Institute for the Wireless Internet of Things at Northeastern University

EECE 5155
Introduction to Simulations and the OMNeT++ Network Simulator

Dr. Francesco Restuccia
Email: f.restuccia@northeastern.edu

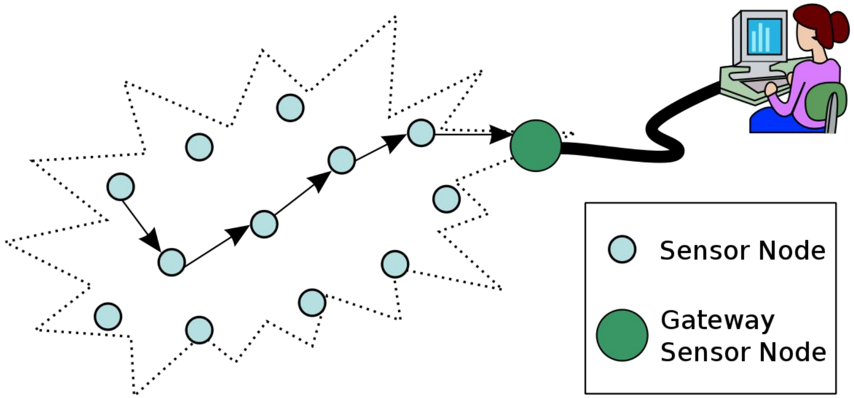
Outline

1. What is a Network Simulation, and Why do we do it?
2. Introduction to OMNeT++
3. OMNeT++ in a Nutshell
   a. What does OMNeT++ provide then?
   b. What does an OMNeT++ simulation model look like?
   c. How do I run simulations?
   d. How do I program a model in C++?
4. Working with OMNeT++: Flow Chart
5. TicToc Tutorial
   a. Getting started
   b. Enhancing the 2-node TicToc
   c. Turning into a real network
   d. Adding statistics collection
   e. Visualizing the results with the OMNeT++ IDE
6. Conclusions

How do we Evaluate Network Performance?

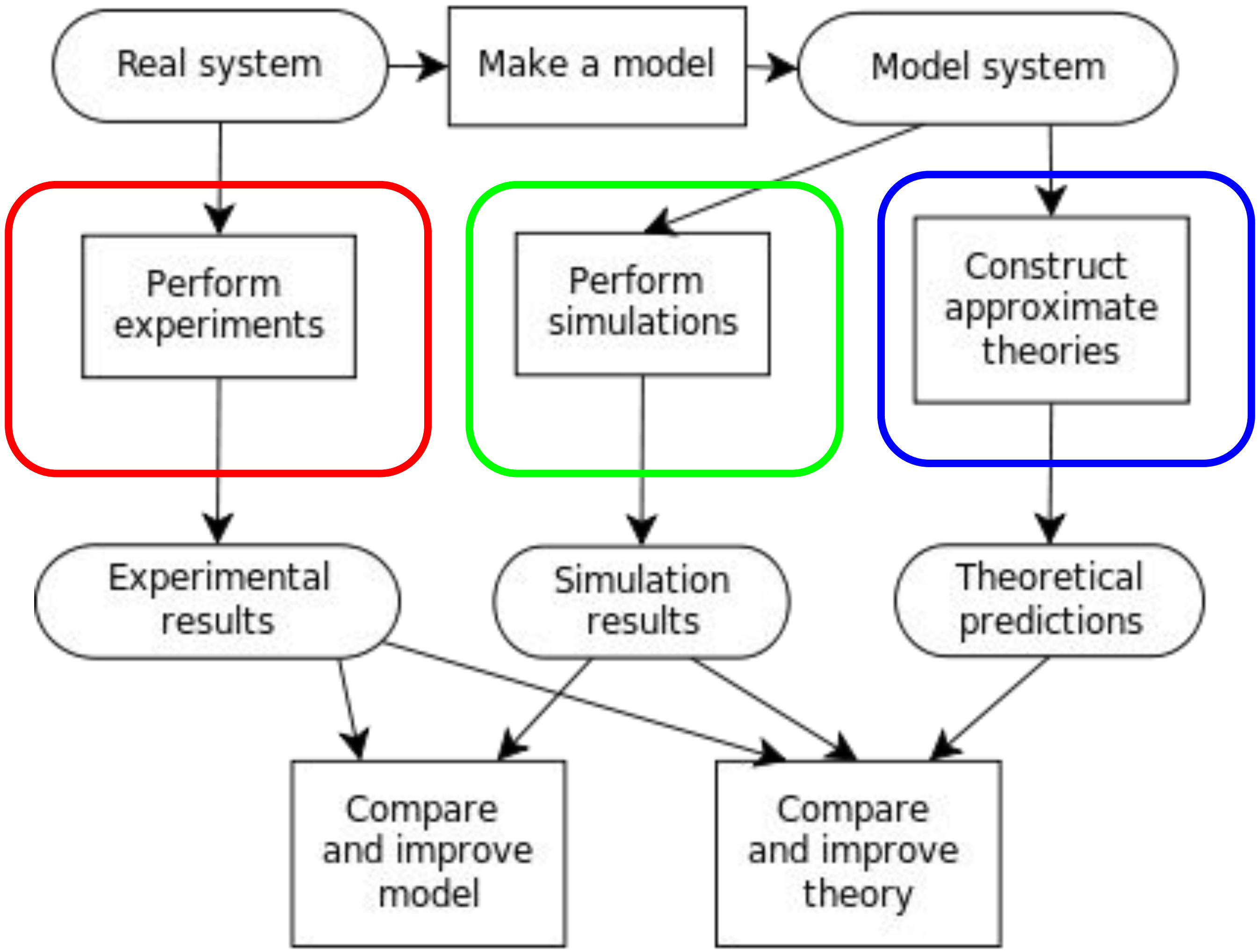
- Real systems are very complex to analyze!

