MINISTRY OF HIGHER EDUCATION

MINISTERE DE L’ENSEIGNEMENT SUPERIEUR

Peace-Work-Fatherland

Paix-Travail-Patrie

REPUBLIC OF CAMEROON

REPUBLIQUE DU CAMEROUN



UNIVERSITY OF NGAOUNDERE

UNIVERSITE DE NGAOUNDERE



FACULTE DES SCIENCES

DEPARTEMENT DE MATHEMATIQUES ET INFORMATIQUE

FACULTY OF SCIENCES

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

**UE** **: Technique de Simulation réseaux**

**PARCOURS :** **Systèmes et Logiciels en Environnements Distribués** (**SLED**)

**NIVEAU** : **MASTER 1**

**Travail réalisé par** :

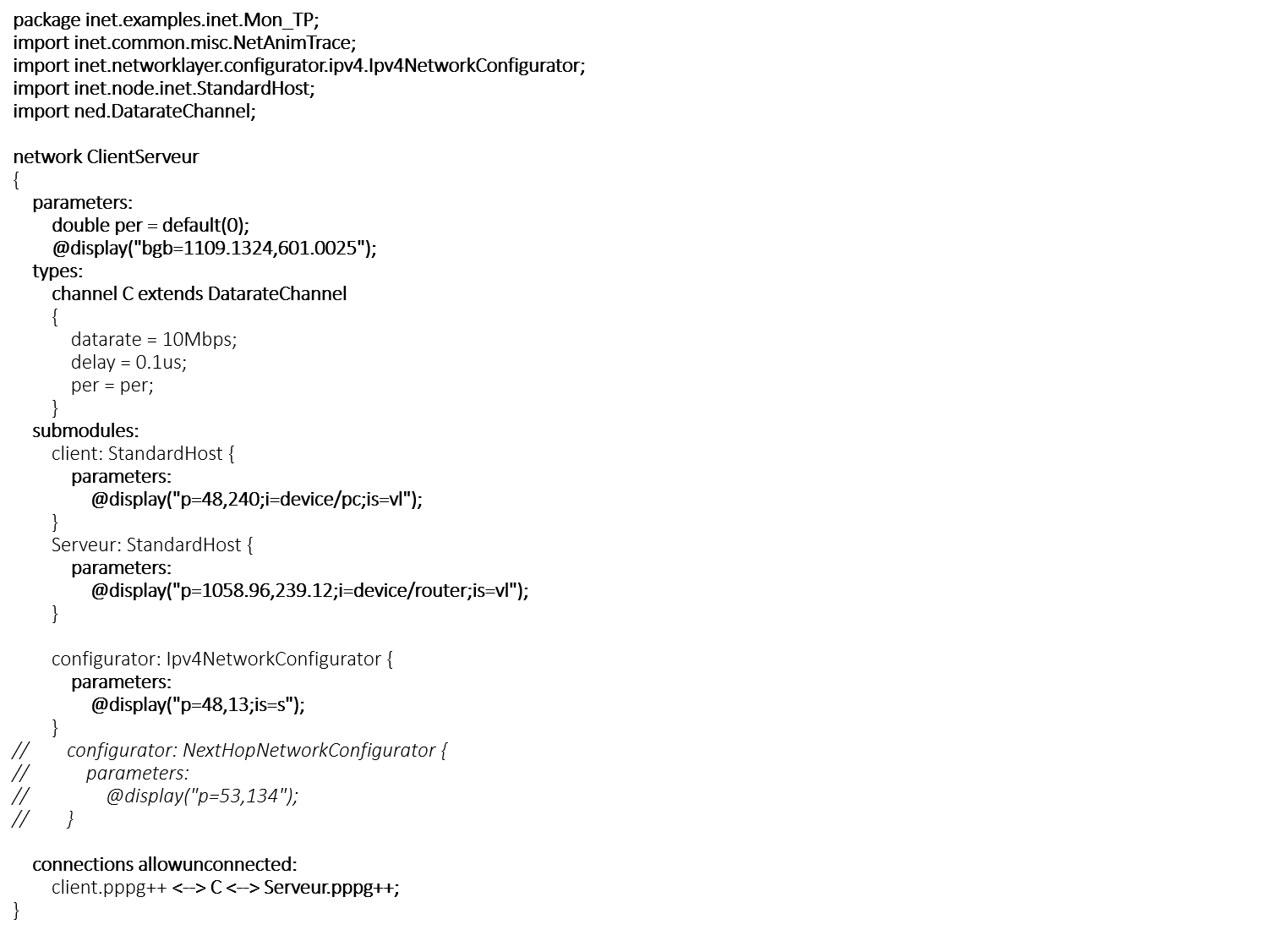
|  |  |  |
| --- | --- | --- |
| **N°** | **NOM ET PRENOM(S)** | **MATRICULE** |
|  | PRAKASSO DANIEL | 19B683FS |
|  | MPON A DANG CHARLES |  |
|  | OUSSOUMANOU BENOIT |  |
|  | NKENG ADONIS |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

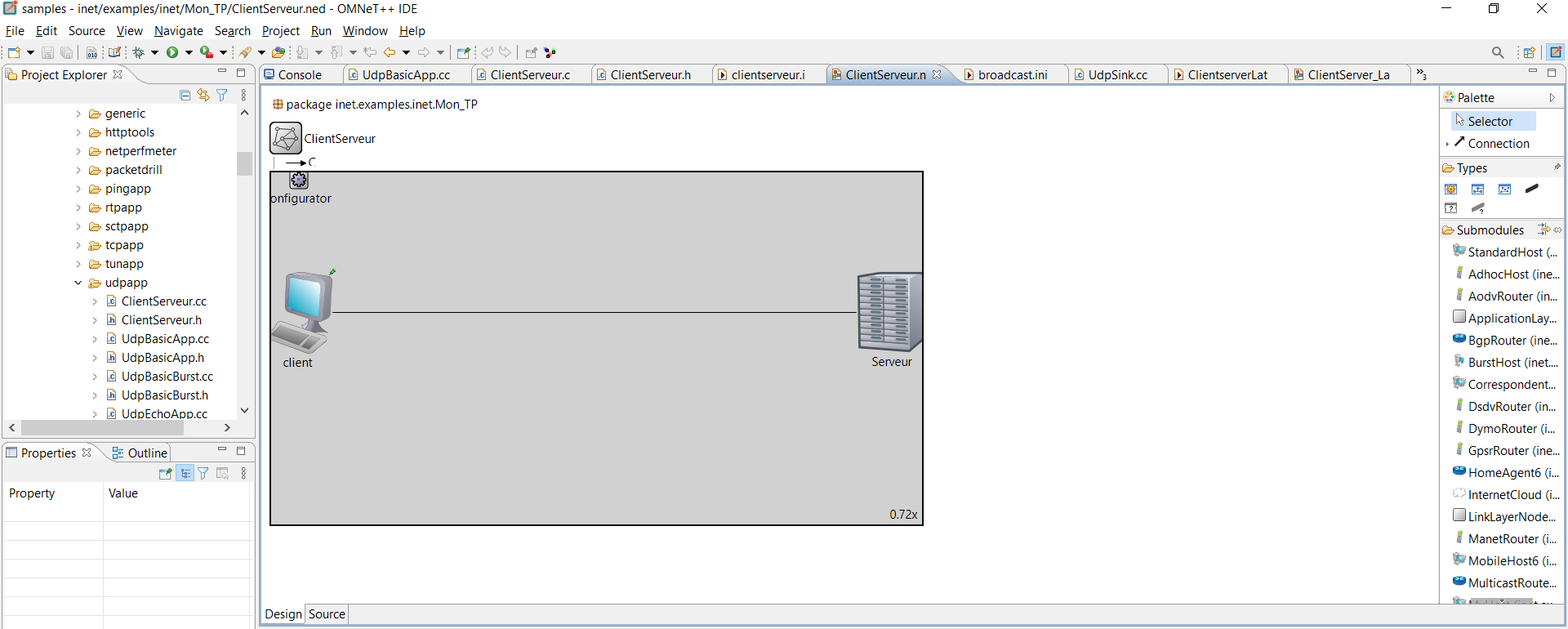
***Sous l’encadrement de :* Mme Ndo Serge née Zongo Minette**

