacket(packetName);
    p->setSeq(seq);
    p->setSource(getIndex());
    p->setType(0);
    seq++;
    double l=exponential(meanLength, 0);
    EV << "\nPacket length: " << l << " Seq: " << seq;
    p->setBitLength(l);

    return p;
}
```

2.4. Módulo nodo intermedio

Este módulo es el más complejo, ya que tiene que implementar varias funciones. Para entender mejor el funcionamiento que se ha llevado, se ha resumido el funcionamiento en el siguiente diagrama de flujo.

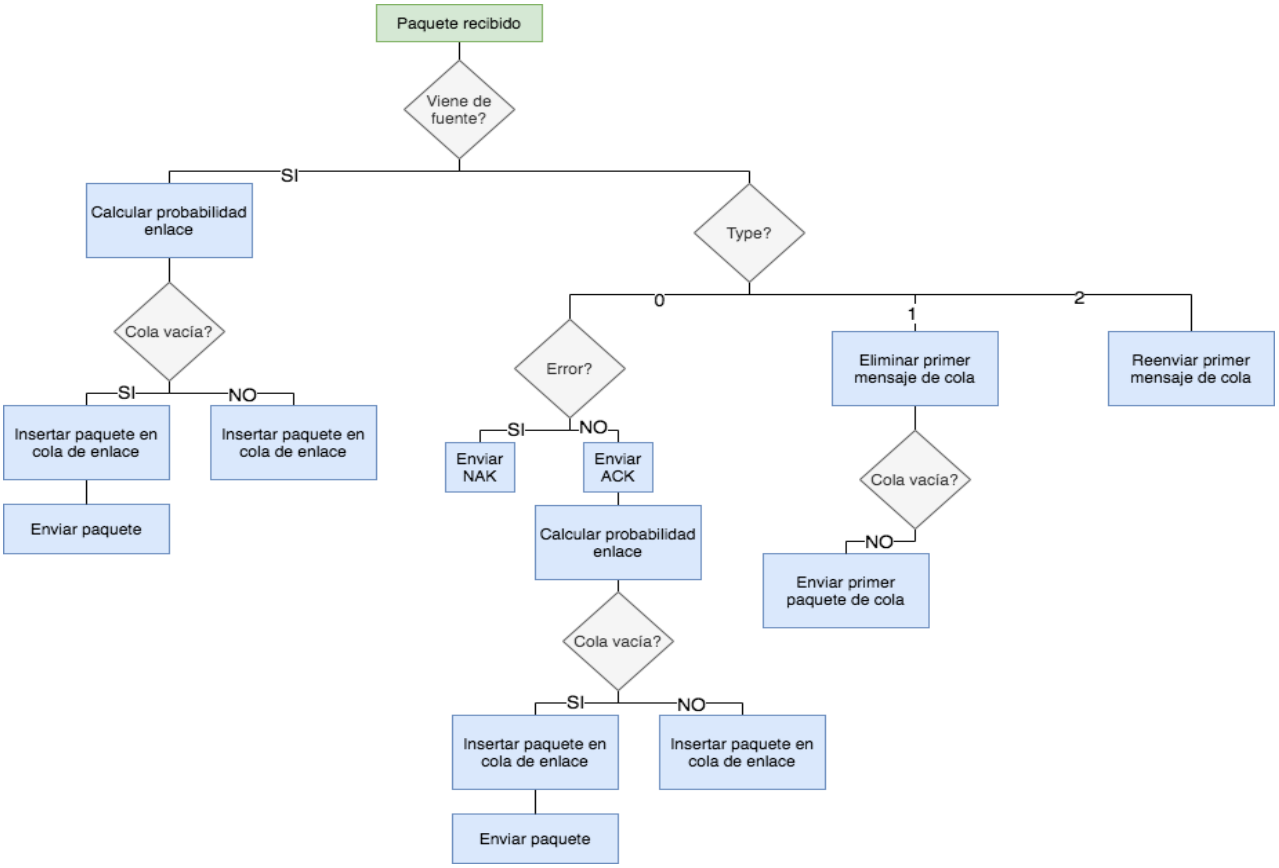


Figura 3: *diagrama de flujo del funcionamiento del nodo intermedio*

Cada vez que llega un paquete y se ejecuta el método *handleMessage*, el nodo comprueba si el paquete viene de un nodo fuente o de un nodo normal. Si viene de una fuente, se ha considerado que esos paquetes nunca tienen error, por lo que el nodo directamente pasa a calcular un número aleatorio, compararlo con la probabilidad de enviarlo por el primer enlace para así decidir por qué enlace tiene que enviar el paquete. Después, tiene que pasar a añadir el paquete a la cola correspondiente al enlace seleccionado, pero para ello primero comprueba si esa cola esta vacía. En caso de estar vacía, inserta el paquete en la cola y lo envía por ese enlace (realmente se envía una copia duplicada, para tener otra copiar en la primera posición de la cola por si hay que volver a enviar más adelante). En caso de no estar vacía, simplemente inserta el paquete al final de la cola.