- First, define performance metrics:
    - Delay
    - Energy consumption
    - Throughput
    - ….



- Three main evaluation strategies:
    - Experiments
    - Mathematical Model
    - Simulations

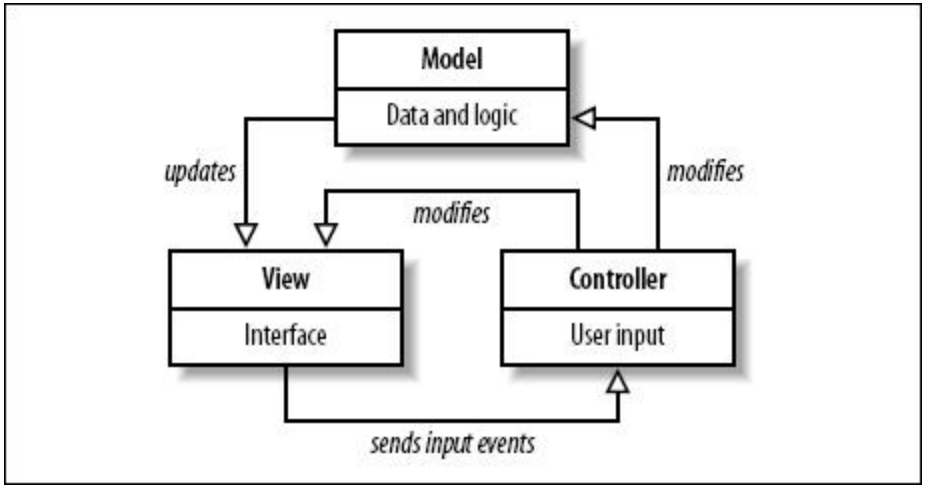
Different Performance Evaluation Tools



What is a Simulation?

- A simulation is a software program that **models** and **controls** the **behavior** of a network

- Done by computing the **interactions** between the different network **entities**



A Metaphor

- Imagine that our sensor network is a **person**
- An <u>experiment</u> is like watching one the possible individual's lives (like a **movie**)
- A <u>mathematical model</u> tells us every possible moment of the individual's life, like the **set of all possible movies**...

Very accurate (and desirable), but…

A Metaphor (2)

- Too **complex** to obtain in some cases!
- Isn't it better to watch only the **most relevant movies and the most relevant scenes of each movie**?

This way, complexity is manageable!

OK, So What is the Best Choice?

- Ideally, we should use all three!
- Each method has its pros and cons…
- Really depends on what we want to achieve!

Methodology	✅	❌
`Simulations`	Evaluate only what's necessary Desired granularity level Desired network conditions	Can take a lot of time to obtain results Few assumptions needed Depends on simulation granularity
`Experiments`	Demonstrate that system works No (or very few) assumptions	Can be impractical to realize (1000s of nodes) Depends on external conditions
`Mathematical Model`	Can show any possible evolution Usually results in short time	Can be impossible to derive Heavy dependent on model's assumptions Need validation w/ experiments and simulations

REPLICABILITY IS KEY TO THE SCIENTIFIC METHOD !

Simulation steps

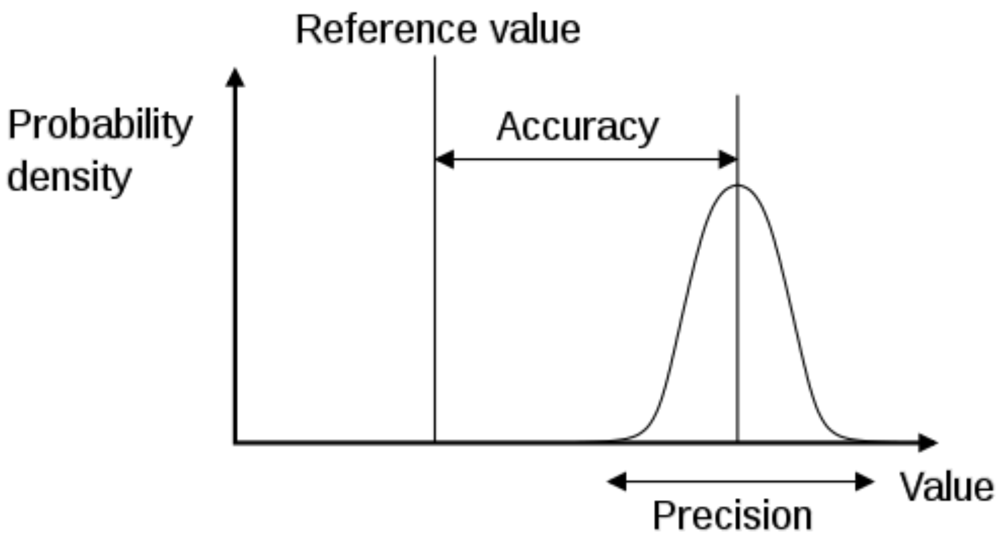
1. Define our **performance metrics of interests**
2. Define the **model** of our entities (*i.e.*, modules)
3. Define **interactions** (*i.e.*, communication) between our modules
4. Write **modular and parameterized** code that implements modules and interactions
5. Run simulations with different **parameters**
6. Obtain performance results
7. Are the results **correct**?