*Année académique 2019/2020*

1. **La communication entre deux machines par sockets UDP**

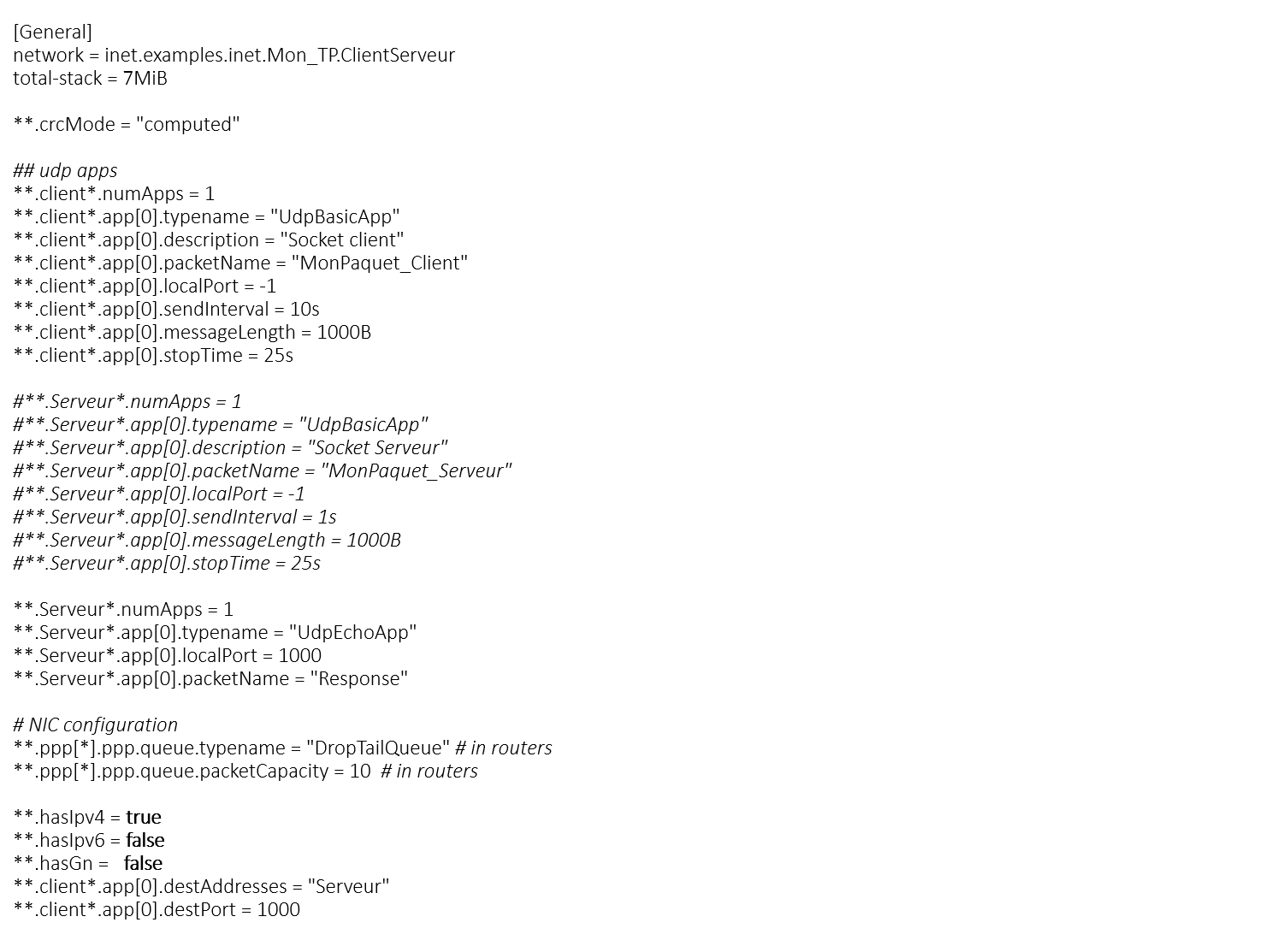
Pour implémenter la communication entre un client et un serveur sur omnetpp, nous allons nous servir de module udapp défini dans inet-src-applications-udapp-UdpBasicApp. Pour le faire nous créons un dossier que nous nommons « Mon\_TP ». Dans ce dossier nous créons un fichier NED, « **inet/examples/inet/Mon\_TP/clientserver.ned** » ayant le contenu suivant :





