# Memoria Prácticas Diseño de Redes

Rubén Burgué Pérez Login: ruben.bperez

Óscar Fernández Orellana Login: oscar.orellana

# Práctica 1. Introducción a Omnet++

**1. Realización del tutorial:**

- Tictoc1: Se crea una red que consiste en 2 nodos. Uno de estos nodos creará un paquete y los nodos lo intercambiarán indefinidamente. Estos nodos se llamarán tic y toc.
  En el fichero tictoc1.net se define la topología de la red. En el fichero txc1.cc se define el comportamiento de los nodos heredando de la clase cSimpleModule.

- Tictoc2: Se añade iconos a los nodos y mensajes de debug en los cuales se muestra un mensaje al crear el paquete y cada vez que se reenvía.

- Tictoc3: Se añade un contador al módulo y se borra el paquete después de que el número de reenvíos sea igual al contador. Se inicializa la variable en la función initialize() y se decrementa en handeMessage(). Se añade información sobre los contadores en los mensajes de debug.

- Tictoc4: Se añade el envío de parámetros a los nodos. Se le pasa el parámetro con el número del contador y además, se permite configurar que nodo inicia la comunicación. Como en el archivo omnetpp.ini sólo se especifica el valor del contador de toc, tic coge el valor por defecto 2. Cuando el contador de uno de ellos llega a cero se acaba la comunicación.

- Tictoc5: Se crean 2 nuevos módulos Tic5 y Toc5 heredando del módulo que ya teníamos Txc5.cc. Tic5 tiene valor "True" en la variable sendMsgOnInit y Toc5 a "false". El nodo tic5 heredará de Tic5 y toc5 heredará de Toc5.
  En este caso en el fichero omnetpp.ini se especifica que el contador en los dos nodos se inicia a 5 utilizando el comodín ** (**.limit=5).

- Tictoc6: Hasta ahora los mensajes se reenviaban nada más recibirse. Ahora se va a añadir un delay, en Omnet++ esto se consigue haciendo que el nodo se envíe un mensaje a sí mismo. Estos mensajes son llamados self-menssages.
  Se añade un nuevo mensaje al módulo Txc6 y se envía este auto mensaje con la función scheduleAt() especificando cuando debe ser devuelto al nodo. Ahora en la función handleMessage() se debe diferenciar cuando se trata del auto mensaje o del mensaje procedente del otro nodo.

- Tictoc7: Se añaden números aleatorios para el delay de reenvío del paquete y adicionalmente se implementa una probabilidad de pérdida del paquete usando un algoritmo determinista de modo que depende del valor de una semilla. Una vez se pierde el paquete, se termina la simulación, por tanto no es necesario el uso de un contador.

- Tictoc8: Se cambia el comportamiento de tic y de toc, ahora toc puede perder el paquete y tic cada vez que envía un paquete establece un timeout para comprobar si el paquete se pierde o no se pierde, en caso de que expire el timeout (el paquete se pierde), envía un paquete nuevo; si no expira (recibe el paquete) cancela el timeout, elimina el paquete y envía uno nuevo. Para simular el timeout se utiliza la función scheduleAt() y para cancelarlo cancelEvent().

- Tictoc9: Cada vez que toc pierde un paquete, tic en lugar de generar uno nuevo, copia el contenido del paquete perdido y lo reenvía. Para comprobar que el paquete es exactamente igual, se etiquetan los paquetes con un número de secuencia.

- Tictoc10: Se genera una red con varios nodos interconectados entre sí de tal manera que se envía el mensaje desde el nodo inicial con destino a un nodo determinado. El mensaje irá recorriendo la red de forma aleatoria hasta alcanzar el destino. Cada nodo elige aleatoriamente hacia que nodo de los que tiene conectados envía el paquete. Cada nodo tiene varios Gates de entrada y salida, para realizar esto, los Gates de entrada y salida se definen como vectores.

- Tictoc11: Se añade el concepto de canal, que define un tipo de conexión con el cual se pueden definir diferentes tipos de conexiones de características distintas. En este ejemplo, se configura un canal con un valor único para el delay. Este nuevo canal hereda de ned.DelayChannel, el canal se define dentro de network en la sección types de modo que sólo se puede usar dentro de esa red. En la sección connections del network donde antes se especificaba el delay, ahora se indica el channel (canal) que se definió en la sección types.

- Tictoc12: OMNeT++ 4 soporta conexiones bidireccionales, en este ejemplo se van a usar este tipo de conexiones. Para ello se eliminan los dos vectores de conexiones definidos anteriormente y se substituyen por conexiones bidireccionales a través de un canal. Para especificar si la conexión es de entrada o de salida se utilizan los sufijos $i y $o en las conexiones bidireccionales.

- Tictoc13: En este ejemplo, la dirección de destino del mensaje no es predefinida, sino que se genera un destino aleatorio, para ello se crea una clase TicTocMsg13 que hereda de cMessage y se le añade el campo destino. Además ahora cuando el nodo destino reciba el mensaje, este creará un nuevo mensaje con un destino aleatorio y se enviará otra vez a la red.

- Tictoc14: Se añaden dos contadores en los nodos, uno de mensajes enviados y otro de mensajes recibidos y se edita la interfaz para que muestre estos contadores encima de cada nodo.

- Tictoc15: Se añade la opción de configuración record-eventlog al omnetpp.ini lo cual hace que los mensajes de log del kernel de Omnet se registren. Se recogen estadísticas de los mensajes entre los nodos y se registran al final de la simulación.

- Tictoc16: Se añaden señales para poder consultar las estadísticas en tiempo de ejecución.

**2. Tictoc17.**

Se añade a omnetpp.ini la red Tictoc17:

[Config Tictoc17]
network = Tictoc17
record-eventlog = true
**.tic[*].hopCount.result-recording-modes = +histogram

El contenido del fichero tictoc17.ned es el mismo que para tictoc16.ned dado que la topología de la red es exactamente la misma. Al mismo tiempo, el contenido de tictoc17_m.h y tictoc17_m.c son iguales a los de tictoc16.

Como primera aproximación para hacer un enrutamiento más inteligente se pensó en eliminar la Gate por la que se recibe el mensaje de los posibles Gates por los que reenviar el mensaje. Para ello, hay cambiar el comportamiento de envío de paquetes en la función forwardMessage(), comprobando por qué Gate llegó el mensaje con getArrivalGate() y getIndex(). En este punto se comprueba si hay más de un Gate en el nodo, en cuyo caso se genera un Gate aleatorio hasta que sea distinto del Gate de entrada; así se asegura que se envía el mensaje por un Gate diferente. Hay que tener en cuenta que el primer mensaje no tiene un Gate de entrada, por ello se elige un puerto de salida aleatoriamente. La imagen 1 muestra el código de la función forwardMessage().

```
void Txc17::forwardMessage(TicTocMsg17 *msg)
{
    // Increment hop count.
    msg->setHopCount(msg->getHopCount()+1);

    int n = gateSize("gate");
    int k = 0;
    cGate *arrivalGate = msg->getArrivalGate();
    int arrivalGateIndex;

    // If is not the node that created the message
    if (arrivalGate != NULL) {
        arrivalGateIndex = arrivalGate->getIndex();

        // If there is more than one gate in the node
        if (n != 1) {
            k = intuniform(0,n-1);
            // While the gate selected is not the arrival gate
            while (k == arrivalGateIndex)
                k = intuniform(0,n-1);
        }

    } else
        if (n != 1)
            k = intuniform(0,n-1);

    EV << "Forwarding message " << msg << " on gate[" << k << "]\n";
    send(msg, "gate$o", k);
}
```

Imagen 1. Código Txc17.cc aproximación 1(forwardMessage())

La segunda aproximación consiste en almacenar en cada nodo porque Gates ha recibido un mensaje. La intención de esto es hacer que cada nodo "aprenda" y decida a donde debe enviar el mensaje teniendo en cuenta por dónde ha recibo el mensaje, evitando bucles para que llegue a su destino lo más rápido posible. Para ello se generan dos nuevas variables en cada nodo, lastMessageId que contendrá el identificador del último mensaje recibido, lastGates que es un array en el que cada posición identifica un Gate del nodo. Cada posición puede tomar el valor 0 o 1; un valor 0 indica que no se ha recibido el mensaje con identificador lastMessageId por ese Gate y un valor 1 indica que se ha recibido el mensaje por ese Gate. Estas variables se inicializan en el método Initialize(), lastMessageId con valor -1 y las posiciones de lastGates con valor 0 como se puede ver en la imagen 2.

```
class Txc17 : public cSimpleModule
{
  private:
    simsignal_t arrivalSignal;
    // Variables to control the last arrival gates of a message in the node
    int lastMessageId;
    int *lastGates;

  protected:
    virtual TicTocMsg17 *generateMessage();
    virtual void forwardMessage(TicTocMsg17 *msg);
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(Txc17);

void Txc17::initialize()
{
    arrivalSignal = registerSignal("arrival");
    // Never received a message
    lastMessageId = -1;
    // Initialization of the control array
    lastGates = (int *) malloc(gateSize("gate") * sizeof(int));
    for (int i = 0; i < gateSize("gate"); i++) {
        lastGates[i] = 0;
    }
    // Module 0 sends the first message
    if (getIndex()==0)
    {
        // Boot the process scheduling the initial message as a self-message.
        TicTocMsg17 *msg = generateMessage();
        scheduleAt(0.0, msg);
    }
}
```