WHAT DOES CORRECT EVEN MEAN?

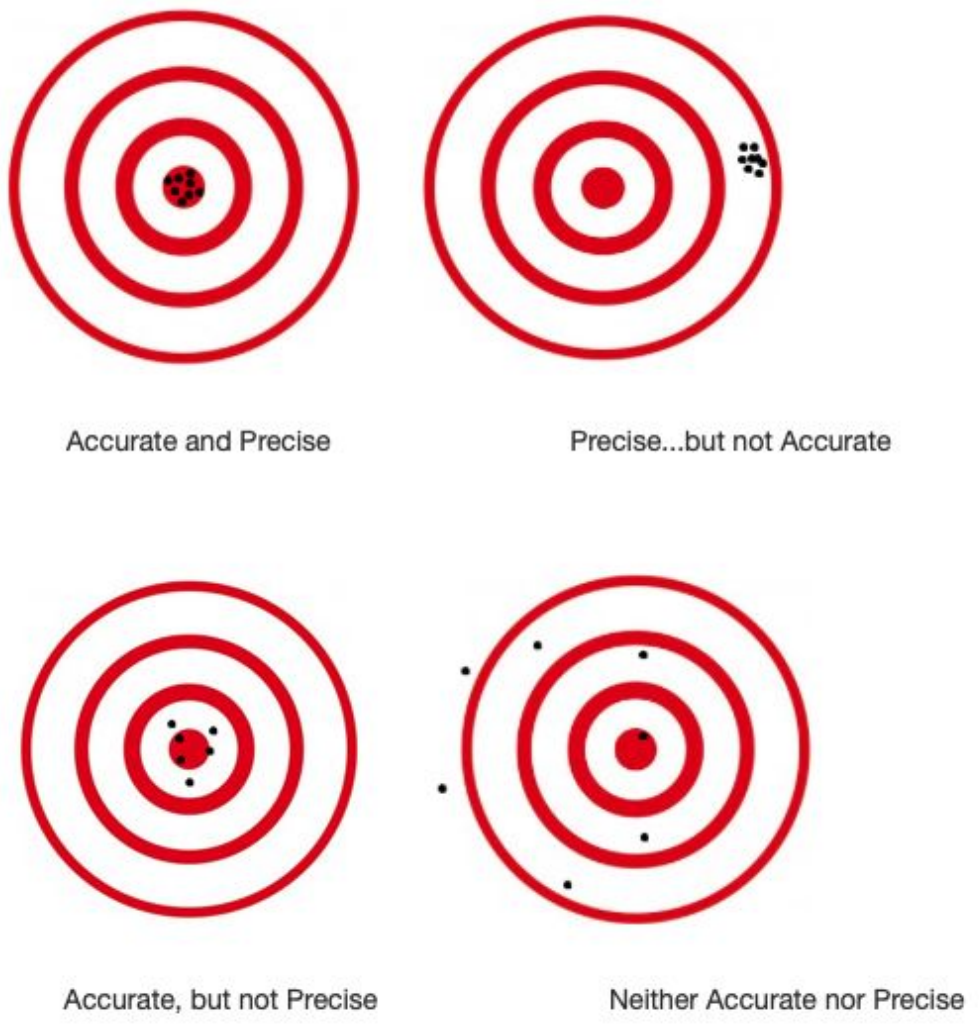
Accuracy and Precision

- Systems are usually **random (but not unpredictable!)**
- To evaluate the performance, run a "sufficient" number of runs.
- *Accuracy* is the difference between measure and "true value"
- *Precision* is a measure of ***statistical variability***

ACCURACY DOES NOT IMPLY PRECISION
PRECISION DOES NOT IMPLY ACCURACY



Accuracy and Precision



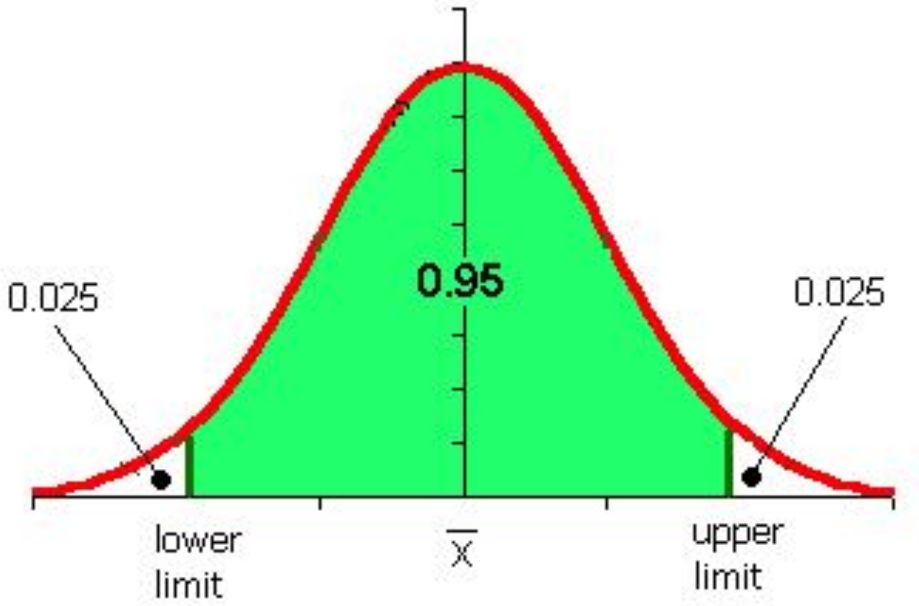
More on Results

- First of all, are your results **reasonable (accuracy)**?
  - You should have an idea of what to expect!
  - Are your results in a reasonable range?
  - Did you test your code with different inputs?
  - Maybe validate your results w/ experiments?
- How is the **precision**?
  - Run more simulations to improve precision
  - Yes, but how many?