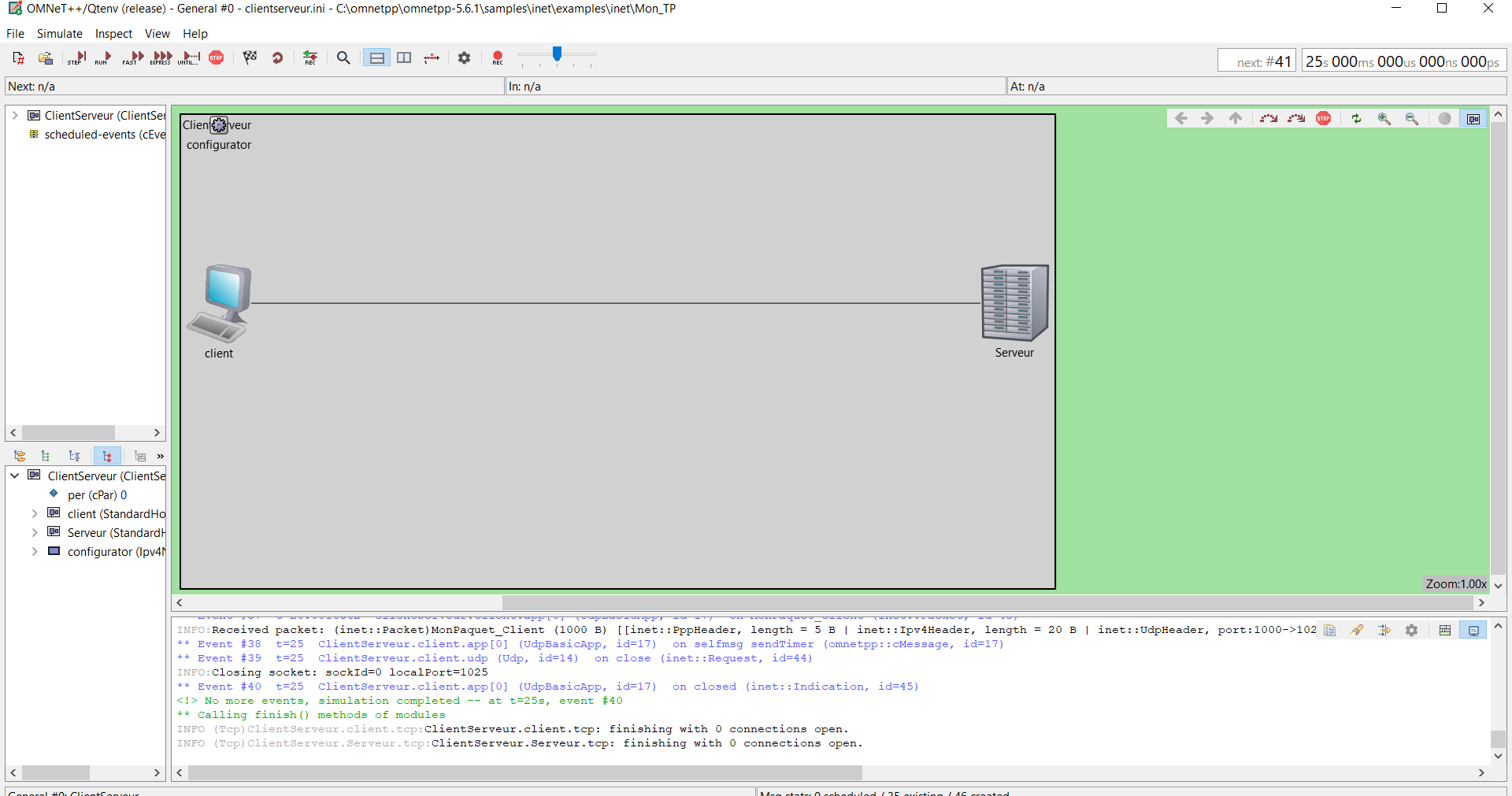
Description : Nous avons un client et un serveur de type StandardHost. Puis nous créons la liaison entre les deux par le biais de « channel ».

Pour finaliser cela, nous créons un fichier d’initialisation, **inet/examples/inet/Mon\_TP/clientserveur.ini** ayant le contenu suivant :

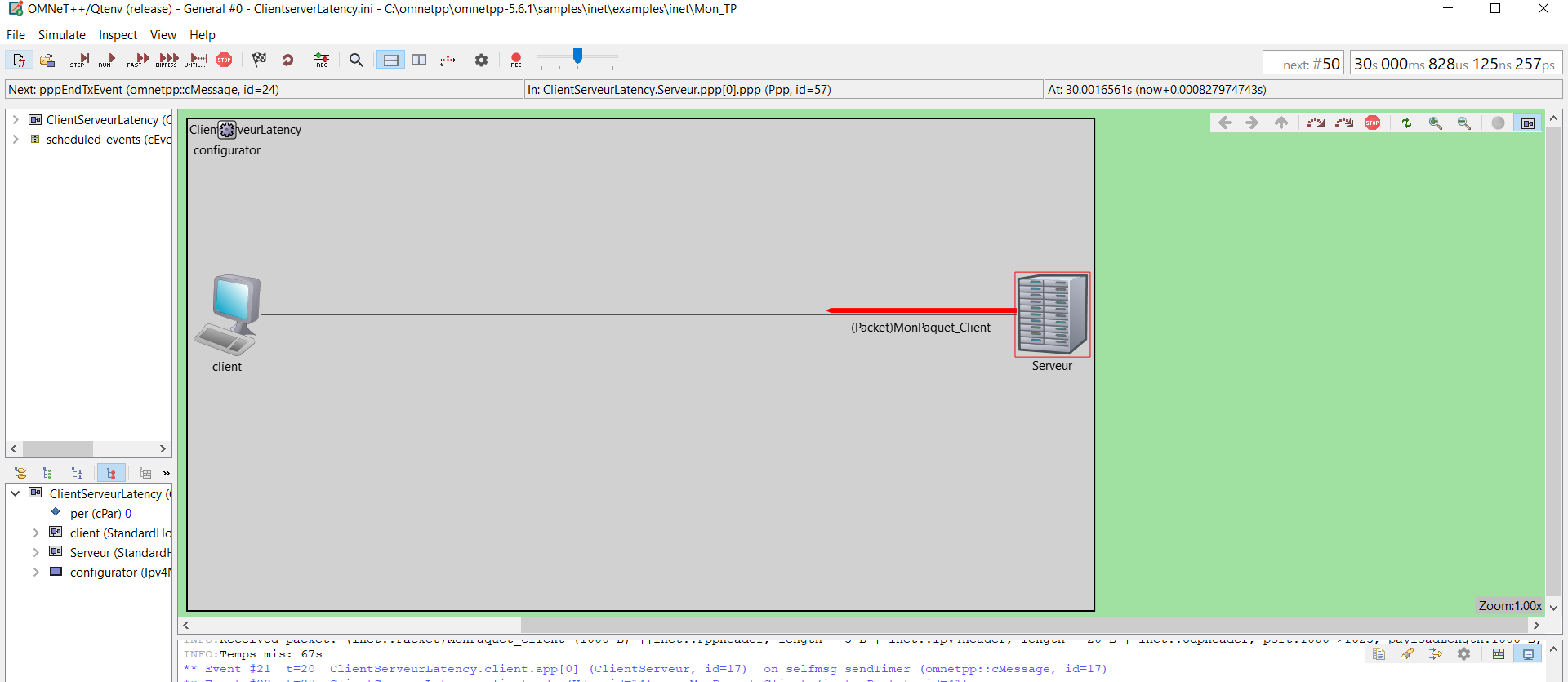


C’est dans ce fichier que certaines propriétés telles que : nom du socket, le port de destination, l’adresse de destination, la taille du message… sont définies.

Il ne reste plus qu’à compiler et exécuter notre programme. Nous obtenons la figure suivante :



En lancant le deroulement nous obtenons :



1. **La communication entre deux machines par sockets UDP avec un temps t de latence**

Pour obtenir le resultat demandé, nous definissons un module ayant les memes methodes que le module UdpBasicApp, à la difference des attributs. Nous allons ajouter deux attributs depart et arrive de type time\_t qui existe dans ctime de c++. Ainsi nous avons le fichier **inet/src/inet/applications/udpapp/ClientServeur.h** ayant le contenu :

**#ifndef** \_\_INET\_CLIENTSERVEUR\_H\_

**#define** \_\_INET\_CLIENTSERVEUR\_H\_

**#include** <vector>

**#include** <ctime>

**#include** "inet/common/INETDefs.h"

**#include** "inet/applications/base/ApplicationBase.h"

**#include** "inet/transportlayer/contract/udp/UdpSocket.h"

**namespace** inet {

/\*\*

\* UDP application. See NED for more info.