Imagen 2. Código Txc17.cc aproximación 2

En la imagen 3, se puede ver el código de la función forwardMessage(). Al recibir un mensaje se comprueba si el identificador del mensaje coincide con lastMessageId. En caso contrario se actualiza el valor de lastMessageId con el identificador del mensaje y se inicializa el array lastGates a 0. Si el nodo tiene más de un Gate, comprueba si ya ha recibido el mensaje por todos sus gates, en cuyo caso reinicializa todos sus valores a 0 excepto el Gate por el que recibió el mensaje para evitar que los nodos que tienen el destino a más de dos saltos con ramificaciones intermedias no vuelvan a recibir el mensaje antes de que llegue a su destino. Si no ha recibido el mensaje por todos sus Gates, escoge aleatoriamente uno por el cual no haya recibido ya ese mensaje y lo envía. En caso de sólo tener un Gate, todos los mensajes serán enviados por ese.

```
void Txc17::forwardMessage(TicTocMsg17 *msg)
{
    // Increment hop count.
    msg->setHopCount(msg->getHopCount()+1);

    int n = gateSize("gate");
    int k = 0;
    cGate *arrivalGate = msg->getArrivalGate();
    int arrivalGateIndex;

    // If is not the node that created the message
    if (arrivalGate != NULL) {
        arrivalGateIndex = arrivalGate->getIndex();

        // Check if the message was never received to initialize the values
        if (msg->getId() != lastMessageId) {
            lastMessageId = msg->getId();
            for (int i = 0; i < gateSize("gate"); i++) {
                    lastGates[i] = 0;
            }
        }
        // Adds the arrival gate to the control array
        lastGates[arrivalGateIndex] = 1;

        // If there is more than one gate in the node
        if (n != 1) {
            // Check if the node received the message from all the gates
            bool allGatesFull = true;
            for(int i = 0; i < n; i++) {
                if (lastGates[i] == 0)
                    allGatesFull = false;
            }

            k = intuniform(0,n-1);
            // If not all the gates were used
            if (!allGatesFull) {
                // Select a new non used gate to send the message
                while (lastGates[k] == 1)
                    k = intuniform(0,n-1);
            } else {
                // If all the gates were used, then reinitialize the control array
                for (int i = 0; i < gateSize("gate"); i++) {
                    if (i != arrivalGateIndex)
                        lastGates[i] = 0;
                }
                while (lastGates[k] == 1)
                    k = intuniform(0,n-1);
            }
        }
    } else
        if (n != 1)
            k = intuniform(0,n-1);

    EV << "Forwarding message " << msg << " on gate[" << k << "]\n";
    send(msg, "gate$o", k);
}
```

Imagen 3. Código Txc17.cc aproximación 2

### 3. Comparación gráfica de número de saltos medio.

En las imágenes 4, 5 y 6 se pueden observar los histogramas que muestran el número de saltos medio en los 3 casos tratados. Los nodos están ordenados del 0 al 5 de izquierda a derecha.

Tictoc16:

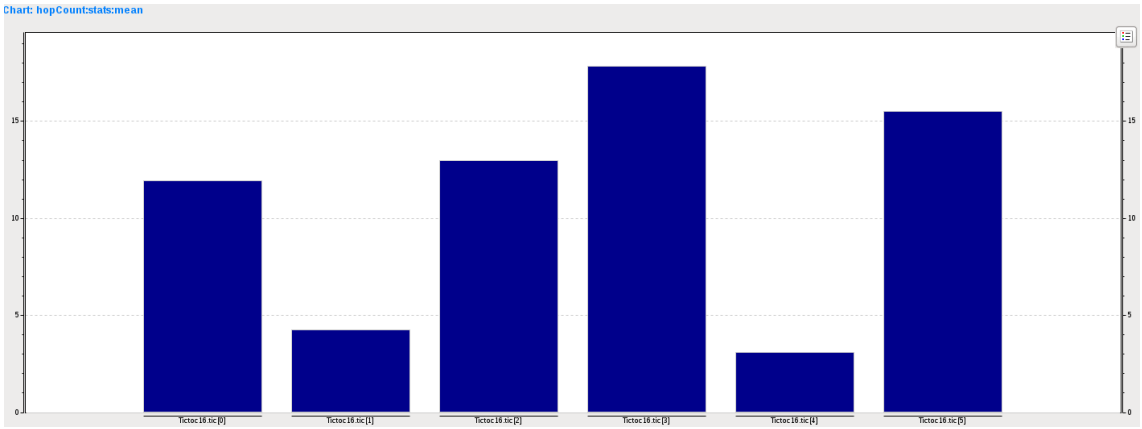


Imagen 4. Histograma TicToc16

tic[0]: 11,93

tic[1]: 04,27

tic[2]: 13,00

tic[3]: 17,82

tic[4]: 03,10

tic[5]: 15,50

Primera aproximación:

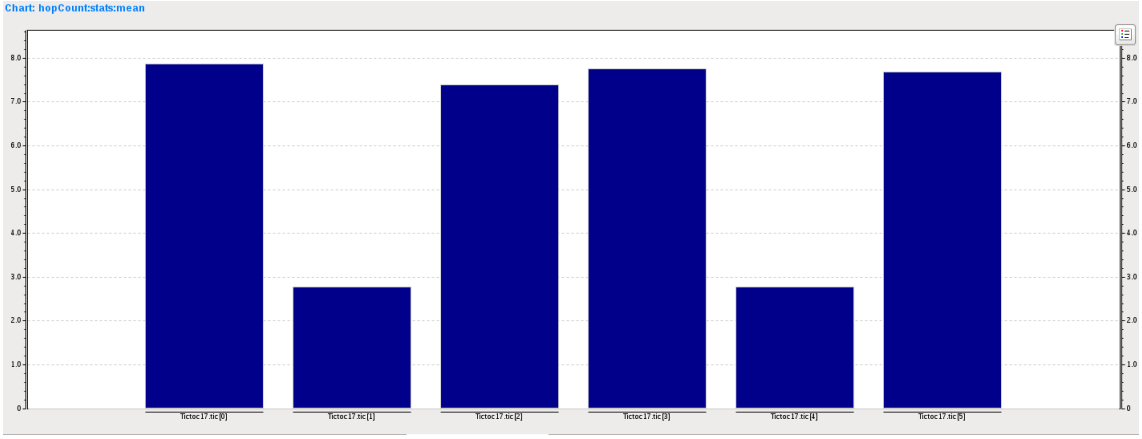


Imagen 5. Histograma aproximación 1

tic[0]: 7,86

tic[1]: 2,77

tic[2]: 7,39

tic[3]: 7,76

tic[4]: 2,77

tic[5]: 7,68

Segunda aproximación:

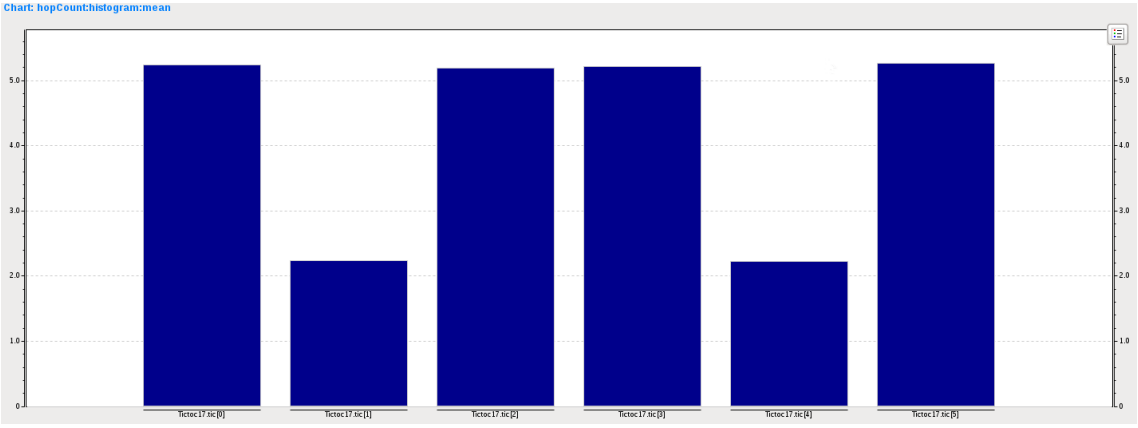


Imagen 6. Histograma Aproximación 2

tic[0]: 5,24

tic[1]: 2,24

tic[2]: 5,20

tic[3]: 5,22

tic[4]: 2,23

tic[5]: 5,26

# Práctica 2. IPv6 sobre INET

### 1. Analizar funcionamiento del ejemplo.

Inicialmente, se analiza cómo funciona el ejemplo /examples/ipv6/nclientes, donde se muestra un escenario básico con un único cliente conectado a un servidor telnet a través de 3 routers intermedios.

En el ejemplo se definen dos topologías de red similares, una utilizando canales Ethernet (NClientsEth.ned) y otra utilizando canales PPP (NClientsPPP.ned). En el fichero omnetpp.ini se definen tres configuraciones de red distintas utilizando las topologías anteriores. Dos de ellas utilizan NClientsPPP.ned, una que define aplicaciones que corren sobre el protocolo TCP y otra que corre sobre el protocolo SCTP. La otra configuración utiliza NClientsEth.ned y define aplicaciones que corren sobre el protocolo TCP.