Here comes statistics to the rescue....

Confidence Interval (CI)

- A CI is a **range of values** we are “fairly sure” our **true value** lies in
- It tells you how confident you can be that your results reflect what you would expect to find if it were possible to **run all possible simulations**
- CI are associated with **Confidence Levels** (CL) and are expressed as a percentage (for example, a 95% CL).
- A CL of 95% means that the probability of the “true value” residing in the CI is 0.95



How do we compute a CI?

$$\bar{x} = \frac{\Sigma x}{n}$$

data values

sample size

mean

$$\sigma = \sqrt{\frac{\Sigma(x - \bar{x})^2}{n}}$$

standard deviation

How do we compute a CI? (2)

standard deviation

confidence level

$$\underbrace{Z_{\alpha/2}}_{\text{confidence coefficient}} \times \frac{\sigma}{\sqrt{(n)}}$$

sample size

How do we compute a CI? (3)

90%

95%

99%

How do we compute a CI? (4)

Confidence Interval	z
80%	1.282
85%	1.440
90%	1.645
95%	1.960
99%	2.576
99.5%	2.807
99.9%	3.291

How do we compute a CI? (3)

$$\bar{x} \pm Z_{\alpha/2} \times \frac{\sigma}{\sqrt{(n)}}$$

Example

- We measure the heights of n = 40 random individuals, we get a mean of x = 175cm and a standard deviation of σ = 20cm

Confidence Interval	Z
80%	1.282
85%	1.440
90%	1.645
95%	1.960
99%	2.576
99.5%	2.807
99.9%	3.291

- We obtain the confidence interval as $175 \pm 1.960 \times 20/\sqrt{40} =$ 175cm ± 6.20cm (from **168.8cm** to **181.2cm**)

How do we run simulations?

Discrete vs. Continuous time

- A **discrete-event simulation** (**DES**) models the operation of a system as a discrete sequence of events in time
- Each event occurs at a particular instant in time and marks a change of state in the system
- Between consecutive events, **no change in the system is assumed to occur** → jump in time from one event to the next
- Continuous simulations **continuously tracks the system dynamics over time**
- Time is broken up into small time slices and the system state is updated according to the set of activities happening in the time slice
- DES typically **run much faster** than the corresponding continuous simulation and is **much easier to implement**

OMNeT++ is a DE simulator tailored for network simulations!

DES Logic

Start

- Initialize Ending Condition (EC) to FALSE
- Initialize system state variables
- Initialize Clock (usually starts at simulation time zero)
- Schedule an initial event, i.e., put some event into the Events List (EL)

"Do loop" or "while loop"

While (EC is FALSE OR no events in EL) then do the following:

- Set clock to next event time;
- Do next event and remove from the EL;
- Generate new event(s), if needed;
- Update statistics;

End

- Generate statistical report

Introduction to OMNeT++

Introduction

- Discrete event simulator
  - Hierarchically nested modules
  - Modules communicate using messages through channels
- Written in C++
  - Source code publicly available
  - Simulation model for Internet, IPv6, Mobility, etc. available
- Pros
  - Well structured, highly modular, not limited to network protocol simulations (e.g., like ns2)
- Cons
  - ~~Relatively young and only few simulation models~~ – not young anymore, mature, comes with IDE, good documentation, and simulation models are widely available!

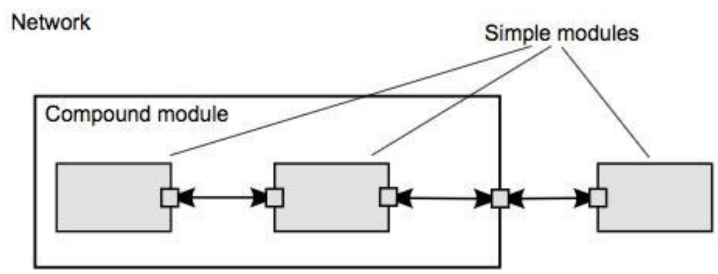
https://doc.omnetpp.org/omnetpp/manual/
https://macappstore.org/qt5/

What Does OMNeT++ Provide?

- C++ class library
  - Simulation kernel
  - Utility classes (for random number generation, statistics collection, topology discovery etc.)
  - ⇨ use to create simulation components (simple modules and channels)
- Infrastructure to assemble simulations
  - Network Description Language (NED)
  - .ini files
- Runtime user interfaces
- Eclipse-based simulation IDE for designing, running and evaluating simulations
- Extension interfaces for real-time simulation, emulation, MRIP, parallel distributed simulation, database connectivity and so on

What Does a Simulation Look Like?

