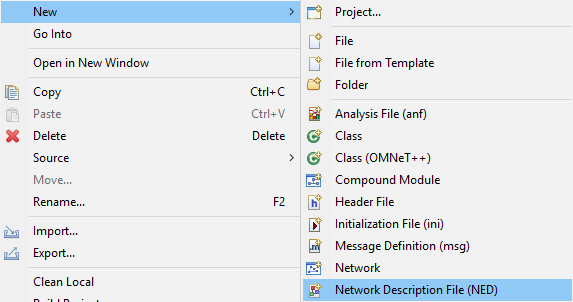
## Wie wird eine Topologie erstellt und dessen Funktionalität realisiert?

OMNeT++ uses NED files to define components and to assemble them into larger units like networks. We start implementing our model by adding a NED file. To add the file to the project, right-click the project directory in the Project Explorer panel on the left, and choose New -> Network Description File (NED) from the menu. Enter tictoc1.ned when prompted for the file name.



Once created, the file can be edited in the Editor area of the OMNeT++ IDE. The OMNeT++ IDE's NED editor has two modes, Design and Source; one can switch between them using the tabs at the bottom of the editor. In Design mode, the topology can be edited graphically, using the mouse and the palette on the right. In Source mode, the NED source code can be directly edited as text. Changes done in one mode will be immediately reflected in the other, so you can freely switch between modes during editing, and do each change in whichever mode it is more convenient. (Since NED files are plain text files, you can even use an external text editor to edit them, although you'll miss syntax highlighting, content assist, cross-references and other IDE features.)

## Kompilieren und Starten einer Simulation

We now need to implement the functionality of the [**Txc1**](https://www.omnetpp.org/doc/omnetpp/tictoc-tutorial/classTxc1.html) simple module in C++.

To be able to run the simulation, we need to create an omnetpp.ini file. omnetpp.ini tells the simulation program which network you want to simulate (as NED files may contain several networks), you can pass parameters to the model, explicitly specify seeds for the random number generators, etc.

Once you complete the above steps, you can launch the simulation by selecting omnetpp.ini (in either the editor area or the *Project Explorer*), and pressing the *Run*button.

The IDE will build your project automatically. If there are compilation errors, you need to rectify those until you get an error-free compilation and linking.

## Hinzufügen von graphischen Elementen, Ausgaben zur Fehlersuche, Zustandsvariablen und (zufälligen) Parametern.

The simulation is just a C++ program, and as such, it often needs to be debugged while it is being developed. In this section we'll look at the basics of debugging to help you acquire this vital task.

The simulation can be started in debug mode by clicking the Debug button on the IDE's main toolbar.

Here we make the model look a bit prettier in the GUI. We assign the "block/routing" icon (the file images/block/routing.png), and paint it cyan for tic and yellow for toc. This is achieved by adding display strings to the NED file. The i= tag in the display string specifies the icon.

private:

int counter; // Note the counter here

[WATCH](https://www.omnetpp.org/doc/omnetpp/api/group__WatchMacros.html#ga0878b62c3a2dcb0388c967a4acb2f18a)(counter);

Module parameters have to be declared in the NED file. The data type can be numeric, string, bool, or xml (the latter is for easy access to XML config files), among others.

Ned:

parameters:

        bool sendMsgOnInit = default(false); // whether the module should send out a message on initialization

        int limit = default(2);   // another parameter with a default value

        @display("i=block/routing");

cc:

// Initialize the counter with the "limit" module parameter, declared

    // in the NED file (tictoc4.ned).

counter = par("limit");

(zufälligen) Parametern: 7

if ([uniform](https://www.omnetpp.org/doc/omnetpp/api/group__RandomNumbersCont.html#ga110654e0d90a50f94b159b71fce85ea0)(0, 1) < 0.1) {

[EV](https://www.omnetpp.org/doc/omnetpp/api/group__Logging.html#ga650ef3eff8a2900bef69dae29c05d2dd) << "\"Losing\" message\n";

delete msg;

}

## Vererbung, Verzögerung, Zeitüberschreitungen und dessen Aufhebungen.

Vererbung 5

If we take a closer look at the NED file we will realize that tic and toc differs only in their parameter values and their display string. We can create a new simple module type by inheriting from an other one and specifying or overriding some of its parameters. In our case we will derive two simple module types (Tic and Toc). Later we can use these types when defining the submodules in the network.

simple [Txc5](https://www.omnetpp.org/doc/omnetpp/tictoc-tutorial/classTxc5.html)

{

parameters:

bool sendMsgOnInit = default(false);

int limit = default(2);

@display("i=block/routing");

gates:

input in;

output out;

}

simple Tic5 extends [Txc5](https://www.omnetpp.org/doc/omnetpp/tictoc-tutorial/classTxc5.html)

{

parameters:

@display("i=,cyan");

sendMsgOnInit = true; // Tic modules should send a message on init

}

simple Toc5 extends [Txc5](https://www.omnetpp.org/doc/omnetpp/tictoc-tutorial/classTxc5.html)

{

parameters:

@display("i=,gold");

sendMsgOnInit = false; // Toc modules should NOT send a message on init

}

Verzögerung 6

In the previous models, tic and toc immediately sent back the received message. Here we'll add some timing: tic and toc will hold the message for 1 simulated second before sending it back. In OMNeT++ such timing is achieved by the module sending a message to itself. Such messages are called self-messages (but only because of the way they are used, otherwise they are ordinary message objects).