Para analizar el ejemplo, se analiza en mayor detenimiento la configuración de TCP y Ethernet que se puede ver en las imágenes 7 y 8.

```
[General]
tkenv-plugin-path = ../../../etc/plugins

# number of client computers
*.n = 1

[Config TCP_APP]

# tcp apps
**.cli[*].numTcpApps = 1
**.cli[*].tcpApp[*].typename = "TelnetApp"
**.cli[0].tcpApp[0].localAddress = "aaaa:b::8aa:ff:fe00:7" #is this the source addr of the client's TCP app?
**.cli[1].tcpApp[0].localAddress = "aaaa:b::8aa:ff:fe00:8"
**.cli[0].tcpApp[0].localPort = -1
**.cli[1].tcpApp[0].localPort = -1
**.cli[*].tcpApp[0].connectAddress = "srv"
#**.cli[*].tcpApp[0].connectAddress="bbbb::"
**.cli[0].tcpApp[0].connectPort = 1000 #same destination port numbers
**.cli[1].tcpApp[0].connectPort = 1000 #same destination port numbers

**.cli[*].tcpApp[0].startTime = uniform(10s,15s)
**.cli[*].tcpApp[0].numCommands = exponential(1)
**.cli[*].tcpApp[0].commandLength = exponential(1B)
**.cli[*].tcpApp[0].keyPressDelay = exponential(0.1s)
**.cli[*].tcpApp[0].commandOutputLength = exponential(40B)
**.cli[*].tcpApp[0].thinkTime = truncnormal(2s,3s)
**.cli[*].tcpApp[0].idleInterval = truncnormal(3600s,1200s)
**.cli[*].tcpApp[0].reconnectInterval = 30s

**.srv.numTcpApps = 1
**.srv.tcpApp[*].typename = "TCPGenericSrvApp"
**.srv.tcpApp[0].localAddress = ""
**.srv.tcpApp[0].localPort = 1000
**.srv.tcpApp[0].replyDelay = 0
#**.srv.tcpApp[1].localAddress=""  #created another one but unused for now
#**.srv.tcpApp[1].localPort=3168
#**.srv.tcpApp[1].replyDelay = 0

# tcp settings
**.tcpApp[*].dataTransferMode = "object"
```

Imagen 7. Definición TCP_APP

```
[Config ETH]
description = "ETH network"
extends = TCP_APP
network = NClientsEth

# Ethernet NIC configuration
**.eth[*].queueType = "DropTailQueue" # in routers
**.eth[*].queue.dataQueue.frameCapacity = 10  # in routers

#**.eth[*].mac.txrate = 10Mbps
**.eth[*].mac.duplexMode = true
```

Imagen 8. Definición configuración ETH network

El funcionamiento de esta red ipv6 es el siguiente:

Los routers se conectan con una dirección ipv6 link-local y hacen un Neighbor Discovery en busca de de Duplicated Address Detection (DAD), si nadie contesta a este mensaje enviado en Multicast a todos los nodos conectados, después de un timeout, el router se asigna la IPv6 así mismo con la convicción de que es única en ese Scope. Es necesario hacerlo para que los routers puedan asociarse una IPv6 única en su ámbito de forma que los paquetes les lleguen solo a ellos y no haya IP duplicadas en la red. Una vez los routers tienen su IP, el cliente y el servidor pueden comenzar a configurarse enviando un Router Discovery (RD) para descubrir cuál es el prefijo de red que tienen asociado. Una vez reciben el identificador a través de un Router Advertisement (RA), generan su IPv6 con Stateless Autoconfiguration, al tener la dirección IPv6 pueden enviar Neighbor Solicitation (NS) a la dirección de multicast con la intención de comprobar si su IP está siendo utilizada por otra interfaz o no (Duplicated Address Detection o DAD), el funcionamiento de este protocolo es igual que en el caso de los routers. Una vez acabado este proceso se puede decir que las IP de la red están debidamente configuradas para iniciar la conexión.

El cliente inicia una conexión telnet con el servidor. Para ello le envía un mensaje con el Flag SYN activado (three-way handshake negotiation o negociación en tres pasos de TCP). Una vez el SYN llega al servidor, este contesta con un mensaje con los flags SYN y ACK activados, a lo que el cliente confirma con un ACK. En este momento puede comenzar el intercambio de información. Cabe destacar, que por cada paquete enviado, el otro  nodo de la conexión debe enviar un ACK. Una vez acabada la conexión, el cliente envía un mensaje con el flag FIN activado. A esto el servidor responde con un ACK, en cuanto el cliente recibe el ACK la conexión quedaría medio abierta, por ello, después de enviar el ACK el servidor envía un FIN al cliente. Una vez el cliente recibe el FIN, responde con un ACK al servidor para cerrar la otra mitad de la conexión TCP.

A medida que se producía esta comunicación, los Routers intermedios planificaban el mantenimiento de sus tablas de enrutamiento, a través del envío de RA a sus nodos vecinos.

## 2. Diseño de red IPv6

En el fichero p2.ned se define la topología de la red, en él se definen dos clientes, dos routers y dos servidores con conexiones Ethernet. En las imágenes 9 y 10 se muestra el fichero de p2.ned que define la topología de la red y la vista del diseño.

```
import ned.DatarateChannel;
import inet.nodes.ipv6.StandardHost6;
import inet.nodes.ipv6.Router6;
import inet.networklayer.autorouting.ipv6.FlatNetworkConfigurator6;

network p2
{
    types:
        channel ethernetline extends DatarateChannel
        {
            delay = 0.1us;
            datarate = 10Mbps;
        }
    submodules:
        configurator: FlatNetworkConfigurator6;
        r1: Router6;
        r2: Router6;
        cli1: StandardHost6;
        cli2: StandardHost6;
        srv1: StandardHost6;
        srv2: StandardHost6;
    connections:
        cli1.ethg++ <--> ethernetline <--> r1.ethg++;
        cli2.ethg++ <--> ethernetline <--> r1.ethg++;
        r1.ethg++ <--> ethernetline <--> r2.ethg++;
        r2.ethg++ <--> ethernetline <--> srv1.ethg++;
        r2.ethg++ <--> ethernetline <--> srv2.ethg++;
}
```

Imagen 9. p2.ned

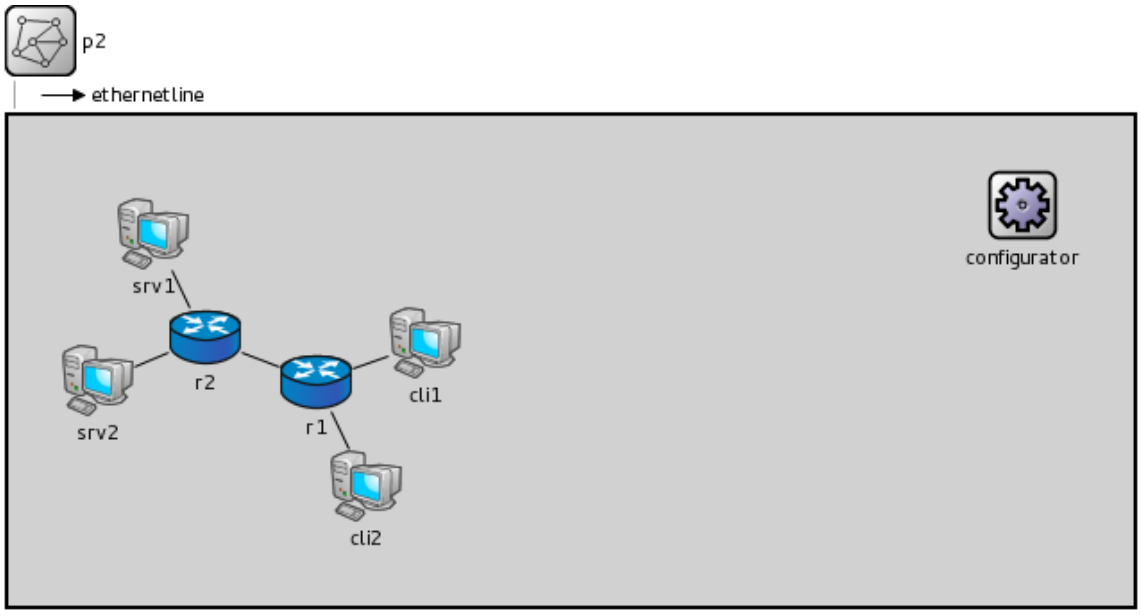


Imagen 10. Vista del diseño

En el fichero omnetpp.ini se define la configuración para la red. La configuración puede verse en la imagen 11.

```
[General]
tkenv-plugin-path = ../../inet/etc/plugins

[Config p2]
network = p2

# tcp apps
**.cli*.numTcpApps = 1
**.cli1.tcpApp[0].typename = "TelnetApp"
**.cli2.tcpApp[0].typename = "TCPBasicClientApp"
**.cli1.tcpApp[0].localAddress = "aaaa:b::8aa:ff:fe00:7" #is this the source addr of the client's TCP app?
**.cli2.tcpApp[0].localAddress = "aaaa:b::8aa:ff:fe00:8"
**.cli1.tcpApp[0].localPort = -1
**.cli2.tcpApp[0].localPort = -1
**.cli1.tcpApp[0].connectAddress = "srv1"
**.cli2.tcpApp[0].connectAddress = "srv2"
#**.cli[*].tcpApp[0].connectAddress="bbbb::"
**.cli1.tcpApp[0].connectPort = 25 #same destination port numbers
**.cli2.tcpApp[0].connectPort = 80 #same destination port numbers

**.cli1.tcpApp[0].startTime = uniform(15s,20s)
**.cli1.tcpApp[0].numCommands = exponential(10)
**.cli1.tcpApp[0].commandLength = exponential(1B)
**.cli1.tcpApp[0].keyPressDelay = exponential(0.1s)
**.cli1.tcpApp[0].commandOutputLength = exponential(40B)
**.cli1.tcpApp[0].thinkTime = truncnormal(2s,3s)
**.cli1.tcpApp[0].idleInterval = truncnormal(3600s,1200s)
**.cli1.tcpApp[0].reconnectInterval = 30s

**.cli2.tcpApp[0].startTime = uniform(10s,15s)
**.cli2.tcpApp[0].numRequestsPerSession = 1
**.cli2.tcpApp[0].requestLength = truncnormal(350B,20B)
**.cli2.tcpApp[0].replyLength = exponential(2000B)
**.cli2.tcpApp[0].thinkTime = 0s
**.cli2.tcpApp[0].idleInterval = 0s

**.srv*.numTcpApps = 1
**.srv*.tcpApp[0].typename = "TCPGenericSrvApp"
**.srv1.tcpApp[0].localAddress = ""
**.srv1.tcpApp[0].localPort = 25
**.srv1.tcpApp[0].replyDelay = 0
**.srv2.tcpApp[0].localAddress = ""  #created another one but unused for now
**.srv2.tcpApp[0].localPort = 80
**.srv2.tcpApp[0].replyDelay = 0

# tcp settings
**.tcpApp[*].dataTransferMode = "object"

# Ethernet NIC configuration
**.eth[*].queueType = "DropTailQueue" # in routers
**.eth[*].queue.dataQueue.frameCapacity = 10  # in routers

#**.eth[*].mac.txrate = 10Mbps
**.eth[*].mac.duplexMode = true
```

Imagen 11. Omnetpp.ini

Se definen dos tipos de aplicaciones TCP: TelnetApp y TCPBasicClientApp. Se asignan direcciones y puertos para estas aplicaciones en ambos clientes y se definen parámetros para configurar el comportamiento de estas aplicaciones.