- Component-based architecture
- Modules (can be combined in various way like LEGO blocks) connected through gates, and combined to form compound modules




```
// Ethernet CSMA/CD MAC
simple EtherMAC {
parameters:
   string address; // others omitted for brevity
  gates:
   input phyIn; // to physical layer or the network
   output phyOut; // to physical layer or netw
   input llcIn; // to EtherLLC or higher layer
   output llcOut; // to EtherLLC or higher layer
}
```

```
// Host with an Ethernet interface
module EtherStation {
parameters: ...
 gates: ...
  input in; // for conn. to switch/hub, etc
  output out;
 submodules:
  app:
  EtherTrafficGen;  llc:
  EtherLLC;
  mac: EtherMAC;
 connections:
 app.out --> llc.hlIn;
 app.in <-- llc.hlOut;
  llc.macIn <-- mac.llcOut;
  llc.macOout --> mac.llcIn;
  mac.phyIn <-- in;
  mac.phyOut --> out;
}
 network EtherLAN {
    ... (submodules of type EtherStation, etc) ...
  }
```

How do I Run a Simulation?

- Building the simulation program
  - `opp_makemake -deep; make`
  - `-deep` cover the whole source tree under the make directory
  - `-X` exclude directories
  - `-I` required in case subdirectories are used
- Run executable
- By default, graphical user interface, Tkenv (or Cmdenv)

```
-u Qtenv or -u Cmdenv
-c specifies the configuration in your .ini
-cmdenv-express-mode
```

```
[General]
 network = EtherLAN
 *.numStations = 20
 **.frameLength = normal(200,1400)
 **.station[0].numFramesToSend = 5000
 **.station[1-5].numFramesToSend = 1000
 **.station[*].numFramesToSend = 0
```

How do I Program a Module in C++?

- Simple modules are C++ classes
  - Subclass from cSimpleModule
  - Redefine a few virtual member functions
  - Register the new class with OMNet++ via Define_Module() macro
  - Yes, it's that simple!
- Modules communicate via **messages** (also timers are self-messages)
  - cMessage or subclass thereof
  - Messages are delivered to handleMessage(cMessage *msg)
  - Send messages to other modules: send(cMessage *msg, const char *outGateName)
  - Self-messages (i.e., timers can be scheduled): scheduleAt(simtime_t time, cMessage *msg) and cancelEvent(cMessage *msg) to cancel the timer

How do I Program a Module in C++? (2)

- cMessage data members
  - name, length, kind, etc.
  - encapsulate(cMessage *msg), decapsulate() to facilitate protocol simulations
- .msg file for user-defined message types
  - OMNet++ opp_msgc to generate C++ class: _m.h and _m.cc
  - Support inheritance, composition, array members, etc.

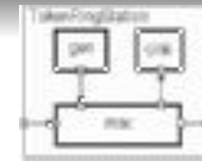
```
message NetworkPacket {
 fields:
   int srcAddr;  int
   destAddr;
}
```

How do I Program a Module in C++? (3)

- Other cSimpleModule virtual member functions
  - initialize(), finish()
  - Reading NED parameters: par(const char *paramName)
  - Typically done in initialize() and values are stored in data members of the module class

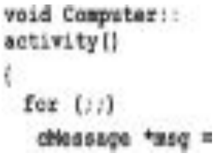
OMNeT++ Development Flow

1. An OMNeT++ model is build from modules that communicate through messages. You need to map your system into a hierarchy of communicating modules.

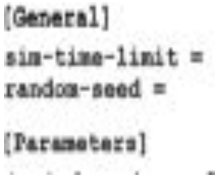


2. Define the model structure in the NED language. You can use a text editor or in the graphical editor of the Eclipse-based OMNeT++ Simulation IDE.

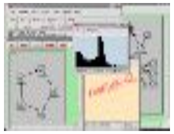
3. The simple modules have to be programmed in C++, using the simulation kernel and class library.



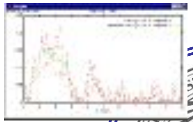
4. Provide a suitable omnetpp.ini to hold OMNeT++ configuration and parameters to your model. A config file can describe several simulation runs with different parameters.



5. Build the simulation program and run it. You'll link the code with the OMNeT++ simulation kernel. There are command line and interactive, graphical user interface



6. Simulation results are written into output vector and output scalar files. You can use the Analysis Tool in the Simulation IDE to visualize them. Result files are text-based, so you can also process them with R, Matlab or other tools.



TicToc Tutorial

- Short tutorial to OMNeT++ guides you through an example of modeling and simulation
- Showing you along the way some of the commonly used OMNeT++ features
- Based on the Tictoc example simulation: samples/tictoc – http://www.omnetpp.org/doc/omnetpp/tictoc-tutorial/
- Many other tutorials and examples are available!

Getting Started

- Starting point
  - "network" that consists of two nodes (cf. simulation of telecommunications network)
  - One node will create a message and the two nodes will keep passing the same packet back and forth
  - Nodes are called “tic” and “toc”
- Here are the steps you take to implement your first simulation from scratch:
  1. Create a working directory called tictoc, and cd to this directory
  2. Describe your example network by creating a topology file (.NED)
  3. Implement the functionality (.CPP)
  4. Create the Makefile (not needed if using GUI)
  5. Compile and link the simulation
  6. Create the configuration file omnetpp.ini
  7. Start the executable using GUI
  8. Press the Run button on the toolbar to start the simulation
  9. You can play with slowing down the animation or making it faster
  10. You can exit the simulation program …

Topology

```
// This file is part of an OMNeT++/OMNEST simulation example.
// Copyright (C) 2003 Ahmet Sekercioglu
// Copyright (C) 2003-2008 Andras Varga
// This file is distributed WITHOUT ANY WARRANTY. See the file
// `license' for details on this and other legal
matters.

simple Txc1 {
  gates:
    input in;
    output
    out;
}
// Two instances (tic and toc) of Txc1 connected both ways.
// Tic and toc will pass messages to one another.
network Tictoc1 {
  submodules:
    tic:
    Txc1;
    toc:
    Txc1;
  connections:
}   tic.out --> { delay = 100ms; } -->
    toc.in;  tic.in <-- { delay = 100ms; }
    <-- toc.out;
```

Txc1 is a simple module type, i.e., atomic on NED level and implemented in C++.
Txc1 has one input gate named in, and one output gate named out .