\*/

**class** INET\_API ClientServeur : **public** ApplicationBase, **public** UdpSocket::ICallback

{

**protected**:

**enum** SelfMsgKinds { *START* = 1, *SEND*, *STOP* };

// parameters

std::vector<L3Address> destAddresses;

std::vector<std::string> destAddressStr;

**int** localPort = -1, destPort = -1;

simtime\_t startTime;

simtime\_t stopTime;

**bool** dontFragment = **false**;

**const** **char** \*packetName = **nullptr**;

// state

UdpSocket socket;

cMessage \*selfMsg = **nullptr**;

// statistics

**int** numSent = 0;

**int** numReceived = 0;

time\_t depart;

time\_t arrive;

**protected**:

**virtual** **int** **numInitStages**() **const** **override** { **return** *NUM\_INIT\_STAGES*; }

**virtual** **void** **initialize**(**int** stage) **override**;

**virtual** **void** **handleMessageWhenUp**(cMessage \*msg) **override**;

**virtual** **void** **finish**() **override**;

**virtual** **void** **refreshDisplay**() **const** **override**;

// chooses random destination address

**virtual** L3Address **chooseDestAddr**();

**virtual** **void** **sendPacket**();

**virtual** **void** **processPacket**(Packet \*msg);

**virtual** **void** **setSocketOptions**();

**virtual** **void** **processStart**();

**virtual** **void** **processSend**();

**virtual** **void** **processStop**();

**virtual** **void** **handleStartOperation**(LifecycleOperation \*operation) **override**;

**virtual** **void** **handleStopOperation**(LifecycleOperation \*operation) **override**;

**virtual** **void** **handleCrashOperation**(LifecycleOperation \*operation) **override**;

**virtual** **void** **socketDataArrived**(UdpSocket \*socket, Packet \*packet) **override**;

**virtual** **void** **socketErrorArrived**(UdpSocket \*socket, Indication \*indication) **override**;

**virtual** **void** **socketClosed**(UdpSocket \*socket) **override**;

**public**:

**ClientServeur**() {}

**~ClientServeur**();

};

} // namespace inet

**#endif** // ifndef \_\_INET\_CLIENTSERVEUR\_H

Et nous creons un fichier source **inet/src/inet/applications/udpapp/ClientServeur.cc** ayant le contenu :

**#include** <ctime>

**#include** <iostream>

**#include** "inet/applications/base/ApplicationPacket\_m.h"

**#include** "ClientServeur.h"

**#include** "inet/common/ModuleAccess.h"

**#include** "inet/common/TagBase\_m.h"

**#include** "inet/common/TimeTag\_m.h"

**#include** "inet/common/lifecycle/ModuleOperations.h"

**#include** "inet/common/packet/Packet.h"

**#include** "inet/networklayer/common/FragmentationTag\_m.h"

**#include** "inet/networklayer/common/L3AddressResolver.h"

**#include** "inet/transportlayer/contract/udp/UdpControlInfo\_m.h"

**namespace** inet {

Define\_Module(ClientServeur);

**ClientServeur::~ClientServeur**()

{

**cancelAndDelete**(selfMsg);

}

**void** **ClientServeur::initialize**(**int** stage)

{

ApplicationBase::initialize(stage);

**if** (stage == *INITSTAGE\_LOCAL*) {

numSent = 0;

numReceived = 0;

WATCH(numSent);

WATCH(numReceived);

localPort = **par**("localPort");

destPort = **par**("destPort");

startTime = **par**("startTime");

stopTime = **par**("stopTime");

packetName = **par**("packetName");

dontFragment = **par**("dontFragment");

**if** (stopTime >= SIMTIME\_ZERO && stopTime < startTime)

**throw** cRuntimeError("Invalid startTime/stopTime parameters");

selfMsg = **new** cMessage("sendTimer");

}

}

**void** **ClientServeur::finish**()

{

**recordScalar**("packets sent", numSent);

**recordScalar**("packets received", numReceived);

ApplicationBase::**finish**();

}

**void** **ClientServeur::setSocketOptions**()

{

**int** timeToLive = **par**("timeToLive");

**if** (timeToLive != -1)

socket.setTimeToLive(timeToLive);

**int** dscp = **par**("dscp");

**if** (dscp != -1)

socket.setDscp(dscp);

**int** tos = **par**("tos");

**if** (tos != -1)

socket.setTos(tos);

**const** **char** \*multicastInterface = **par**("multicastInterface");

**if** (multicastInterface[0]) {

IInterfaceTable \*ift = getModuleFromPar<IInterfaceTable>(**par**("interfaceTableModule"), **this**);

InterfaceEntry \*ie = ift->findInterfaceByName(multicastInterface);

**if** (!ie)

**throw** cRuntimeError("Wrong multicastInterface setting: no interface named \"%s\"", multicastInterface);

socket.setMulticastOutputInterface(ie->getInterfaceId());

}

**bool** receiveBroadcast = **par**("receiveBroadcast");

**if** (receiveBroadcast)

socket.setBroadcast(**true**);

**bool** joinLocalMulticastGroups = **par**("joinLocalMulticastGroups");

**if** (joinLocalMulticastGroups) {

MulticastGroupList mgl = getModuleFromPar<IInterfaceTable>(**par**("interfaceTableModule"), **this**)->collectMulticastGroups();

socket.joinLocalMulticastGroups(mgl);

}

socket.setCallback(**this**);

}

L3Address **ClientServeur::chooseDestAddr**()

{

**int** k = intrand(destAddresses.size());

**if** (destAddresses[k].isUnspecified() || destAddresses[k].isLinkLocal()) {

L3AddressResolver().tryResolve(destAddressStr[k].c\_str(), destAddresses[k]);

}

**return** destAddresses[k];

}

**void** **ClientServeur::sendPacket**()

{

time\_t now = **time**(0);

depart = now;

std::ostringstream str;

str << packetName;

Packet \*packet = **new** Packet(str.str().c\_str());

**if**(dontFragment)

packet->addTag<FragmentationReq>()->setDontFragment(**true**);

**const** **auto**& payload = makeShared<ApplicationPacket>();

payload->setChunkLength(B(**par**("messageLength")));

payload->setSequenceNumber(numSent);

payload->addTag<CreationTimeTag>()->setCreationTime(simTime());

packet->insertAtBack(payload);

L3Address destAddr = chooseDestAddr();

**emit**(packetSentSignal, packet);

socket.sendTo(packet, destAddr, destPort);

numSent++;

}

**void** **ClientServeur::processStart**()

{

socket.setOutputGate(**gate**("socketOut"));

**const** **char** \*localAddress = **par**("localAddress");

socket.bind(\*localAddress ? L3AddressResolver().resolve(localAddress) : L3Address(), localPort);

setSocketOptions();

**const** **char** \*destAddrs = **par**("destAddresses");

cStringTokenizer tokenizer(destAddrs);

**const** **char** \*token;

**while** ((token = tokenizer.**nextToken**()) != **nullptr**) {

destAddressStr.push\_back(token);

L3Address result;

L3AddressResolver().tryResolve(token, result);

**if** (result.isUnspecified())

EV\_ERROR << "cannot resolve destination address: " << token << **endl**;

destAddresses.push\_back(result);

}

**if** (!destAddresses.empty()) {

selfMsg->setKind(*SEND*);

processSend();