Se define un tipo de aplicación servidora TCPGenericSrvApp y se asignan direcciones y puertos para ella en ambos servidores.

Finalmente, se definen los parámetros de configuración para Ethernet.

### 2.1. Comportamiento paquetes NS, NA, RS y RA

- NS: Es un paquete llamado Neighbor Solicitation (NS), la finalidad de este paquete es conocer los nodos vecinos a un nodo. Se utiliza para detectar si la red contiene otro nodo con la misma dirección IPv6 que un nodo dado, esto se conoce como Duplicate Address Detection (DAD). En este protocolo, el nodo envía un NS a la

dirección de Multicast indicando su dirección IPv6 para comprobar si alguien más en ese scope tiene la misma IPv6. En la imagen 12 se puede ver al Router 2 asignándose su dirección IP después de un DAD timeout.

```
** Event #124  t=1.138053837068  p2.r2.networkLayer.neighbourDiscovery (IPv6NeighbourDiscovery, id=56), on selfmsg `dadTimeout' (cMessage, id=86)
Self message received!
DAD Timeout message received
numOfDADMessagesSent is: 1
dupAddrDetectTrans is: 1
delete dadEntry and msg
p2.r2.routingTable6: ** Notification at T=1.138053837068 to p2.r2.routingTable6: IPv6-CFG eth2  on:nwLayer.ifOut[3]  MTU:1500 BROADCAST MULTICAST  macAddr:0A-AA-00-00-00-06 IPv6:{
p2.r2.routingTable6:  Addrs:aaaa:1:3:0:8aa:ff:fe00:6(global)  expiryTime: inf prefExpiryTime: inf
p2.r2.routingTable6:                , fe80::8aa:ff:fe00:6(link)  expiryTime: inf prefExpiryTime: inf
p2.r2.routingTable6:  mcastgrps:ff02::1,ff02::2 AdvPrefixes: aaaa:1:3::/64(R-Flag = 0 expires:2592000)
p2.r2.routingTable6:   Node: dupAddrDetectTrans=1 reachableTime=33.706910891924
p2.r2.routingTable6:  Router: maxRtrAdvInt=600 minRtrAdvInt=198
p2.r2.routingTable6:    }
p2.r2.routingTable6:
```

Imagen 12. Neighbor auto-configuration

- NA: Ante un NS, un nodo puede contestar un NA (Neighbor Advertisement), este paquete "publicita" un nodo de la red a sus nodos vecinos o le indica a otro nodo que la Ipv6 que intenta utilizar esta en uso por el (respuesta al DAD). Este tipo de paquete es utilizado en nuestro ejemplo para rellenar las tablas de enrutamiento. Cuando se va a enviar un paquete, un nodo envía un NS para comprobar si en la red algún nodo tiene la dirección del siguiente salto, en este momento si algún nodo tiene asignada esta dirección IP, éste envía un NA indicándole la dirección Ethernet a dónde debe ser enviado el paquete. En la imagen 13, se puede ver un ejemplo de este paquete.

```
** Event #287  t=12.964482681944  p2.r2.networkLayer.ipv6 (IPv6, id=53), on `NSpacket' (IPv6NeighbourSolicitation, id=483)
** Event #288  t=12.964482681944  p2.r2.eth[2].encap (EtherEncap, id=76), on `NSpacket' (IPv6Datagram, id=486)
Encapsulating higher layer packet `NSpacket' for MAC
** Event #289  t=12.964482681944  p2.r2.eth[2].queue.classifier (EtherFrameClassifier, id=77), on `NSpacket' (EthernetIIFrame, id=488)
** Event #290  t=12.964482681944  p2.r2.eth[2].queue.dataQueue (DropTailQueue, id=79), on `NSpacket' (EthernetIIFrame, id=488)
** Event #291  t=12.964482681944  p2.r2.eth[2].queue.scheduler (PriorityScheduler, id=80), on `NSpacket' (EthernetIIFrame, id=488)
** Event #292  t=12.964482681944  p2.r2.eth[2].mac (EtherMACFullDuplex, id=75), on `NSpacket' (EthernetIIFrame, id=488)
Received frame from upper layer: (EthernetIIFrame)NSpacket
Transmitting a copy of frame (EthernetIIFrame)NSpacket
Starting transmission of (EthernetIIFrame)NSpacket
** Event #293  t=12.964492181944  p2.r1.eth[2].mac (EtherMACFullDuplex, id=39), on selfmsg `EndIFG' (cMessage, id=8)
Self-message (cMessage)EndIFG received
IFG elapsed
** Event #294  t=12.964540281944  p2.r2.eth[2].mac (EtherMACFullDuplex, id=75), on selfmsg `EndTransmission' (cMessage, id=17)
Self-message (cMessage)EndTransmission received
Transmission of (EthernetIIFrame)NSpacket successfully completed
Start IFG period
** Event #295  t=12.964540381944  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on `NSpacket' (EthernetIIFrame, id=497)
Received frame from network: (EthernetIIFrame)NSpacket
** Event #296  t=12.964540381944  p2.srv2.eth[0].encap (EtherEncap, id=164), on `NSpacket' (EthernetIIFrame, id=497)
Decapsulating frame `NSpacket', passing up contained packet `NSpacket' to higher layer
** Event #297  t=12.964540381944  p2.srv2.networkLayer.ipv6 (IPv6, id=155), on `NSpacket' (IPv6Datagram, id=501)
destination address ff02::1:ff00:a is multicast, doing multicast routing
local delivery of multicast packet
0 extension header(s) for processing...
Neigbour Discovery packet: passing it to ND module
forwarding is off
** Event #298  t=12.964540381944  p2.srv2.networkLayer.neighbourDiscovery (IPv6NeighbourDiscovery, id=158), on `NSpacket' (IPv6NeighbourSolicitation, id=504)
Process NS for Non-Tentative target address.
Neighbour Entry not found. Create a Neighbour Cache Entry.
** Event #299  t=12.964540381944  p2.srv2.networkLayer.ipv6 (IPv6, id=155), on `NApacket' (IPv6NeighbourAdvertisement, id=506)
** Event #300  t=12.964540381944  p2.srv2.eth[0].encap (EtherEncap, id=164), on `NApacket' (IPv6Datagram, id=508)
Encapsulating higher layer packet `NApacket' for MAC
** Event #301  t=12.964540381944  p2.srv2.eth[0].queue.classifier (EtherFrameClassifier, id=165), on `NApacket' (EthernetIIFrame, id=510)
** Event #302  t=12.964540381944  p2.srv2.eth[0].queue.dataQueue (DropTailQueue, id=167), on `NApacket' (EthernetIIFrame, id=510)
** Event #303  t=12.964540381944  p2.srv2.eth[0].queue.scheduler (PriorityScheduler, id=168), on `NApacket' (EthernetIIFrame, id=510)
** Event #304  t=12.964540381944  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on `NApacket' (EthernetIIFrame, id=510)
Received frame from upper layer: (EthernetIIFrame)NApacket
Transmitting a copy of frame (EthernetIIFrame)NApacket
Starting transmission of (EthernetIIFrame)NApacket
```

Imagen 13. NS and NA example

- RS: Un nodo puede solicitar a la red información acerca del Router que tiene por defecto. Para ello, envía por la dirección de Multicast un Router Solicitation (RS) para intentar conocer el prefijo de red y poder auto configurar una dirección IPv6 única. Esto se puede ver en la imagen 14.

```
** Event #146  t=2.721406817296  p2.srv2.networkLayer.ipv6 (IPv6, id=155), on `RSpacket' (IPv6RouterSolicitation, id=252)
** Event #147  t=2.721406817296  p2.srv2.eth[0].encap (EtherEncap, id=164), on `RSpacket' (IPv6Datagram, id=255)
Encapsulating higher layer packet `RSpacket' for MAC
** Event #148  t=2.721406817296  p2.srv2.eth[0].queue.classifier (EtherFrameClassifier, id=165), on `RSpacket' (EthernetIIFrame, id=257)
** Event #149  t=2.721406817296  p2.srv2.eth[0].queue.dataQueue (DropTailQueue, id=167), on `RSpacket' (EthernetIIFrame, id=257)
** Event #150  t=2.721406817296  p2.srv2.eth[0].queue.scheduler (PriorityScheduler, id=168), on `RSpacket' (EthernetIIFrame, id=257)
** Event #151  t=2.721406817296  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on `RSpacket' (EthernetIIFrame, id=257)
Received frame from upper layer: (EthernetIIFrame)RSpacket
Transmitting a copy of frame (EthernetIIFrame)RSpacket
Starting transmission of (EthernetIIFrame)RSpacket
** Event #152  t=2.721464417296  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on selfmsg `EndTransmission' (cMessage, id=33)
Self-message (cMessage)EndTransmission received
Transmission of (EthernetIIFrame)RSpacket successfully completed
Start IFG period
** Event #153  t=2.721464517296  p2.r2.eth[2].mac (EtherMACFullDuplex, id=75), on `RSpacket' (EthernetIIFrame, id=262)
Received frame from network: (EthernetIIFrame)RSpacket
** Event #154  t=2.721464517296  p2.r2.eth[2].encap (EtherEncap, id=76), on `RSpacket' (EthernetIIFrame, id=262)
Decapsulating frame `RSpacket', passing up contained packet `RSpacket' to higher layer
** Event #155  t=2.721464517296  p2.r2.networkLayer.ipv6 (IPv6, id=53), on `RSpacket' (IPv6Datagram, id=265)
destination address ff02::2 is multicast, doing multicast routing
local delivery of multicast packet
0 extension header(s) for processing...
Neigbour Discovery packet: passing it to ND module
multicast dest address is link-local (or smaller) scope
** Event #156  t=2.721464517296  p2.r2.networkLayer.neighbourDiscovery (IPv6NeighbourDiscovery, id=56), on `RSpacket' (IPv6RouterSolicitation, id=268)
This is an advertising interface, processing RS
RS message validated
MAC Address extracted
** Event #157  t=2.721474017296  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on selfmsg `EndIFG' (cMessage, id=34)
Self-message (cMessage)EndIFG received
IFG elapsed
** Event #158  t=2.771659406553  p2.cli2.networkLayer.neighbourDiscovery (IPv6NeighbourDiscovery, id=114), on selfmsg `dadTimeout' (cMessage, id=203)
Self message received!
DAD Timeout message received
numOfDADMessagesSent is: 1
dupAddrDetectTrans is: 1
delete dadEntry and msg
creating router discovery message timer
** Event #159  t=2.825445231003  p2.cli1.networkLayer.neighbourDiscovery (IPv6NeighbourDiscovery, id=92), on selfmsg `dadTimeout' (cMessage, id=218)
Self message received!
DAD Timeout message received
numOfDADMessagesSent is: 1
dupAddrDetectTrans is: 1
delete dadEntry and msg
creating router discovery message timer
** Event #160  t=3.094023414051  p2.r2.networkLayer.neighbourDiscovery (IPv6NeighbourDiscovery, id=56), on selfmsg `sendSolicitedRA' (cMessage, id=251)
Self message received!
Sending solicited RA
Send Solicited RA invoked!
Testing condition!
Create and send RA invoked!
Number of Adv Prefixes: 1

+=+=+=+= Appendign Prefix Info Option to RA +=+=+=+=
Prefix Vaue: aaaa:1:2::
Prefix Length: 64
L-Flag: 1
A-Flag: 1
R-Flag: 0
Global Address from Prefix: aaaa:1:2:0:8aa:ff:fe00:5
```