Tictoc1 is a network assembled from two submodules tic and toc (instances from a module type called Txc1).
Connection: tic's output gate (out) to toc's input gate and vice versa with propagation delay 100ms.

Implement the Functionality

Txc1 simple module represented as C++ class Txc1; subclass from cSimpleModule.

Need to redefine two methods from cSimpleModule: initialize() and handleMessage().

```
#include <string.h>
#include <omnetpp.h>
using namespace omnetpp;
class Txc1 : public cSimpleModule {
  protected: // The following redefined virtual function holds the algorithm.
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};
Define_Module(Txc1); // The module class needs to be registered with OMNeT++

void Txc1::initialize() {
  // Initialize is called at the beginning of the simulation. To bootstrap the tic-toc-tic-toc process, one of the modules needs
  // to send the first message. Let this be `tic‘. Am I Tic or Toc?
  if (strcmp("tic", getName()) == 0) {
    // create and send first message on gate "out". "tictocMsg" is an arbitrary string which will be the name of the message object.
    cMessage *msg = new cMessage("tictocMsg");
    send(msg, "out");
  }
}
void Txc1::handleMessage(cMessage *msg) {
  // The handleMessage() method is called whenever a message arrives at the module.
  // Here, we just send it to the other module, through gate `out'.
  // Because both `tic' and `toc' does the same, the message will bounce between the two.
  send(msg, "out");
}
```

Create cMessage and send it to out on gate `out`.

Send it back.

Create Makefile, Compile and Link Simulation

- Create Makefile: `opp_makemake`
  - This command should have now created a Makefile in the working directory tictoc.
- Compile and link the simulation: `make`

If you start the executable now, it will complain that it cannot find the file `omnetpp.ini`

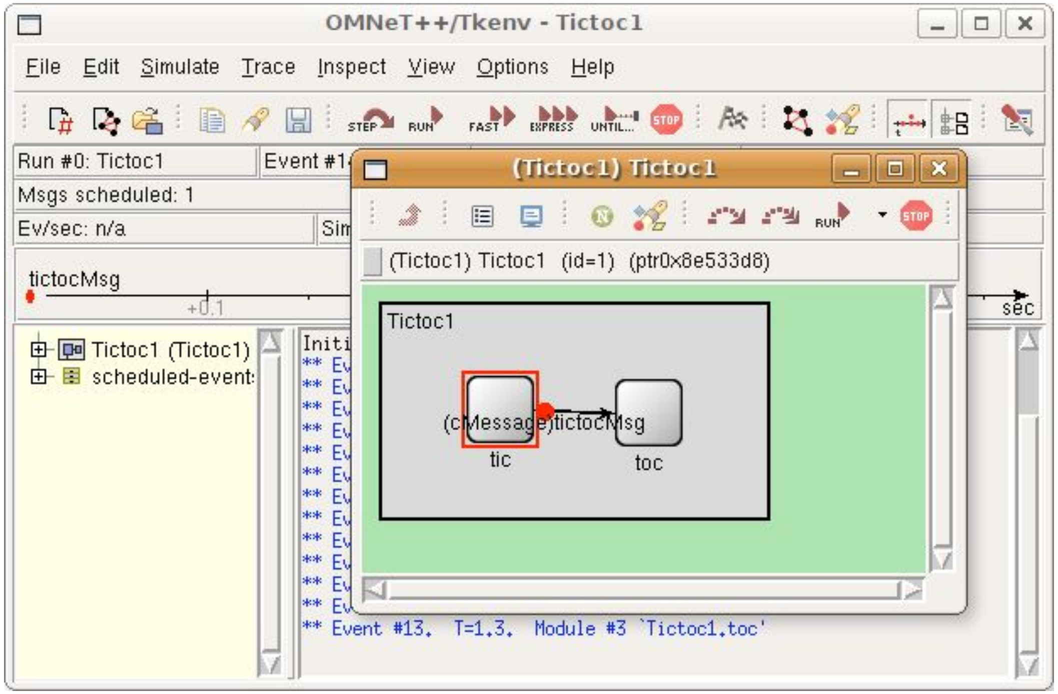
Create “omnetpp.ini”

- … tells the simulation program which network you want to simulate
- … you can pass parameters to the model
- … explicitly specify seeds for the random number generators etc.

```
# This file is shared by all tictoc simulations.
# Lines beginning with `#' are comments
[General]
# nothing here
[Config Tictoc1]
 network = Tictoc1 # this line is for Cmdenv, Tkenv will still let you choose from a dialog
...
[Config Tictoc4]
 network = Tictoc4
 tictoc4.toc.limit = 5
...
[Config Tictoc7]
 network = Tictoc7
 # argument to exponential() is the mean; truncnormal() returns values from
 # the normal distribution truncated to nonnegative values
 tictoc7.tic.delayTime = exponential(3s)
 tictoc7.toc.delayTime = truncnormal(3s,1s)
```