We added two [cMessage](https://www.omnetpp.org/doc/omnetpp/api/classomnetpp_1_1cMessage.html) \* variables, event and tictocMsg to the class, to remember the message we use for timing and message whose processing delay we are simulating.

scheduleAt(simTime()+1.0, event);

 // There are several ways of distinguishing messages, for example by message

    // kind (an int attribute of cMessage) or by class using dynamic\_cast

    // (provided you subclass from cMessage). In this code we just check if we

    // recognize the pointer, which (if feasible) is the easiest and fastest

    // method.

    if (msg == event) {

        // The self-message arrived, so we can send out tictocMsg and nullptr out

        // its pointer so that it doesn't confuse us later.

        EV << "Wait period is over, sending back message\n";

        send(tictocMsg, "out");

        tictocMsg = nullptr;

    }

    else {

        // If the message we received is not our self-message, then it must

        // be the tic-toc message arriving from our partner. We remember its

        // pointer in the tictocMsg variable, then schedule our self-message

        // to come back to us in 1s simulated time.

        EV << "Message arrived, starting to wait 1 sec...\n";

        tictocMsg = msg;

        scheduleAt(simTime()+1.0, event);

}

Zeitüberschreitungen: 8

In order to get one step closer to modelling networking protocols, let us transform our model into a stop-and-wait simulation. This time we'll have separate classes for tic and toc. The basic scenario is similar to the previous ones: tic and toc will be tossing a message to one another. However, toc will "lose" the message with some nonzero probability, and in that case tic will have to resend it.

void Tic8::handleMessage(cMessage \*msg)

{

    if (msg == timeoutEvent) {

        // If we receive the timeout event, that means the packet hasn't

        // arrived in time and we have to re-send it.

        EV << "Timeout expired, resending message and restarting timer\n";

        cMessage \*newMsg = new cMessage("tictocMsg");

        send(newMsg, "out");

        scheduleAt(simTime()+timeout, timeoutEvent);

    }

    else {  // message arrived

            // Acknowledgement received -- delete the received message and cancel

            // the timeout event.

        EV << "Timer cancelled.\n";

        cancelEvent(timeoutEvent);

        delete msg;

        // Ready to send another one.

        cMessage \*newMsg = new cMessage("tictocMsg");

        send(newMsg, "out");

        scheduleAt(simTime()+timeout, timeoutEvent);

    }

}

void Toc8::handleMessage(cMessage \*msg)

{

    if (uniform(0, 1) < 0.6) {

        EV << "\"Losing\" message.\n";

        bubble("message lost");  // making animation more informative...

        delete msg;

    }

    else {

        EV << "Sending back same message as acknowledgement.\n";

        send(msg, "out");

    }

}

## Netzwerktopologien mit mehr als zwei Knoten.

10

Now we'll make a big step: create several tic modules and connect them into a network. For now, we'll keep it simple what they do: one of the nodes generates a message, and the others keep tossing it around in random directions until it arrives at a predetermined destination node.

The NED file will need a few changes. First of all, the Txc module will need to have multiple input and output gates:

simple Txc10

{

    parameters:

        @display("i=block/routing");

    gates:

        input in[];  // declare in[] and out[] to be vector gates

        output out[];

}

network Tictoc10

{

    @display("bgb=238,186");

    submodules:

        tic[6]: Txc10 {

            @display("p=110,104");

        }

    connections:

        tic[0].out++ --> {  delay = 100ms; } --> tic[1].in++;

        tic[0].in++ <-- {  delay = 100ms; } <-- tic[1].out++;

        tic[1].out++ --> {  delay = 100ms; } --> tic[2].in++;

        tic[1].in++ <-- {  delay = 100ms; } <-- tic[2].out++;

        tic[1].out++ --> {  delay = 100ms; } --> tic[4].in++;

        tic[1].in++ <-- {  delay = 100ms; } <-- tic[4].out++;

        tic[3].out++ --> {  delay = 100ms; } --> tic[4].in++;

        tic[3].in++ <-- {  delay = 100ms; } <-- tic[4].out++;

        tic[4].out++ --> {  delay = 100ms; } --> tic[5].in++;

        tic[4].in++ <-- {  delay = 100ms; } <-- tic[5].out++;

}

void Txc10::initialize()

{

    if (getIndex() == 0) {

        // Boot the process scheduling the initial message as a self-message.

        char msgname[20];

        sprintf(msgname, "tic-%d", getIndex());

        cMessage \*msg = new cMessage(msgname);

        scheduleAt(0.0, msg);

    }

}

void Txc10::handleMessage(cMessage \*msg)

{

    if (getIndex() == 3) {

        // Message arrived.