Imagen 14. Ejemplo RS y RA

- RA: Un router responde a un RS con un Router Advertisement (RA), en este paquete, el router le envía la información al nodo para indicarle el prefijo de red y alguna información de la red. El paquete puede ser observado en la imagen 14.

### 2.2. Comportamiento paquetes

- SYN: El paquete SYN es enviado por cada uno de los clientes a través de su interfaz eth0 hacia el router que tiene conectado. El router lo reenvía hacia su destino. Este paquete inicia una conexión (inicia lo que se conoce como three-way handshake negotiation). Una vez el paquete llega al servidor de destino, este servidor sabe que hay un cliente que quiere conectarse con él y le contesta con un paquete con SYN-ACK.

- SYN-ACK: Un paquete con estos flags activados es enviado por un servidor hacia el cliente correspondiente tras haber recibido un paquete con el flag SYN. El ACK representa el acknowledge de que el servidor ha recibido el SYN del cliente, el SYN indica que el servidor también quiere conectarse con el cliente. En la imagen 15, se puede observar al servidor srv2 recibiendo un paquete SYN y enviando el SYN-ACK correspondiente.

```
** Event #32625  t=13.310857581574  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on `SYN' (EthernetIIFrame, id=15547)
Received frame from network: (EthernetIIFrame)SYN
** Event #32626  t=13.310857581574  p2.srv2.eth[0].encap (EtherEncap, id=164), on `SYN' (EthernetIIFrame, id=15547)
Decapsulating frame `SYN', passing up contained packet `SYN' to higher layer
** Event #32627  t=13.310857581574  p2.srv2.networkLayer.ipv6 (IPv6, id=155), on `SYN' (IPv6Datagram, id=15550)
Routing datagram `SYN' with dest=aaaa:1:3:0:8aa:ff:fe00:a:
local delivery
0 extension header(s) for processing...
Protocol 6, passing up on gate 0
** Event #32628  t=13.310857581574  p2.srv2.tcp (TCP, id=154), on `SYN' (TCPSegment, id=15552)
Connection <unspec>:80 to <unspec>:-1  on app[0], connId=3  in LISTEN
Seg arrived: .1079 > .80: SYN [3327640..3327640) (l=0) win 7504 options MSS
TCB: snd una=0 snd nxt=0 snd max=0 snd wnd=0 rcv nxt=0 rcv wnd=0 snd cwnd=0 rto=3 ssthresh=65535
SYN bit set: filling in foreign socket and sending SYN+ACK
Connection forked: this connection got new connId=113, spinoff keeps LISTENing with connId=3
Updating send window from segment: new wnd=7504
TCP Header Option(s) received:
Option type 2 (MSS), length 4
TCP Header Option MSS(=536) received, SMSS is set to 536
TCP Header Option MSS(=536) sent
Sending: .80 > .1079: SYN+ACK [3327714..3327714) (l=0) ack 3327641 win 7504 options MSS
Transition: LISTEN --> SYN_RCVD  (event was: RCV SYN)
** Event #32629  t=13.310857581574  p2.srv2.networkLayer.ipv6 (IPv6, id=155), on `SYN+ACK' (TCPSegment, id=15562)
Routing datagram `SYN+ACK' with dest=aaaa:0:2:0:8aa:ff:fe00:8:
Looking up tunnels...found vIf=-1
next hop for aaaa:0:2:0:8aa:ff:fe00:8 is fe80::8aa:ff:fe00:6, interface eth0
p2.srv2.networkLayer.neighbourDiscovery: NUD in progress.
link-layer address: 0A-AA-00-00-00-06
** Event #32630  t=13.310857581574  p2.srv2.eth[0].encap (EtherEncap, id=164), on `SYN+ACK' (IPv6Datagram, id=15565)
Encapsulating higher layer packet `SYN+ACK' for MAC
** Event #32631  t=13.310857581574  p2.srv2.eth[0].queue.classifier (EtherFrameClassifier, id=165), on `SYN+ACK' (EthernetIIFrame, id=15567)
** Event #32632  t=13.310857581574  p2.srv2.eth[0].queue.dataQueue (DropTailQueue, id=167), on `SYN+ACK' (EthernetIIFrame, id=15567)
** Event #32633  t=13.310857581574  p2.srv2.eth[0].queue.scheduler (PriorityScheduler, id=168), on `SYN+ACK' (EthernetIIFrame, id=15567)
** Event #32634  t=13.310857581574  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on `SYN+ACK' (EthernetIIFrame, id=15567)
Received frame from upper layer: (EthernetIIFrame)SYN+ACK
Transmitting a copy of frame (EthernetIIFrame)SYN+ACK
Starting transmission of (EthernetIIFrame)SYN+ACK
** Event #32635  t=13.310867081574  p2.r2.eth[2].mac (EtherMACFullDuplex, id=75), on selfmsg `EndIFG' (cMessage, id=18)
Self-message (cMessage)EndIFG received
IFG elapsed
** Event #32636  t=13.310929581574  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on selfmsg `EndTransmission' (cMessage, id=33)
Self-message (cMessage)EndTransmission received
Transmission of (EthernetIIFrame)SYN+ACK successfully completed
Start IFG period
** Event #32637  t=13.310929681574  p2.r2.eth[2].mac (EtherMACFullDuplex, id=75), on `SYN+ACK' (EthernetIIFrame, id=15572)
Received frame from network: (EthernetIIFrame)SYN+ACK
** Event #32638  t=13.310929681574  p2.r2.eth[2].encap (EtherEncap, id=76), on `SYN+ACK' (EthernetIIFrame, id=15572)
Decapsulating frame `SYN+ACK', passing up contained packet `SYN+ACK' to higher layer
** Event #32639  t=13.310929681574  p2.r2.networkLayer.ipv6 (IPv6, id=53), on `SYN+ACK' (IPv6Datagram, id=15575)
Routing datagram `SYN+ACK' with dest=aaaa:0:2:0:8aa:ff:fe00:8:
Looking up tunnels...found vIf=-1
```