```
void node::handleMessage(cMessage *msg){
    //delete msg;
    myPacket* p=check_and_cast<myPacket*>(msg);

    EV << "\nPaquete recibido de " << p->getSenderModule()->getClassName() <<
p->getSenderModule()->getIndex();

    if(strcmp(p->getSenderModule()->getClassName(),"source")==0){//Paquete
viene de las fuentes, no tienen error, no hay que enviar ACK/NAK
        EV << "\nIf de paquetes llegados de fuente";

        double rnd=((double)rand())/RAND_MAX;
        if(rnd<prob){
            enlace=0;
        }else{
            enlace=1;
        }

        if(queueOut[enlace]->isEmpty()){ //Si la cola esta vacia: insertar y
enviar
            queueOut[enlace]->insert(p);
            EV << "\nPaquete guardado en cola " << enlace;

            if(channelOut[enlace]->isBusy()==false){
                enviarPaqueteCola();
            }
        }else{ //Si hay paquetes en la cola: solo insertar
            queueOut[enlace]->insert(p);
            EV << "\nPaquete guardado en cola " << enlace;
        }
    }
```

Si el paquete no ha llegado de una fuente, significa que ha llegado de otro nodo y que por lo tanto puede ser de 3 tipos diferentes, por lo que hay que comprobar del nodo. Si es del tipo 0, un paquete normal, el nodo comprueba si el paquete ha llegado con error. Si ha tenido error, genera un paquete de tipo NAK y lo envía por el mismo enlace por el que ha llegado el paquete. Si no ha tenido error, genera un paquete ACK, lo envía por el mismo enlace y pasa a calcular un número aleatorio para decidir por qué enlace reenviar el paquete y lo inserta en esa cola.

Si es del tipo 1, ACK, significa que el último paquete que se ha enviado por ese enlace ha llegado correctamente, por lo que hay que eliminar el primer paquete de la cola de ese enlace. Una vez borrado, se envía por el mismo enlace el que ahora es el primer paquete de la cola, siempre y cuando que no esté vacía

Si es del tipo 2, NAK, significa que el paquete que se ha enviado por ese enlace no ha llegado bien, por lo que hay que volver a enviar por ese enlace el primer paquete almacenado en la cola correspondiente.

Función que comprueba el tipo de paquete y realiza algunas funciones:

```
else{//Paquete viene de otro nodo
        EV << "\n If de paquetes llegados de nodo";

        int type=p->getType();
        int gateIndex=p->getArrivalGate()->getIndex();

        switch (type){
                case 0://Mensaje normal
                    EV << "\nPaquete recibido";
                    tratarPaqueteNodo(p);
                    break;
                case 1://ACK
                    EV << "\nACK recibido";
                    delete(p);
                    queueOut[gateIndex]->pop();

                    EV << "\nqueue length: " << queueOut[gateIndex]-
>getLength();

                    if(!queueOut[enlace]->isEmpty() && !channelOut[enlace]-
>isBusy()){
                        enviarPaqueteCola();
                    }
                    break;
                case 2://NAK
                    EV << "\nNAK recibido";
                    delete(p);
                    if(channelOut[enlace]->isBusy()==false){
                        enviarPaqueteCola();
                    }
                    break;
            }
    }
```

Función que envía una copia duplicada del primer mensaje de la cola:

```
void node::enviarPaqueteCola(){
    myPacket* p=check_and_cast<myPacket*> (queueOut[enlace]->front());
    send(p->dup(),"out",enlace);

    EV << "\nPaquete enviado por enlace " << enlace;
}
```

Función para cuando se recibe un paquete de tipo 0:

```
void node::tratarPaqueteNodo(myPacket *p){
    cGate* g=p->getArrivalGate();

    if(p->hasBitError()){
        //SEND NAK
        myPacket* nak=new myPacket("NAK");
        nak->setSeq(seqNak);
        nak->setSource(getIndex());
        nak->setType(2);
        seqNak++;

        send(nak,"out",g->getIndex());
        EV << "\nRespuesta " << nak->getName() << " enviada por enlace " << g-
>getIndex();

    }else{
        //SEND ACK
        myPacket* ack=new myPacket("ACK");
        ack->setSeq(seqAck);
        ack->setSource(getIndex());
        ack->setType(1);
        seqAck++;
```

```
        send(ack,"out",g->getIndex());
        EV << "\nRespuesta " << ack->getName() << " enviada por enlace " << g-
>getIndex();

        double rnd=((double)rand())/RAND_MAX;
        if(rnd<prob){
            enlace=0;
        }else{
            enlace=1;
        }
        queueOut[enlace]->insert(p);

        EV << "\nPaquete guardado en cola " << enlace;
    }
}
```

Aún así, antes de que se ejecuten las funciones que se acaban de describir, en este módulo se han definido varias variables como unos números de secuencia/contadores para paquetes ACK y NAK, un identificador del número del enlace seleccionado en cada momento, la probabilidad de enviar un paquete por el primer enlace (se recupera de los parámetros del módulo), dos canales y dos colas para los dos enlaces. Además, algunas de estas variables se inicializan al principio.

```
class node : public cSimpleModule{
    protected:
        int seqAck;
        int seqNak;
        int enlace;
        double prob;
        cChannel* channelOut[2];
        cQueue* queueOut[2];
    protected:
        virtual void initialize() override;
        virtual void handleMessage(cMessage *msg) override;
        virtual void tratarPaqueteNodo(myPacket *p);
        virtual void enviarPaqueteCola();
};
```

```
void node::initialize(){
    seqAck=0;
    seqNak=0;
    prob=(double)par("prob1");
    EV << "\nProb primer enlace: " << prob;

    channelOut[0]=gate("out", 0)->getTransmissionChannel();
    channelOut[1]=gate("out", 1)->getTransmissionChannel();

    queueOut[0]=new cQueue("queue1");
    queueOut[1]=new cQueue("queue2");
}
```

2.5. Módulo nodo final

El módulo del nodo final implementa parte de la funcionalidad del nodo intermedio. De hecho, este módulo solamente va a recibir paquetes del tipo 0 y que además siempre van a venir de un nodo intermedio, por lo que no tiene que comprobar el origen ni el tipo de mensaje. Suponiendo eso, cada vez que le llega un paquete tiene que comprobar si ha tenido error y actuar de la misma forma que lo hacía el nodo para enviar los ACK y NAK. Sin embargo, al tratarse del nodo final, cuando se envía un ACK no hay que reenviar el paquete por ningún otro enlace.

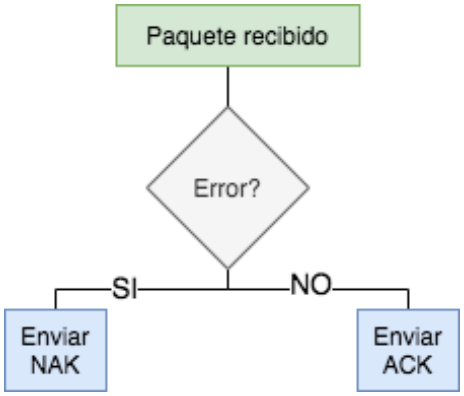


Figura 4: *Diagrama de flujos del nodo final*

Para ello, se han creado algunas variables como contadores que se inicializan al principio de la simulación:

```
class endNode : public cSimpleModule{
    private:
        int seqAck;
        int seqNak;
        int rcvPack;
    protected:
       virtual void initialize() override;
       virtual void handleMessage(cMessage *msg) override;

};
```

```
void endNode::initialize(){
    seqAck=0;
    seqNak=0;
    rcvPack=0;
    WATCH(rcvPack);
}
```

Después, se ha implementado la funcionalidad de comprobar error y enviar ACK o NAK en cada caso, como se ha explicado. Además, cuando se recibe un paquete sin error, se incrementa el contador de paquetes recibidos para llevar un control de los paquetes que recibe cada nodo final.

```
int gateIndex=p->getArrivalGate()->getIndex();
if(p->hasBitError()){//SEND NAK
    myPacket* nak=new myPacket("NAK");
    nak->setSeq(seqNak);
    nak->setSource(getIndex());
    nak->setType(2);
    seqNak++;
    send(nak,"out",gateIndex);
EV << "\nRespuesta " << nak->getName() << " enviada por enlace " << gateIndex;

}else{//SEND ACK
    myPacket* ack=new myPacket("ACK");
    ack->setSeq(seqAck);
    ack->setSource(getIndex());
    ack->setType(1);
    seqAck++;
    send(ack,"out",gateIndex);
EV << "\nRespuesta " << ack->getName() << " enviada por enlace " << gateIndex;
EV << "\nPaquete llegado a destino sin errores";
    rcvPack++;
       }
   }
}
```

2.6. Simulación

Una vez programados todos los módulos, se ha pasado a simular la red. De la forma que se ha programado, los parámetros de probabilidad de cada enlace se han dejado sin valor hasta el momento de la simulación, para poder hacer cambios de forma sencilla y ver las diferencias al cambiar valores. Por lo tanto, lo primero que se hace al simular es introducir las probabilidades para cada uno de los tres módulos.

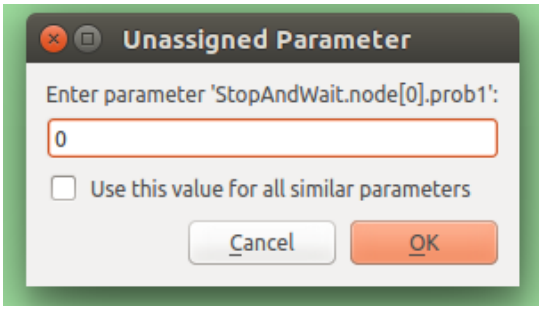
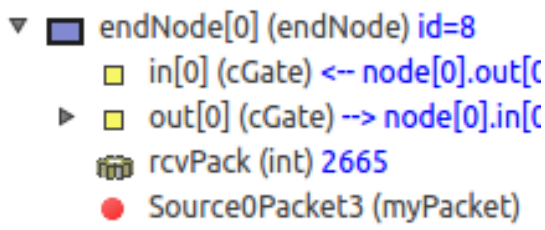
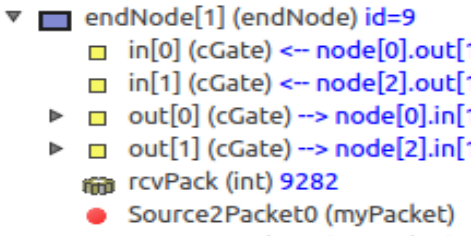


Figura 5: *Asignar valor a los parámetros de probabilidad*

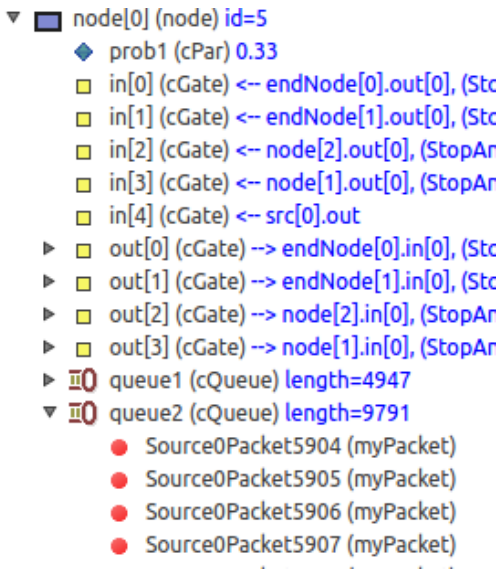
Una vez asignados los valores, la red se construye y está preparada para simularse. Se puede simular paso a paso viendo en detalle cada evento, a velocidad normal o a velocidad más rápida. A la hora de hacer la simulación se han ido comprobando por una parte que los logs que salen en la consola son los esperados según lo programado, y así se ha podido ir comprobando el funcionamiento de la red. Por lo tanto, observando tanto los logs como la red, se ha visto que la red funciona correctamente según el protocolo Stop and Wait que se ha programado.

Por ejemplo, en las siguientes figuras se van a mostrar los resultados después de haber simulado la red durante 8.000 segundos. Al parar la simulación al cabo de este tiempo, se puede observar que el nodo final 0 ha recibido 2665 paquetes correctamente, mientras que el nodo final 1 9282. Esto se debe a que se han introducido las probabilidades del enunciado de la práctica, por lo que es lógico que más paquetes sigan un camino que les lleve al nodo final 1.