}

**else** {

**if** (stopTime >= SIMTIME\_ZERO) {

selfMsg->setKind(*STOP*);

**scheduleAt**(stopTime, selfMsg);

}

}

}

**void** **ClientServeur::processSend**()

{

sendPacket();

simtime\_t d = simTime() + **par**("sendInterval");

**if** (stopTime < SIMTIME\_ZERO || d < stopTime) {

selfMsg->setKind(*SEND*);

**scheduleAt**(d, selfMsg);

}

**else** {

selfMsg->setKind(*STOP*);

**scheduleAt**(stopTime, selfMsg);

}

}

**void** **ClientServeur::processStop**()

{

socket.close();

}

**void** **ClientServeur::handleMessageWhenUp**(cMessage \*msg)

{

**if** (msg->isSelfMessage()) {

ASSERT(msg == selfMsg);

**switch** (selfMsg->getKind()) {

**case** *START*:

processStart();

**break**;

**case** *SEND*:

processSend();

**break**;

**case** *STOP*:

processStop();

**break**;

**default**:

**throw** cRuntimeError("Invalid kind %d in self message", (**int**)selfMsg->getKind());

}

}

**else**

socket.processMessage(msg);

}

**void** **ClientServeur::socketDataArrived**(UdpSocket \*socket, Packet \*packet)

{

// process incoming packet

processPacket(packet);

}

**void** **ClientServeur::socketErrorArrived**(UdpSocket \*socket, Indication \*indication)

{

EV\_WARN << "Ignoring UDP error report " << indication->getName() << **endl**;

**delete** indication;

}

**void** **ClientServeur::socketClosed**(UdpSocket \*socket)

{

**if** (operationalState == State::*STOPPING\_OPERATION*)

startActiveOperationExtraTimeOrFinish(**par**("stopOperationExtraTime"));

}

**void** **ClientServeur::refreshDisplay**() **const**

{

ApplicationBase::refreshDisplay();

**char** buf[100];

sprintf(buf, "rcvd: %d pks\nsent: %d pks", numReceived, numSent);

**getDisplayString**().**setTagArg**("t", 0, buf);

}

**void** **ClientServeur::processPacket**(Packet \*pk)

{

time\_t now = **time**(0);

arrive = now;

**emit**(packetReceivedSignal, pk);

EV\_INFO << "Received packet: " << UdpSocket::*getReceivedPacketInfo*(pk) <<**endl**;

EV << "Temps mis: "<<(arrive - depart)/2<< "s"<<**endl**;

**delete** pk;

numReceived++;

}

**void** **ClientServeur::handleStartOperation**(LifecycleOperation \*operation)

{

simtime\_t start = std::max(startTime, simTime());

**if** ((stopTime < SIMTIME\_ZERO) || (start < stopTime) || (start == stopTime && startTime == stopTime)) {

selfMsg->setKind(*START*);

**scheduleAt**(start, selfMsg);

}

}

**void** **ClientServeur::handleStopOperation**(LifecycleOperation \*operation)

{

**cancelEvent**(selfMsg);

socket.close();

delayActiveOperationFinish(**par**("stopOperationTimeout"));

}

**void** **ClientServeur::handleCrashOperation**(LifecycleOperation \*operation)

{

**cancelEvent**(selfMsg);

socket.destroy(); //**TODO** in real operating systems, program crash detected by OS and OS closes sockets of crashed programs.

}

} // namespace inet

Pour calculer le temps mis:

Lors de l’envoi du paquet, nous utilisons la fonction time(0) de ctime qui nous renvoie le temps en secondes que nous attribuons à la variable départ, à l’arrivée, on applique le même principe, on attribue à la variable arrive. Le temps mis est la différence entre arrive et depart.

Il ne reste qu’à définir un fichier de description **inet/src/inet/applications/udpapp/ClientServeur.ned** ayant le meme contenu que **inet/src/inet/applications/udpapp/UdpBasicApp.ned** pour que notre module soit utilisable.

Maintenant, nous creons un autre fichier de description **inet/examples/inet/Mon\_TP/clientserveur\_Latency.ned avec le contenu :**

**package** inet.examples.inet.Mon\_TP;

**import** inet.common.misc.NetAnimTrace;

**import** inet.networklayer.configurator.ipv4.Ipv4NetworkConfigurator;

**import** inet.node.inet.StandardHost;

**import** ned.DatarateChannel;

**network** ClientServeur

{

**parameters**:

**double** per = **default**(0);

**@display**("bgb=1109.1324,601.0025");

**types**:

**channel** C **extends** DatarateChannel

{

datarate = 10Mbps;

delay = 0.1us;

per = per;

}

**submodules**:

client: StandardHost {

**parameters**:

**@display**("p=48,240;i=device/pc;is=vl");

}

Serveur: StandardHost {

**parameters**:

**@display**("p=1058.96,239.12;i=device/router;is=vl");

}

configurator: Ipv4NetworkConfigurator {

**parameters**:

**@display**("p=48,13;is=s");

}

*// configurator: NextHopNetworkConfigurator {*

*// parameters:*

*// @display("p=53,134");*

*// }*

**connections** **allowunconnected**:

client.pppg++ **<-->** C **<-->** Serveur.pppg++;

}

Et un fichier **inet/examples/inet/Mon\_TP/ClientServeurLatency.ini**

[General]

network = inet.examples.inet.Mon\_TP.ClientServeurLatency

total-stack = 7MiB

\*\*.crcMode = "computed"

*## udp apps*

\*\*.client\*.numApps = 1

\*\*.client\*.app[0].typename = "ClientServeur"

\*\*.client\*.app[0].description = "Socket client"

\*\*.client\*.app[0].packetName = "MonPaquet\_Client"

\*\*.client\*.app[0].localPort = -1

\*\*.client\*.app[0].sendInterval = 10s

\*\*.client\*.app[0].messageLength = 1000B

*#\*\*.Serveur\*.numApps = 1*

*#\*\*.Serveur\*.app[0].typename = "UdpBasicApp"*

*#\*\*.Serveur\*.app[0].description = "Socket Serveur"*

*#\*\*.Serveur\*.app[0].packetName = "MonPaquet\_Serveur"*

*#\*\*.Serveur\*.app[0].localPort = -1*

*#\*\*.Serveur\*.app[0].sendInterval = 1s*

*#\*\*.Serveur\*.app[0].messageLength = 1000B*

*#\*\*.Serveur\*.app[0].stopTime = 25s*

\*\*.Serveur\*.numApps = 1

\*\*.Serveur\*.app[0].typename = "UdpEchoApp"

\*\*.Serveur\*.app[0].localPort = 1000

\*\*.Serveur\*.app[0].packetName = "Response"