Imagen 15. Ejemplo SYN y SYN-ACK

- ACK: Es el cliente el que envía este paquete tras recibir un SYN-ACK por parte del servidor. El cliente le manda un acklowledge para indicar que ha recubido el SYN del servidor. En la imagen 16, se puede observar al cliente cli2 recibiendo un SYN-ACK y enviando el ACK correspondiente.

```
** Event #32657  t=13.311073881574  p2.cli2.eth[0].mac (EtherMACFullDuplex, id=119), on `SYN+ACK' (EthernetIIFrame, id=15594)
Received frame from network: (EthernetIIFrame)SYN+ACK
** Event #32658  t=13.311073881574  p2.cli2.eth[0].encap (EtherEncap, id=120), on `SYN+ACK' (EthernetIIFrame, id=15594)
Decapsulating frame `SYN+ACK', passing up contained packet `SYN+ACK' to higher layer
** Event #32659  t=13.311073881574  p2.cli2.networkLayer.ipv6 (IPv6, id=111), on `SYN+ACK' (IPv6Datagram, id=15597)
Routing datagram `SYN+ACK' with dest=aaaa:0:2:0:8aa:ff:fe00:8:
local delivery
0 extension header(s) for processing...
Protocol 6, passing up on gate 0
** Event #32660  t=13.311073881574  p2.cli2.tcp (TCP, id=110), on `SYN+ACK' (TCPSegment, id=15599)
Connection <unspec>:1079 to aaaa:1:3:0:8aa:ff:fe00:a:80  on app[0], connId=112  in SYN_SENT
Seg arrived: .80 > .1079: SYN+ACK [3327714..3327714) (l=0) ack 3327641 win 7504 options MSS
TCB: snd una=3327640 snd nxt=3327641 snd max=3327641 snd wnd=0 rcv nxt=0 rcv wnd=7504 snd cwnd=0 rto=3 ssthresh=65535
ACK bit set, AckNo acceptable
Updating send window from segment: new wnd=7504
SYN+ACK bits set, connection established.
TCP Header Option(s) received:
Option type 2 (MSS), length 4
TCP Header Option MSS(=536) received, SMSS is set to 536
Completing connection setup by sending ACK (possibly piggybacked on data)
Sending: .1079 > .80: ack 3327715 win 7504
Notifying app: ESTABLISHED
Transition: SYN_SENT --> ESTABLISHED  (event was: RCV SYN_ACK)
** Event #32661  t=13.311073881574  p2.cli2.networkLayer.ipv6 (IPv6, id=111), on `ACK' (TCPSegment, id=15601)
Routing datagram `ACK' with dest=aaaa:1:3:0:8aa:ff:fe00:a:
Looking up tunnels...found vIf=-1
next hop for aaaa:1:3:0:8aa:ff:fe00:a is fe80::8aa:ff:fe00:2, interface eth0
p2.cli2.networkLayer.neighbourDiscovery: NUD in progress.
link-layer address: 0A-AA-00-00-00-02
** Event #32662  t=13.311073881574  p2.cli2.tcpApp[0] (TCPBasicClientApp, id=109), on `ESTABLISHED' (cMessage, id=15604)
connected
sending request, 0 more to go
sending 333 bytes, expecting 504
** Event #32663  t=13.311073881574  p2.cli2.eth[0].encap (EtherEncap, id=120), on `ACK' (IPv6Datagram, id=15606)
Encapsulating higher layer packet `ACK' for MAC
** Event #32664  t=13.311073881574  p2.cli2.tcp (TCP, id=110), on `data' (GenericAppMsg, id=15608)
Connection aaaa:0:2:0:8aa:ff:fe00:8:1079 to aaaa:1:3:0:8aa:ff:fe00:a:80  on app[0], connId=112  in ESTABLISHED
App command: SEND
333 bytes in queue, plus 0 bytes unacknowledged
Restarting idle connection, CWND is set to 536
Will send 333 bytes (effectiveWindow 536, in buffer 333 bytes)
Sending: .1079 > .80: [3327641..3327974) (l=333) ack 3327715 win 7504
Starting REXMIT timer
Starting rtt measurement on seq=3327641
Staying in state: ESTABLISHED (event was: SEND)
** Event #32665  t=13.311073881574  p2.cli2.eth[0].queue.classifier (EtherFrameClassifier, id=121), on `ACK' (EthernetIIFrame, id=15610)
** Event #32666  t=13.311073881574  p2.cli2.networkLayer.ipv6 (IPv6, id=111), on `data(l=333,1msg)' (TCPSegment, id=15614)
Routing datagram `data(l=333,1msg)' with dest=aaaa:1:3:0:8aa:ff:fe00:a:
Looking up tunnels...found vIf=-1
next hop for aaaa:1:3:0:8aa:ff:fe00:a is fe80::8aa:ff:fe00:2, interface eth0
p2.cli2.networkLayer.neighbourDiscovery: NUD in progress.
link-layer address: 0A-AA-00-00-00-02
** Event #32667  t=13.311073881574  p2.cli2.eth[0].queue.dataQueue (DropTailQueue, id=123), on `ACK' (EthernetIIFrame, id=15610)
** Event #32668  t=13.311073881574  p2.cli2.eth[0].encap (EtherEncap, id=120), on `data(l=333,1msg)' (IPv6Datagram, id=15619)
Encapsulating higher layer packet `data(l=333,1msg)' for MAC
** Event #32669  t=13.311073881574  p2.cli2.eth[0].queue.scheduler (PriorityScheduler, id=124), on `ACK' (EthernetIIFrame, id=15610)
** Event #32670  t=13.311073881574  p2.cli2.eth[0].queue.classifier (EtherFrameClassifier, id=121), on `data(l=333,1msg)' (EthernetIIFrame, id=15622)
** Event #32671  t=13.311073881574  p2.cli2.eth[0].mac (EtherMACFullDuplex, id=119), on `ACK' (EthernetIIFrame, id=15610)
Received frame from upper layer: (EthernetIIFrame)ACK
Transmitting a copy of frame (EthernetIIFrame)ACK
Starting transmission of (EthernetIIFrame)ACK
```

Imagen 16. SYN-ACK y ACK

- FIN: El FIN es mandado por el cliente hacia el cliente para indicarle que quiere terminar la conexión y este puede liberar los recursos.

- ACK: Al FIN del cliente, el servidor contesta con un ACK hacia el cliente para indicarle que recibió su fin y se va a proceder a finalizar la conexión. En la imagen 17 se puede ver al srv2 recibiendo un FIN y devolviendo el correspondiente ACK.

```
** Event #32856  t=13.313937981574  p2.srv2.eth[0].mac (EtherMACFullDuplex, id=163), on `FIN' (EthernetIIFrame, id=15850)
Received frame from network: (EthernetIIFrame)FIN
** Event #32857  t=13.313937981574  p2.srv2.eth[0].encap (EtherEncap, id=164), on `FIN' (EthernetIIFrame, id=15850)
Decapsulating frame `FIN', passing up contained packet `FIN' to higher layer
** Event #32858  t=13.313937981574  p2.srv2.networkLayer.ipv6 (IPv6, id=155), on `FIN' (IPv6Datagram, id=15853)
Routing datagram `FIN' with dest=aaaa:1:3:0:8aa:ff:fe00:a:
local delivery
0 extension header(s) for processing...
Protocol 6, passing up on gate 0
** Event #32859  t=13.313937981574  p2.srv2.tcp (TCP, id=154), on `FIN' (TCPSegment, id=15855)
Connection aaaa:1:3:0:8aa:ff:fe00:a:80 to aaaa:0:2:0:8aa:ff:fe00:8:1079  on app[0], connId=113  in ESTABLISHED
Seg arrived: .1079 > .80: FIN(+ACK) ack 3328219 win 7504
TCB: snd una=3328219 snd nxt=3328219 snd max=3328219 snd wnd=7504 rcv nxt=3327974 rcv wnd=7504 snd cwnd=1072 rto=2.626063375 ssthresh=65535
ACK looks duplicate but we have currently no unacked data (snd una == snd max)
FIN arrived, advancing rcv nxt over the FIN
rcv nxt changed to 3327975, (delayed ACK disabled) sending ACK now
Sending: .80 > .1079: ack 3327975 win 7504
Transition: ESTABLISHED --> CLOSE WAIT  (event was: RCV FIN)
Notifying app: PEER CLOSED
** Event #32860  t=13.313937981574  p2.srv2.networkLayer.ipv6 (IPv6, id=155), on `ACK' (TCPSegment, id=15857)
Routing datagram `ACK' with dest=aaaa:0:2:0:8aa:ff:fe00:8:
Looking up tunnels...found vIf=-1
next hop for aaaa:0:2:0:8aa:ff:fe00:8 is fe80::8aa:ff:fe00:6, interface eth0
p2.srv2.networkLayer.neighbourDiscovery: NUD in progress.
link-layer address: 0A-AA-00-00-00-06
```

Imagen 17. Ejemplo FIN y ACK

### 2.3. Paquetes SYN con inicio cliente telnet 0-1s.

Entre los segundos 0 y 1 es posible que las direcciones de Scope Global no estén configuradas, por tanto cuando el cliente intenta iniciar la conexión TCP con el servidor no puede alcanzarlo y añade a su tabla de enrutamiento la dirección link-local del router como siguiente salto hacia el servidor. Esto se puede observar en la imagen 18.

Como consecuencia de esto, cuando pasa el tiempo de reenvío, el cliente vuelve a intentar enviar el SYN pero en este momento, al tener el router ya configurada su IP de Scope global, obtiene un Destination Unreachable. Esto se puede contemplar en la imagen 19.

Una vez que el cliente configura su dirección IP de Scope Global, cuando recibe el Router Advertisement la dirección del router se actualiza a su nueva dirección de Scope Global y a partir de ahí ya podría enviar el paquete.

```
** Event #90  t=0.548813502304  p2.cli1.networkLayer.ipv6 (IPv6, id=89), on `SYN' (TCPSegment, id=231)
Routing datagram `SYN' with dest=fe80::8aa:ff:fe00:9:
Looking up tunnels...found vIf=-1
do longest prefix match in routing table
finished longest prefix match in routing table
next hop for fe80::8aa:ff:fe00:9 is fe80::8aa:ff:fe00:9, interface eth0
no link-layer address for next hop yet, passing datagram to Neighbour Discovery module
** Event #91  t=0.548813502304  p2.cli1.networkLayer.neighbourDiscovery (IPv6NeighbourDiscovery, id=92), on `SYN' (IPv6Datagram, id=234)
Packet (IPv6Datagram)SYN arrived from IPv6 module.
Determining Next Hop
Find out if supplied dest addr is on-link or off-link.
Dest is on-link, next-hop addr is same as dest addr.
Next Hop Address is: fe80::8aa:ff:fe00:9 on interface: 101
No Entry exists in the Neighbour Cache.
Creating an INCOMPLETE entry in the neighbour cache.
Initiating Address Resolution for:fe80::8aa:ff:fe00:9 on Interface:101
Preparing to send NS to solicited-node multicast group
on the next hop interface
Reachability State is INCOMPLETE. Address Resolution already initiated.
Add packet to entry's queue until Address Resolution is complete.
** Event #92  t=0.548813502304  p2.cli1.networkLayer.ipv6 (IPv6, id=89), on `NSpacket' (IPv6NeighbourSolicitation, id=236)
** Event #93  t=0.548813502304  p2.cli1.eth[0].encap (EtherEncap, id=98), on `NSpacket' (IPv6Datagram, id=239)
Encapsulating higher layer packet `NSpacket' for MAC
** Event #94  t=0.548813502304  p2.cli1.eth[0].queue.classifier (EtherFrameClassifier, id=99), on `NSpacket' (EthernetIIFrame, id=241)
** Event #95  t=0.548813502304  p2.cli1.eth[0].queue.dataQueue (DropTailQueue, id=101), on `NSpacket' (EthernetIIFrame, id=241)
** Event #96  t=0.548813502304  p2.cli1.eth[0].queue.scheduler (PriorityScheduler, id=102), on `NSpacket' (EthernetIIFrame, id=241)
** Event #97  t=0.548813502304  p2.cli1.eth[0].mac (EtherMACFullDuplex, id=97), on `NSpacket' (EthernetIIFrame, id=241)
Received frame from upper layer: (EthernetIIFrame)NSpacket
Transmitting a copy of frame (EthernetIIFrame)NSpacket
Starting transmission of (EthernetIIFrame)NSpacket
** Event #98  t=0.548871102304  p2.cli1.eth[0].mac (EtherMACFullDuplex, id=97), on selfmsg `EndTransmission' (cMessage, id=21)
Self-message (cMessage)EndTransmission received
Transmission of (EthernetIIFrame)NSpacket successfully completed
Start IFG period
```