        EV << "Message " << msg << " arrived.\n";

        delete msg;

    }

    else {

        // We need to forward the message.

        forwardMessage(msg);

    }

}

void Txc10::forwardMessage(cMessage \*msg)

{

    // In this example, we just pick a random gate to send it on.

    // We draw a random number between 0 and the size of gate `out[]'.

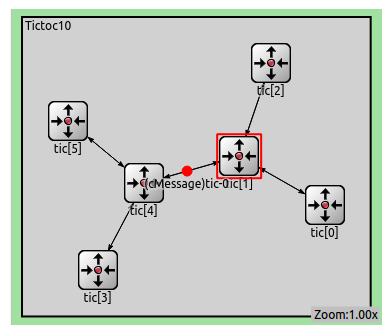
    int n = gateSize("out");

    int k = intuniform(0, n-1);

    EV << "Forwarding message " << msg << " on port out[" << k << "]\n";

    send(msg, "out", k);

}



## Wie kann ein eigenes Nachrichtenformat definiert und verwendet werden?

Our new network definition is getting quite complex and long, especially the connections section. Let's try to simplify it. The first thing we notice is that the connections always use the same delay parameter. It is possible to create types for the connections (they are called channels) similarly to simple modules. We should create a channel type which specifies the delay parameter and we will use that type for all connections in the network.

types:

        channel Channel extends ned.DelayChannel {

            delay = 100ms;

        }

 tic[0].out++ --> Channel --> tic[1].in++;

inout gate[] // two way

send(msg, "gate$o", k);

message class

In this step the destination address is no longer hardcoded tic[3] – we draw a random destination, and we'll add the destination address to the message.

The best way is to subclass [cMessage](https://www.omnetpp.org/doc/omnetpp/api/classomnetpp_1_1cMessage.html) and add destination as a data member. Hand-coding the message class is usually tedious because it contains a lot of boilerplate code, so we let OMNeT++ generate the class for us.

message TicTocMsg13

{

    int source;

    int destination;

    int hopCount = 0;

}

 // Produce source and destination addresses.

    int src = getIndex();  // our module index

    int n = getVectorSize();  // module vector size

    int dest = intuniform(0, n-2);

    if (dest >= src)

        dest++;

    char msgname[20];

    sprintf(msgname, "tic-%d-to-%d", src, dest);

    // Create message object and set source and destination field.

    TicTocMsg13 \*msg = new TicTocMsg13(msgname);

    msg->setSource(src);

    msg->setDestination(dest);

    return msg;

## Hinzufügen von Statistiken zur anschließenden Auswertung und Visualisierung.

To get an overview at runtime how many messages each node sent or received, we've added two counters to the module class: numSent and numReceived.

 private:

    long numSent;

    long numReceived;

They are set to zero and WATCH'ed in the initialize() method. Now we can use the Find/inspect objects dialog (Inspect menu; it is also on the toolbar) to learn how many packets were sent or received by the various nodes.

Adding statistics collection

The previous simulation model does something interesting enough so that we can collect some statistics. For example, you may be interested in the average hop count a message has to travel before reaching its destination.

We'll record in the hop count of every message upon arrival into an output vector (a sequence of (time,value) pairs, sort of a time series). We also calculate mean, standard deviation, minimum, maximum values per node, and write them into a file at the end of the simulation. Then we'll use tools from the OMNeT++ IDE to analyse the output files.

For that, we add an output vector object (which will record the data into Tictoc15-#0.vec) and a histogram object (which also calculates mean, etc) to the class.

 numSent = 0;

    numReceived = 0;

    WATCH(numSent);

    WATCH(numReceived);

    hopCountStats.setName("hopCountStats");

    hopCountStats.setRangeAutoUpper(0, 10, 1.5);

    hopCountVector.setName("HopCount");

Visualisierung:

The results directory in the project folder contains .vec and .sca files, which are the files that store the results in vector and scalar form, respectively. Vectors record data values as a function of time, while scalars typically record aggregate values at the end of the simulation. To open the Result Analysis tool, double click on either the .vec or the .sca files in the OMNeT++ IDE. Both files will be loaded by the Result Analysis tool. You can find the Browse Data tab at the bottom of the Result Analysis tool panel. Here you can browse results by type by switching the various tabs at the top of the tool panel, ie. Scalars, Vectors, or Histograms. By default, all results of a result type are displayed. You can filter them by the module filter to view all or some of the individual modules, or the statistic name filter to display different types of statistics, ie. mean, max, min, standard deviation, etc. You can select some or all of the individual results by highlighting them. If you select multiple results, they will be plotted on one chart. Right click and select Plot to display the figures.