*# NIC configuration*

\*\*.ppp[\*].ppp.queue.typename = "DropTailQueue" *# in routers*

\*\*.ppp[\*].ppp.queue.packetCapacity = 10 *# in routers*

\*\*.hasIpv4 = **true**

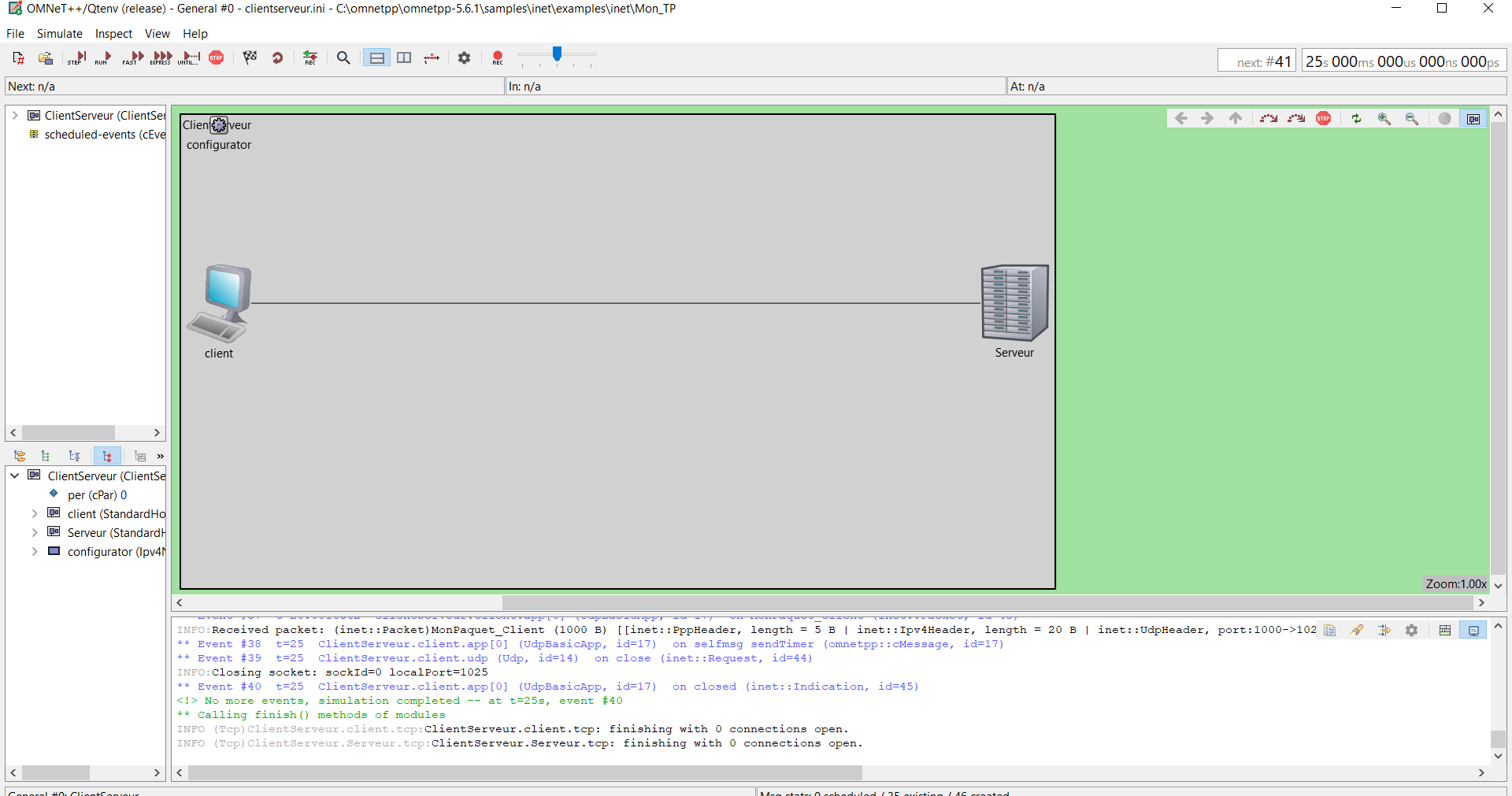
\*\*.hasIpv6 = **false**

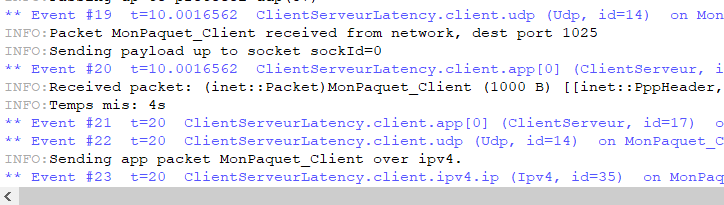
\*\*.hasGn = **false**

\*\*.client\*.app[0].destAddresses = "Serveur"

\*\*.client\*.app[0].destPort = 1000

Ici, nous donnons comme typeName ‘ClientServeur’

Il reste exécuter notre programme. Nous obtenons : 



**Nous obtenons un temps mis de 4s.**

1. **Broadcast de 250 machines avec le pourcentage de reception**

Pour le faire nous definissons un fichier **inet/examples/inet/Mon\_TP/Broadcast.ned :**

**package** inet.examples.inet.Mon\_TP;

**import** inet.networklayer.configurator.ipv4.Ipv4NetworkConfigurator;

**import** inet.node.ethernet.EtherSwitch;

**import** inet.node.inet.Router;

**import** inet.node.inet.StandardHost;

**import** ned.DatarateChannel;

**network** Broadcast

{

**parameters**:

**int** numTargets;

**@display**("bgb=2300.826,1723.4099");

**types**:

**channel** C **extends** DatarateChannel

{

datarate = 10Mbps;

delay = 0.1us;

}

**submodules**:

target[numTargets]: StandardHost {

**parameters**:

**@display**("i=device/pc3;p=1157.09,110.528,col,50");

}

Serveur: StandardHost {

**parameters**:

**@display**("p=138.46199,111.948;i=device/pc2");

}

R2: Router {

**@display**("p=536.172,170.868");

}

configurator: Ipv4NetworkConfigurator {

**parameters**:

config = **xmldoc**("config.xml");

**@display**("p=12,7;is=s");

}

switch: EtherSwitch {

**@display**("p=228,804;is=vl");

}

**connections**:

Serveur.pppg++ **<-->** C **<-->** R2.pppg++;