Imagen 18. SYN servidor

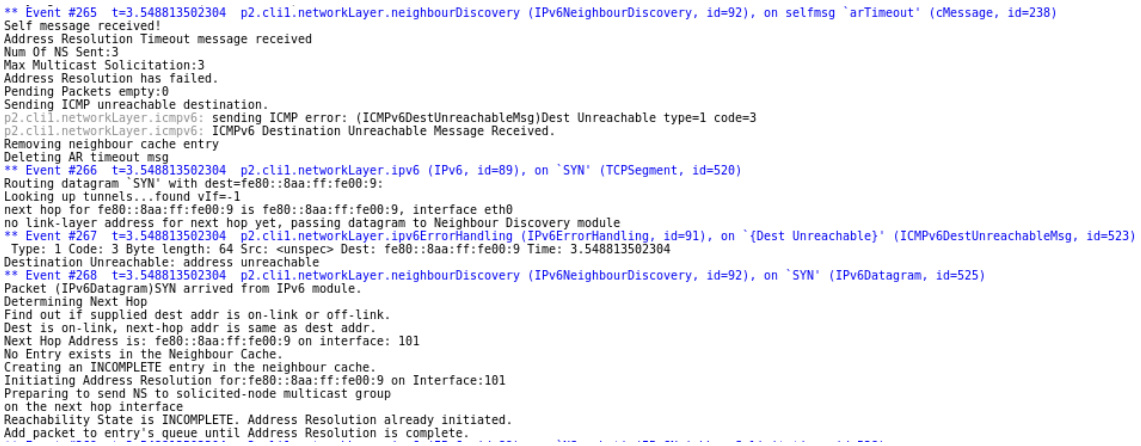


Imagen 19. Destination Unreachable

# Práctica 3. Nodos móviles

### 1. Red IPv6 con nodos inalámbricos.

Basándose en el ejemplo de IPv6 se simula una red IPv6 con nodos inalámbricos. La red generada consta de los siguientes elementos:

Un punto de acceso (PA) conectado a un Home Agent (HA) a través de una conexión Ethernet.

Un Router intermedio conectado al Home Agent anterior a través de una conexión Ethernet.

Un servidor Telnet conectado al Router anterior a través de una conexión Ethernet.

Un nodo móvil, que está conectado inalámbricamente al Punto de acceso.

El funcionamiento de la red es el siguiente:

El nódo móvil se comunica con su HA a través del punto de acceso al que está asociado (si no está muy lejos y puede asociarse a él). El HA enruta los paquetes entre el nodo móvil y el Router al que se conecta el servidor telnet que es con quién quiere conectarse el nodo móvil, por tanto, los paquetes destinados al Servidor telnet son enrutados a través del HA hacia el Router intermedio. Para las respuestas, los datos viajan en sentido contrario.

Para que el nodo móvil pueda tener constancia de si está conectado al PA, este envía Beacons (paquetes baliza) para comprobar que el nodo móvil sigue conectado. Al mismo tiempo, el HA envía NA para refrescar la conexión con el nodo.

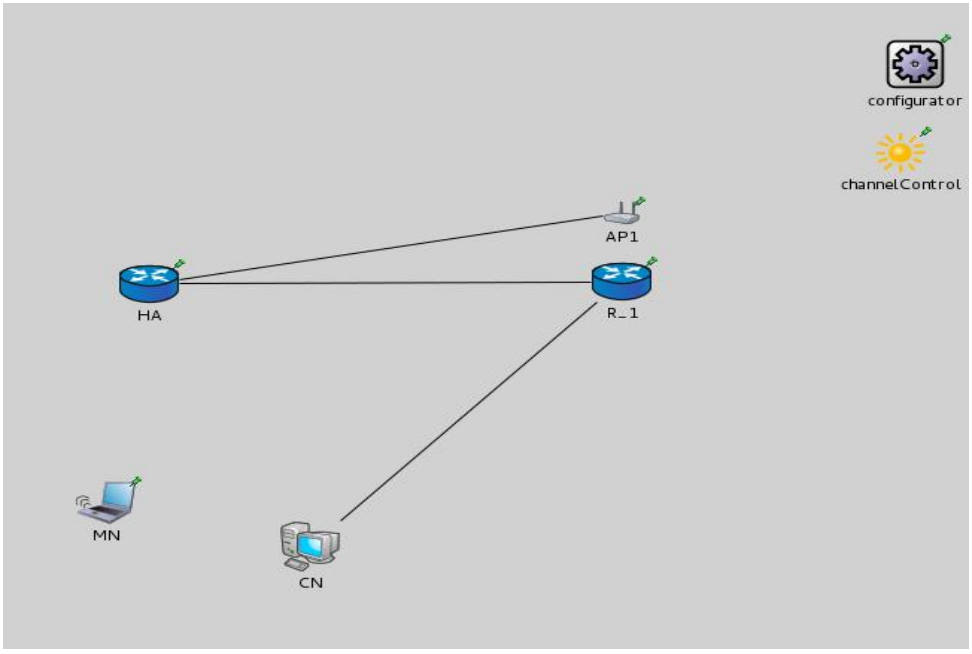


Imagen 20. La red generada

```
[General]
num-rngs = 3
seed-set = 1
**.gen[*].rng-0 = 1
**.mobility.rng-0 = 2

debug-on-errors = false

network = p3

cmdenv-express-mode = true

tkenv-plugin-path = ../../inet/etc/plugins

**.neighbourDiscovery.minIntervalBetweenRAs = 0.03s #MinRtrAdvInterval (RFC 3775),applicable when MIPv6Support is true
**.neighbourDiscovery.maxIntervalBetweenRAs = 0.07s #3 ;MaxRtrAdvInterval (RFC 3775),applicable when MIPv6Support is true

# channel physical parameters
*.channelControl.carrierFrequency = 2.4GHz
*.channelControl.pMax = 2.0mW
*.channelControl.sat = -82dBm
*.channelControl.alpha = 2

**.wlan*.mgmt.numAuthSteps = 4
**.mgmt.frameCapacity = 10

# ALL APs common parameters
**.AP*.wlan*.mgmt.beaconInterval = 0.1s

# Access Point AP_Home ; AP_1 ; AP_2 ; AP_3 Parameters for EtherMAC
**.AP1.wlan*.mgmt.ssid = "HOME"
**.AP1.wlan*.mac.address = "10:AA:00:00:00:01"
**.AP1.eth[0].*.scalar-recording = false

# mobility
**.MN.mobilityType = "RectangleMobility"
**.MN.mobility.constraintAreaMinX = 180m
**.MN.mobility.constraintAreaMinY = 100m
**.MN.mobility.constraintAreaMaxX = 530m
**.MN.mobility.constraintAreaMaxY = 110m
**.MN.mobility.startPos = 10
**.MN.mobility.speed = 1mps
**.MN.mobility.updateInterval = 0.1s

# tcp apps
**.MN.numTcpApps = 1                  #changed from 1 to 0
**.MN.tcpApp[0].typename = "TelnetApp"
**.MN.tcpApp[0].localAddress = "aaaa:b::8aa:ff:fe00:7"#is this the source addr of the client's TCP app?
**.MN.tcpApp[0].localPort = -1
**.MN.tcpApp[0].connectAddress = "CN"
**.MN.tcpApp[0].connectPort = 1000 #same destination port numbers
```

Imagen 21. Código omnetpp.ini

```
**.MN.tcpApp[0].startTime = uniform(10s,15s)
**.MN.tcpApp[0].numCommands = 500
**.MN.tcpApp[0].commandLength = exponential(1B)
**.MN.tcpApp[0].keyPressDelay = exponential(0.1s)
**.MN.tcpApp[0].commandOutputLength = exponential(40B)
**.MN.tcpApp[0].thinkTime = truncnormal(2s,3s)
**.MN.tcpApp[0].idleInterval = truncnormal(3600s,1200s)
**.MN.tcpApp[0].reconnectInterval = 30s

**.tcpApp[*].dataTransferMode = "object"

**.CN.numTcpApps = 1                        #changed from 1 to 0
**.CN.tcpApp[0].typename = "TCPGenericSrvApp"
**.CN.tcpApp[0].localAddress = ""
**.CN.tcpApp[0].localPort = 1000
**.CN.tcpApp[0].replyDelay = 0


**.ipv6.procDelay = 10us
**.IPForward = false
#**.routingFile = xmldoc("empty.xml")


# Ethernet NIC configuration
**.eth[*].queueType = "DropTailQueue" # in routers
**.eth[*].encap.*.scalar-recording = false
**.eth[*].mac.promiscuous = false
**.eth[*].mac.address = "auto"
```

Imagen 22. Código omnetpp.ini 2

```
##################################
**.eth*.mac.duplexMode = true
**.eth*.mac.*.scalar-recording = false

**.ap.*.scalar-recording = false
**.hub.*.scalar-recording = false

# wireless channels
**.AP1.wlan*.radio.channelNumber = 1
**.MN.wlan*.radio.channelNumber = 0  # just initially -- it'll scan

# wireless configuration
**.wlan*.agent.activeScan = true
**.wlan*.agent.channelsToScan = "1 2"  # "" means all
**.wlan*.agent.probeDelay = 0.1s
**.wlan*.agent.minChannelTime = 0.15s
**.wlan*.agent.maxChannelTime = 0.3s
**.wlan*.agent.authenticationTimeout = 5s
**.wlan*.agent.associationTimeout = 5s

# nic settings
**.wlan*.bitrate = 2Mbps

**.mac.address = "auto"
**.mac.maxQueueSize = 14
**.mac.rtsThresholdBytes = 4000B
**.wlan*.mac.retryLimit = 7
**.wlan*.mac.cwMinData = 7

**.radio.transmitterPower = 2.0mW
**.radio.carrierFrequency = 2.4GHz
**.radio.thermalNoise = -110dBm
**.radio.sensitivity = -82dBm
**.radio.pathLossAlpha = 2
**.radio.snirThreshold = 4dB

**.coreDebug = false

**.constraintAreaMinX = 0m
**.constraintAreaMinY = 0m
**.constraintAreaMaxX = 850m
**.constraintAreaMaxY = 850m
```