De la misma forma, también es posible ver cuántos paquetes hay almacenados en las colas de los nodos intermedios, es decir, cuántos paquetes hay en la red esperando a ser enviados. Por ejemplo, en el caso del nodo 0, se puede ver que la primera cola tiene 4947 paquetes almacenados, mientras que la segunda cola 9791. Además, también es posible observar que dentro de cada una de las colas los paquetes están bien ordenados por tiempos de llegada.



A simple vista se puede observar que la red tiene un problema de saturación, ya que las colas están llenas de paquetes, porque el ritmo al que se están generando los paquetes es superior a la capacidad de la red a procesar los paquetes.

Para solucionar esto, se ha intentado modificar algunos valores, y poner como *datarate* del canal 9,6 kbps para que la capacidad sea mayor. Aún así al cambiar este valor la simulación no se realiza correctamente, ya que cuando pasa un tiempo de simulación se para debido a un error. Lo mismo ocurre si se intenta reducir la longitud media de los paquetes.

No ha sido posible encontrar el motivo de este error, y la única forma que he encontrado de al menos reducir el ritmo de saturación ha sido reduciendo el ritmo al que se generan los paquetes en las fuentes. Por lo tanto, un tiempo medio entre paquetes de 2 segundos para que la red no se sature tan pronto y poder observar mejor las estadísticas que se van a mostrar.

3. Visualización de estadísticas

En este apartado se han sacado estadísticas sobre el número de saltos que realiza cada paquete para cuando llega a cualquiera de los nodos finales. Para ello ha sido necesario añadir un campo extra en los paquetes, llamado *hopCount*, que se va incrementando cada vez que llega a un nodo sin errores.

Después, en el nodo final se ha recuperado la información sobre el número de saltos y se ha almacenado. Por una parte, se han guardado los datos de forma que se vaya representando el número de saltos en tiempo real cada uno de los paquetes que llega sin errores a un nodo final. Por ejemplo, la siguiente figura muestra los paquetes que se han recibido entre los segundos 580 y 660 de simulación en el nodo final 1, y se puede observar que la mayoría de paquetes llegan con 2 o 3 saltos, mientras que algunos realizan 4 saltos.

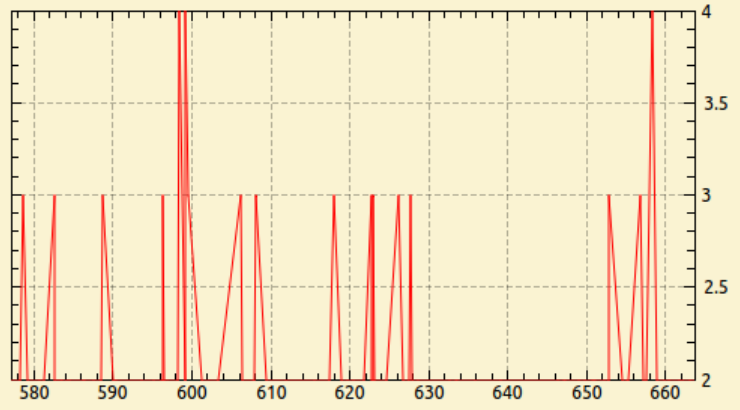


Figura 6: *Número de saltos en el tiempo*

De la misma forma, se ha representado un histograma para ver cómo se distribuyen los números de saltos en todos los paquetes que han llegado, y se puede ver claramente que casi el 80% de los paquetes han tenido 2 saltos, mientras que algo mas del 20% han tenido 3 saltos y menos del 10% 4 saltos.

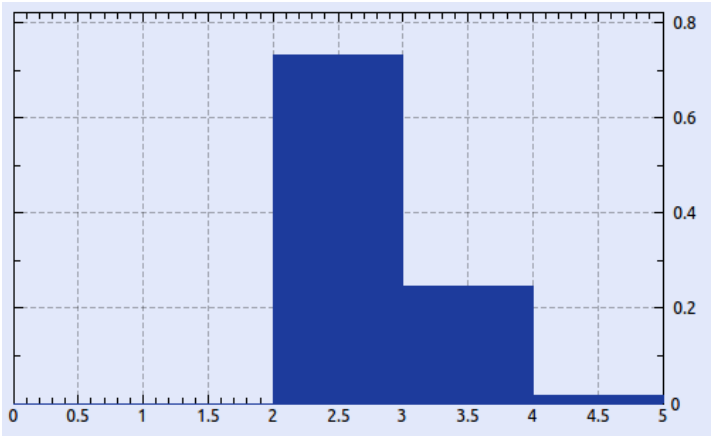


Figura 7: *Histograma de número de saltos*

Por último, también se han creado unos ficheros que guardan información general sobre el máximo, mínimo y media de número de saltos, entre otras cosas. Siguiendo con el nodo final 1, se puede apreciar que ha recibido 703 mensajes, con una media de 2,28 saltos, un mínimo de 2 saltos y un máximo de 4.