Start the Executable, Run, Play, Exit

- `./tictoc` shall start the simulation environment
- Hit Run button to start the actual simulation
- Play around, e.g., fast forward, stop, etc.
- Hit File->Exit to close the simulation



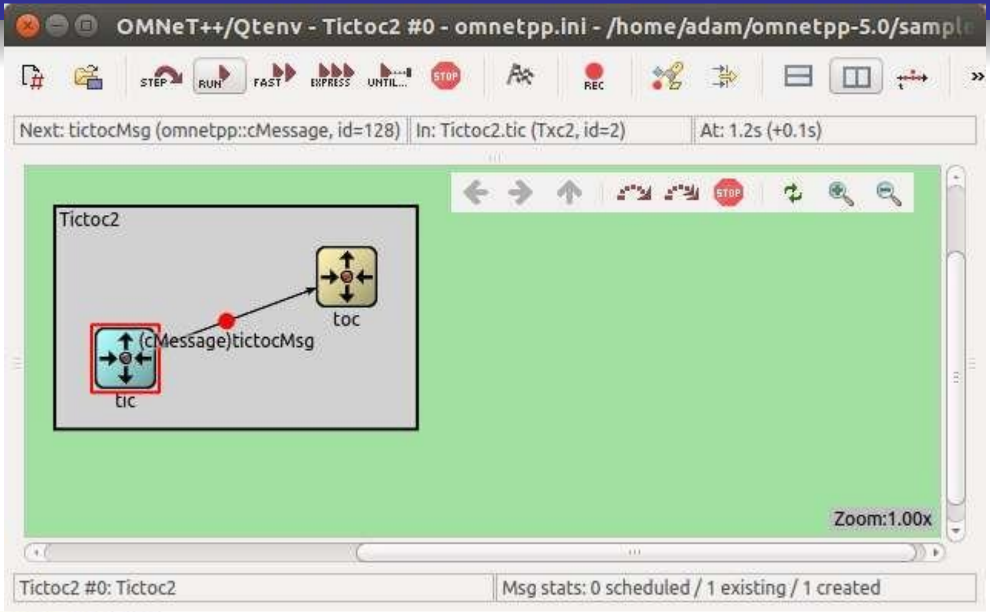
Enhancing the 2-node TicToc

- Step 2: Refining the graphics, and adding debugging output
- Step 3: Adding state variables
- Step 4: Adding parameters
- Step 5: Using inheritance
- Step 6: Modeling processing delay
- Step 7: Random numbers and parameters
- Step 8: Timeout, cancelling timers
- Step 9: Retransmitting the same message

Step 2: Refining the Graphics, Adding Debugger Output

```
simple Txc2
{
    parameters:
        @display("i=block/routing"); // add a default icon
    gates:
        input in;
        output out;
}

//
// Make the two module look a bit different with colorization effect.
// Use cyan for `tic', and yellow for `toc'.
//
network Tictoc2
{
    submodules:
        tic: Txc2 {
            parameters:
                @display("i=,cyan"); // do not change the icon (first arg of i=) just colorize it
        }
        toc: Txc2 {
            parameters:
                @display("i=,gold"); // here too
        }
    connections:
        tic.out --> {  delay = 100ms; } --> toc.in;
        tic.in <-- {  delay = 100ms; } <-- toc.out;
}
```

Step 2: Refining the Graphics, Adding Debugger Output



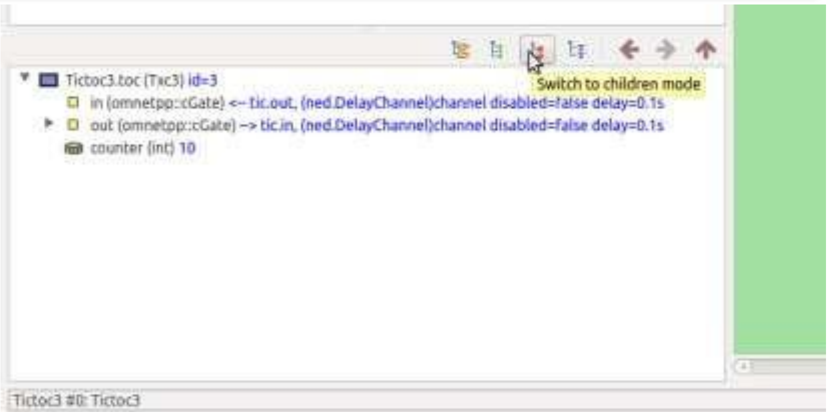
Debug messages

EV << "Sending initial message\n";