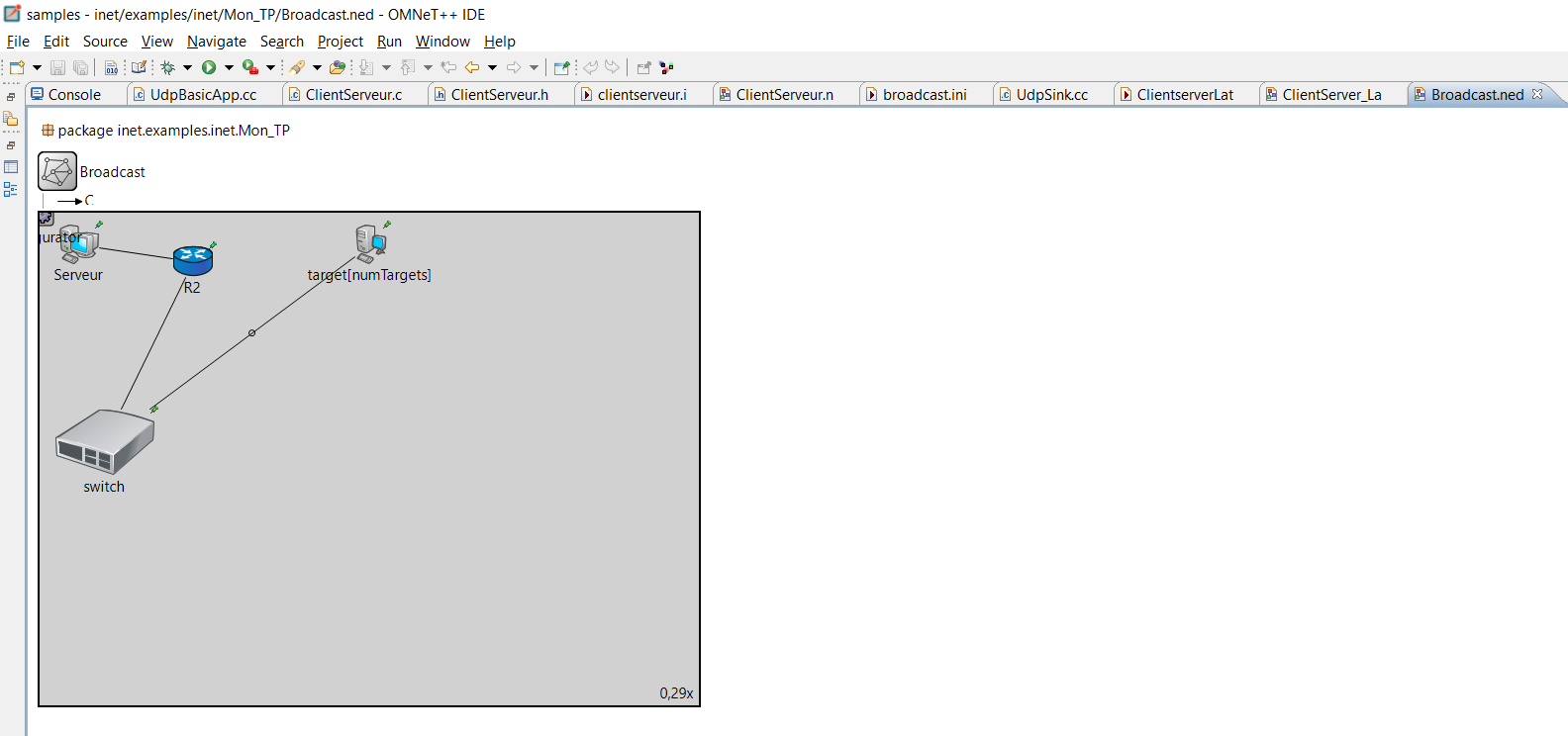
R2.ethg++ **<-->** C **<-->** switch.ethg++;

**for** i=0..numTargets-1 {

switch.ethg++ **<-->** C **<-->** target[i].ethg++;

}

}



Ici, nous définissons un serveur, un router, un switch et un ensemble de numTargets de machines

Il faut definir le fichier d’initialisation **inet/examples/inet/Mon\_TP/Broadcast.ini**

[General]

network = inet.examples.inet.Mon\_TP.Broadcast

\*\*.Serveur.numApps = 1

\*\*.Serveur.app[0].typename = "UdpBasicApp"

\*\*.Serveur.app[\*].destAddresses = "10.0.1.255"

\*\*.Serveur.app[0].destPort = 1000

\*\*.Serveur.app[0].messageLength = 100B

\*\*.Serveur.app[0].startTime = 10s

\*\*.Serveur.app[0].sendInterval = 1s

\*\*.numTargets = 250

\*\*.target[\*].numApps = 1

\*\*.target[\*].app[0].typename = "UdpSink"

\*\*.target[\*].app[0].localPort = 1000

\*.R2.ipv4.ip.directBroadcastInterfaces = "eth0"

Il reste definir le fichier de configuration **inet/examples/inet/Mon\_TP/config.xml :**