```
statistic StopAndWait.endNode[1] "hop count"
field count 703
field mean 2.2859174964438
field stddev 0.49142299222364
field min 2
field max 4
field sum 1607
field sqrsum 3843
attr type int
bin     -inf    0
bin     0       0
bin     1       0
bin     2       515
bin     3       175
bin     4       13
bin     5       0
```

Figura 8: *Resumen de número de saltos*

4. Conclusiones

En esta practica se ha realizado una simulación de una red que utiliza el protocolo Stop and Wait. Aún así, cabe destacar que no se ha implementado la funcionalidad más común de este protocolo, que utiliza temporizadores que esperan hasta recibir ACK, y si no se recibe en ese tiempo se reenvía el paquete. En este caso se ha implementado un Stop and Wait que hace uso de paquetes de tipo NAK para rechazar un paquete que ha llegado con errores.

Por otra parte, también es necesario comentar que de haber tenido más tiempo, esta práctica tiene mucho contenido donde poder seguir profundizando en el funcionamiento de Omnet como herramienta de simulación.

Por ejemplo, por falta de tiempo en el apartado de visualización de parámetros estadísticos solamente se han mostrado los relativos al número de saltos. Sin embargo, estaría bien haber visualizado también el retardo extremo a extremo de cada paquete, el retardo medio de todos los paquetes, información relevante a los circuitos virtuales, etc.

Además, también se ha considerado que los paquetes nunca se pierden en el camino, siempre llegan al otro extremo aunque sea con error. Por lo tanto, estaría bien poder implementar la posibilidad de que los paquetes se pierdan y gestionar esas pérdidas con timeouts, por ejemplo.

Por último, también ha faltado implementar el protocolo Go Back N para poder así comparar los funcionamientos y rendimientos de los dos protocolos.