Imagen 23. Código omnetpp.ini 2

```
import inet.nodes.xmipv6.WirelessHost6;
import inet.nodes.xmipv6.HomeAgent6;
import inet.nodes.xmipv6.CorrespondentNode6;
import inet.nodes.wireless.AccessPoint;
import inet.nodes.ipv6.Router6;
import inet.networklayer.autorouting.ipv6.FlatNetworkConfigurator6;
import inet.linklayer.ethernet.EtherHub;


channel ethernetline extends ned.DatarateChannel
{
    parameters:
        delay = 0.1us;
        datarate = 100Mbps;
}

network p3
{
    parameters:

        @display("bgb=799,698");
    submodules:
        configurator: FlatNetworkConfigurator6 {
            parameters:
                @display("p=763,53");
        }
        channelControl: ChannelControl {
            parameters:
                numChannels = 5;
                @display("p=753,123");
        }
        HA: HomeAgent6 {
            parameters:
                @display("p=249,229;i=abstract/router");
        }
        R_1: Router6 {
            parameters:
                @display("p=566,227");
        }

        MN: WirelessHost6 {
            parameters:
                @display("p=220,404");
        }
        CN: CorrespondentNode6;
        AP1: AccessPoint {
            parameters:
                @display("p=566,172;i=device/accesspoint_s");
        }
    connections allowunconnected:
        AP1.ethg++ <--> ethernetline <--> HA.ethg++;
        HA.ethg++ <--> ethernetline <--> R_1.ethg++;
        R_1.ethg++ <--> ethernetline <--> CN.ethg++;

}
```

Imagen 24. p3.ned

## 2. Analizar impacto de las variables en la red

- Pontencias de transmisión: Radio.transmitterPower

  Su valor por defecto es 2.0mW, el simulador no permite que este valor supere a pMax del canal físico que está fijado a 2.0mW. Entonces, vamos a analizar qué pasa con el defectivo y reduciendo su valor:

  Al disminuir la potencia (se disminuye el valor de Radio.transmitterPower), se disminuye el radio de transmisión, por lo tanto el MN y el AP pasan más tiempo sin poder comunicarse, lo cual implica que la conexión TCP deja de enviar paquetes. Una vez el MN se vuelve a asociar con el AP, éste puede volver a enviar paquetes.

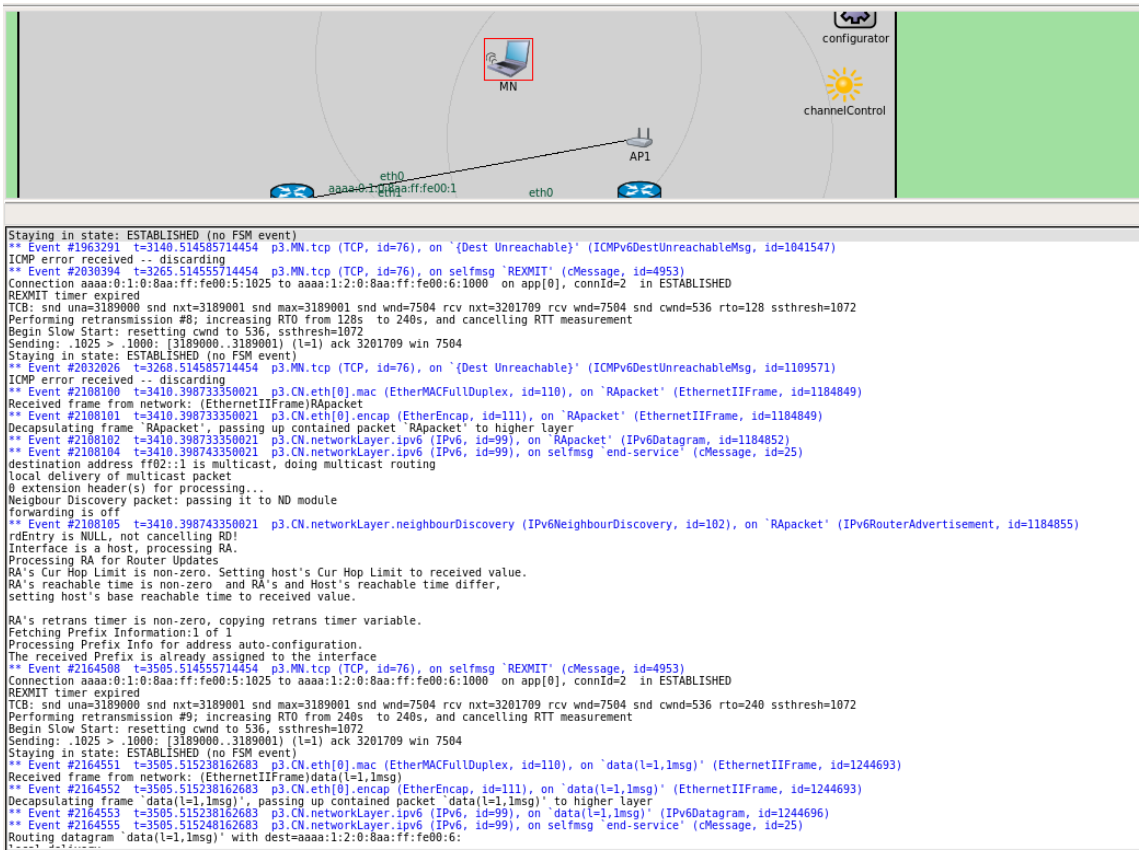


Imagen 25. TCP reconectando

- Ruido: Radio.thermalNoise:
  A medida que se aumenta el ruido (valores menos negativos) se produce más pérdida de paquetes en la transmisión inalámbrica, pudiendo provocar desconexiones en TCP como se ve en la imagen 26.

```
** Event #27816  t=87.721757464096  p3.MN.wlan[0].radio (Ieee80211Radio, id=90), on selfmsg `{}' (cMessage, id=15489)
Radio::handleSelfMsg1
transmission over but noise level too high, switch to recv mode (state:RECV)
p3.MN.wlan[0].mac: ** Notification at T=87.721757464096 to p3.MN.wlan[0].mac: RADIO-STATE RECV, channel #2, 2Mbps
p3.MN.wlan[0].mac: # state information: mode = DCF, state = DEFER, backoff 0..1 = 1
p3.MN.wlan[0].mac: # backoffPeriod 0..1 = -1
p3.MN.wlan[0].mac: # retryCounter 0..1 = 0, radioState = 1, nav = 0, txop is 0
p3.MN.wlan[0].mac: #queue size 0..1 = 0, medium is busy, scheduled AIFS are 0(), scheduled backoff are 0()
p3.MN.wlan[0].mac: # currentAC: 0, oldcurrentAC: 0
p3.MN.wlan[0].mac: # current transmission: none
p3.MN.wlan[0].mac: processing event in state machine Ieee80211Mac State Machine
p3.MN.wlan[0].mac: leaving handleWithFSM
p3.MN.wlan[0].mac:  # state information: mode = DCF, state = DEFER, backoff 0..1 = 1
p3.MN.wlan[0].mac: # backoffPeriod 0..1 = -1
p3.MN.wlan[0].mac: # retryCounter 0..1 = 0, radioState = 1, nav = 0, txop is 0
p3.MN.wlan[0].mac: #queue size 0..1 = 0, medium is busy, scheduled AIFS are 0(), scheduled backoff are 0()
p3.MN.wlan[0].mac: # currentAC: 0, oldcurrentAC: 0
p3.MN.wlan[0].mac: # current transmission: none
p3.MN.wlan[0].mgmt: ** Notification at T=87.721757464096 to p3.MN.wlan[0].mgmt: RADIO-STATE RECV, channel #2, 2Mbps
p3.MN.wlan[0].mgmt: busy radio channel detected during scanning
Radio::handleSelfMsg END
** Event #27830  t=87.753989521879  p3.MN.tcp (TCP, id=76), on selfmsg `CONN-ESTAB' (cMessage, id=2100)
Connection <unspec>:1025 to aaaa:1:2:0:8aa:ff:fe00:6:1000  on app[0], connId=2  in SYN SENT
CONN-ESTAB timer expired
Notifying app: TIMED OUT
Transition: SYN SENT --> CLOSED  (event was: TIMEOUT CONN ESTAB)
Notifying app: CLOSED
Deleting TCP connection
```

Imagen 26. Ruido

- Diferentes movimientos: WirelessHost6.mobilityType
  Este parámetro permite cambiar el tipo de movimiento que realiza el nodo móvil pudiendo provocar desconexiones con el PA a lo largo del tiempo. Dependiendo de

la posición inicial del MN con respecto al PA, algunos movimientos podrían provocar mayores desconexiones que otros tipos de movimientos. En algunos casos, esto puede provocar que se permanezca más tiempo lejos del PA lo cual incrementaría el ruído de la señal.
Algunos ejemplos de movimientos son: CircleMobility, RectangleMobility, LinearMobility.

- Velocidades: WirelessHost6.mobility.speed
  Velocidad a la que se mueve el nodo. Esto es influyente dependiendo del tipo de movimiento usado en el MN. Si el MN puede salirse del rango del PA, esto implicaría mayores conexiones y desconexiones del MN con una frecuencia menor.
  En este caso, como hay un único MN y un único punto de acceso, este parámetro tiene menor importancia.