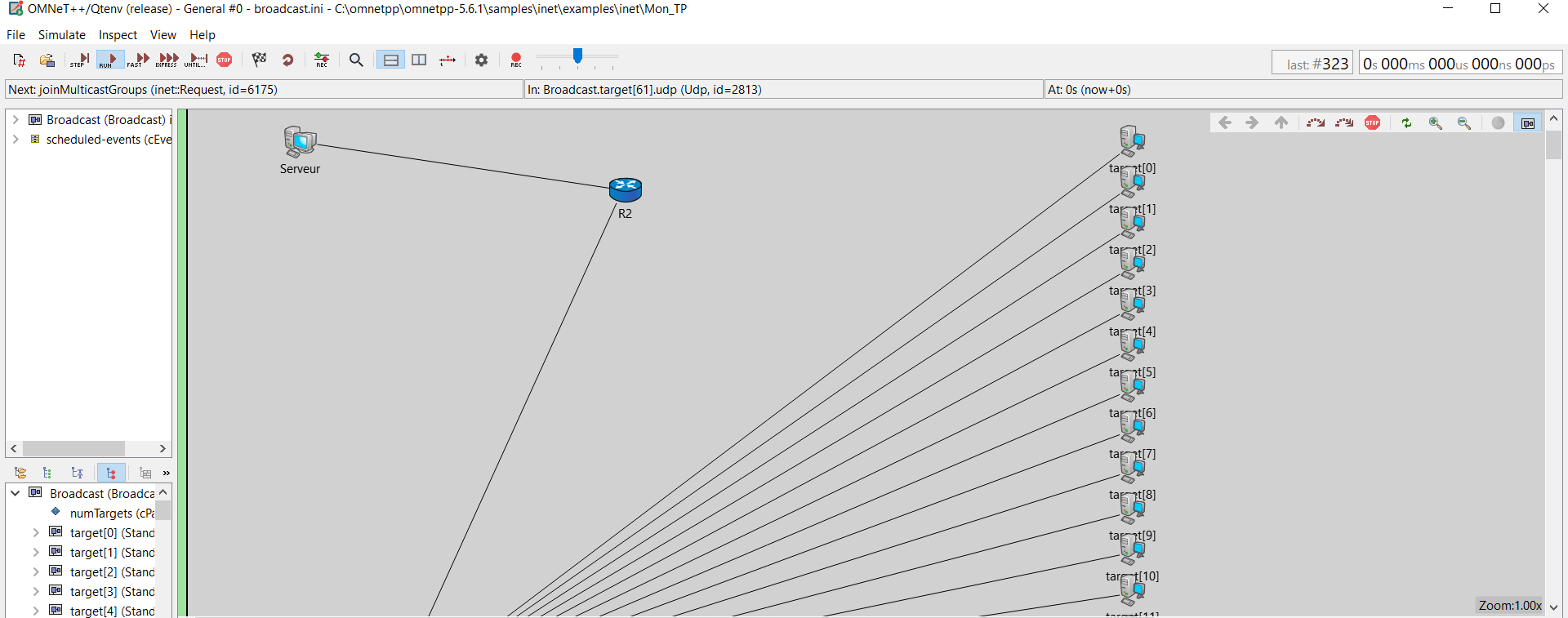
<config>

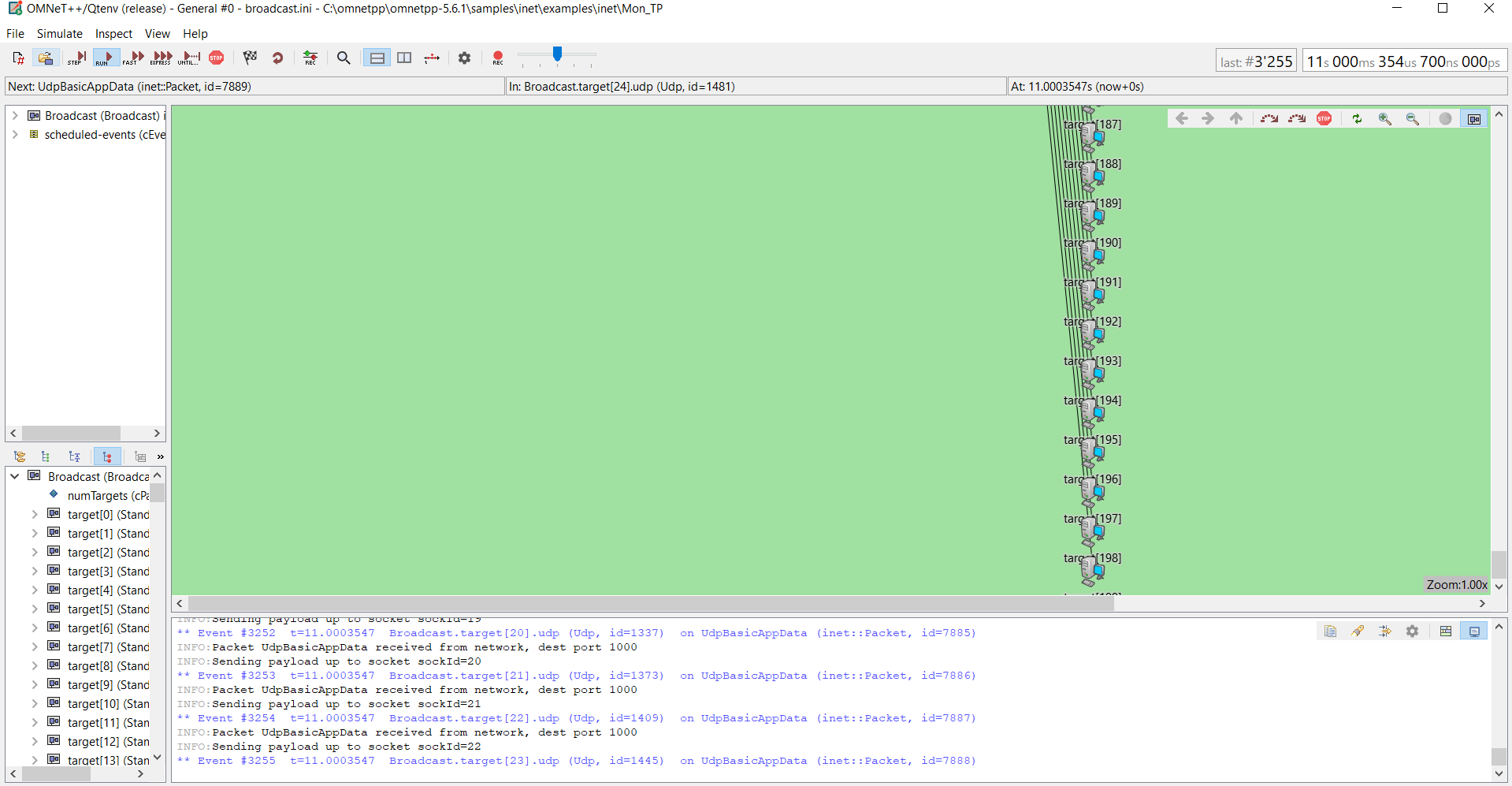
<interface hosts='target\*' address='10.0.1.x' netmask='255.255.255.0'/>

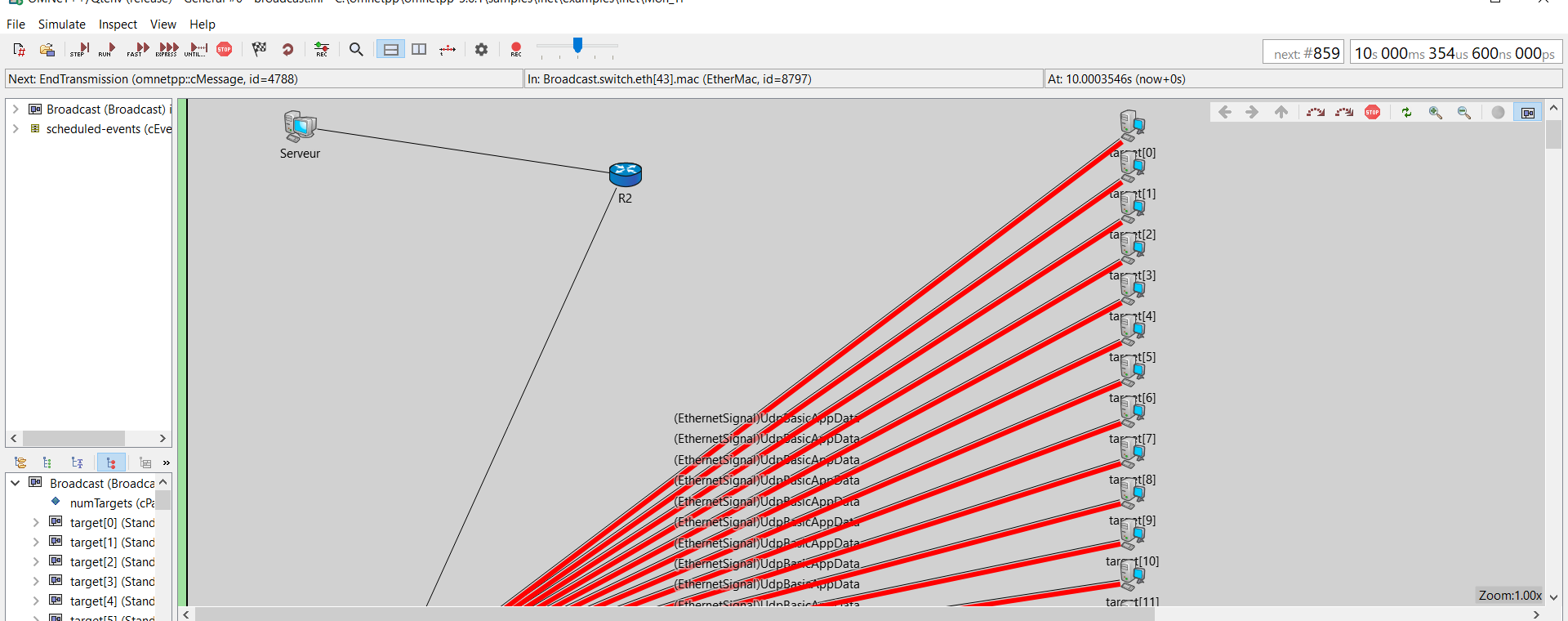
<interface hosts='\*' address='10.x.x.x' netmask='255.255.255.0'/>

</config>

**Maintenant il reste compiler et executer notre travail :**

****





1. **Graphes de résultats**

Nous obtenons les graphes suivants :