EV << "Received message `" << msg->name() << "', sending it out again\n";

```
** Event #95   t=9.5   Tictoc2.toc (Txc2, id=3)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.toc: Received message `tictocMsg', sending it out again
** Event #96   t=9.6   Tictoc2.tic (Txc2, id=2)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.tic: Received message `tictocMsg', sending it out again
** Event #97   t=9.7   Tictoc2.toc (Txc2, id=3)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.toc: Received message `tictocMsg', sending it out again
** Event #98   t=9.8   Tictoc2.tic (Txc2, id=2)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.tic: Received message `tictocMsg', sending it out again
** Event #99   t=9.9   Tictoc2.toc (Txc2, id=3)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.toc: Received message `tictocMsg', sending it out again
** Event #100   t=10   Tictoc2.tic (Txc2, id=2)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.tic: Received message `tictocMsg', sending it out again
** Event #101   t=10.1   Tictoc2.toc (Txc2, id=3)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.toc: Received message `tictocMsg', sending it out again
** Event #102   t=10.2   Tictoc2.tic (Txc2, id=2)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.tic: Received message `tictocMsg', sending it out again
** Event #103   t=10.3   Tictoc2.toc (Txc2, id=3)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.toc: Received message `tictocMsg', sending it out again
** Event #104   t=10.4   Tictoc2.tic (Txc2, id=2)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.tic: Received message `tictocMsg', sending it out again
** Event #105   t=10.5   Tictoc2.toc (Txc2, id=3)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.toc: Received message `tictocMsg', sending it out again
** Event #106   t=10.6   Tictoc2.tic (Txc2, id=2)  on tictocMsg (omnetpp::cMessage, id=128)
INFO (Txc2)Tictoc2.tic: Received message `tictocMsg', sending it out again
```

Step 3: Adding State Variables




```
#include <stdio.h> #include <string.h> #include <omnetpp.h> using name
class Txc3 : public cSimpleModule {
  private:
    int counter; // Note the counter here
  protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};
Define_Module(Txc3);
void Txc3::initialize() {
counter = 10;
  /* The WATCH() statement below will let you examine the variable under Tkenv. After doing a few steps in the
simulation,  double-click either `tic' or `toc', select the Contents tab in the dialog that pops up, and you'll find "counter"
in the list. */  WATCH(counter);
  if (strcmp("tic", getName()) == 0) {
    EV << "Sending initial message\n";
    cMessage *msg = new cMessage("tictocMsg");
    send(msg, "out");
  }
}
void Txc3::handleMessage(cMessage *msg) {
  counter--; // Increment counter and check value.
  if (counter==0) { /* If counter is zero, delete message. If you run the model, you'll find that the simulation will stop at this
point  with the message "no more events". */
    EV << getName() << "'s counter reached zero, deleting message\n";
    delete msg;
  } else { EV << getName() << "'s counter is " << counter << ", sending back message\n"; send(msg, "out"); }
}
```

Step 4: Adding Parameters

- Module parameters have to be declared in the NED file: numeric, string, bool, or xml

```
simple Txc4 {
 parameters:
    // whether the module should send out a message on initialization
    bool sendMsgOnInit = default(false);
    int limit = default(2); // another parameter with a default value
    @display("i=block/routing");
 gates:
```

- Modify the C++ code to read the parameter in initialize(), and assign it to the counter.

```
counter = par("limit");
```

- Now, we can assign the parameters in the NED file or from omnetpp.ini. Assignments in the NED file take precedence.

```
network Tictoc4 {
   submodules:
      tic: Txc4 {
         parameters:
            sendMsgOnInit = true;
         @display("i=,cyan"); }
      toc: Txc4 { // note that we leave toc's limit unbound here  parameters:
            sendMsgOnInit = false;
         @display("i=,gold"); }
   connections:
```

Step 4: Adding Parameters (2)

- … and the other in omnetpp.ini:
  - Tictoc4.toc.limit = 5
- Note that because omnetpp.ini supports wildcards, and parameters assigned from NED files take precedence over the ones in omnetpp.ini, we could have used
  - Tictoc4.t*c.limit = 5
- or
  - Tictoc4.*.limit = 5
- or even
  - **.limit = 5
- with the same effect. (The difference between * and ** is that * will not match a dot and ** will.)

Enhancing the 2-node TicToc

- Step 2: Refining the graphics, and adding debugging output
- Step 3: Adding state variables
- Step 4: Adding parameters
- Step 5: Using inheritance
- Step 6: Modeling processing delay
- Step 7: Random numbers and parameters
- Step 8: Timeout, cancelling timers
- Step 9: Retransmitting the same message

Step 5: Using Inheritance

- Take a closer look:

tic and toc differs only in their parameter values and their display string => use inheritance

```
simple Txc5
{
    parameters:
        bool sendMsgOnInit = default(false);
        int limit = default(2);
        @display("i=block/routing");
    gates:
        input in;
        output out;
}
```

```
simple Tic5 extends Txc5
{
    parameters:
        @display("i=,cyan");
        sendMsgOnInit = true;   // Tic modules should send a message on init
}
```

```
simple Toc5 extends Txc5
{
    parameters:
        @display("i=,gold");
        sendMsgOnInit = false;  // Toc modules should NOT send a message on init
}
```

```
network Tictoc5
{
    submodules:
        tic: Tic5;  // the limit parameter is still unbound here. We will get it from the ini file
        toc: Toc5;
    connections:
```

Enhancing the 2-node TicToc

- Step 2: Refining the graphics, and adding debugging output
- Step 3: Adding state variables
- Step 4: Adding parameters
- Step 5: Using inheritance
- Step 6: Modeling processing delay
- Step 7: Random numbers and parameters
- Step 8: Timeout, cancelling timers
- Step 9: Retransmitting the same message

Step 6: Modeling Processing Delay

- Hold messages for some time before sending it back ⇨ timing is achieved by the module sending a message to itself (i.e., self-messages)

```
class Txc6 : public cSimpleModule
{
  private:
    cMessage *event; // pointer to the event object which we'll use for timing
    cMessage *tictocMsg; // variable to remember the message until we send it back

  public:
```

- We "send" the self-messages with the scheduleAt() function

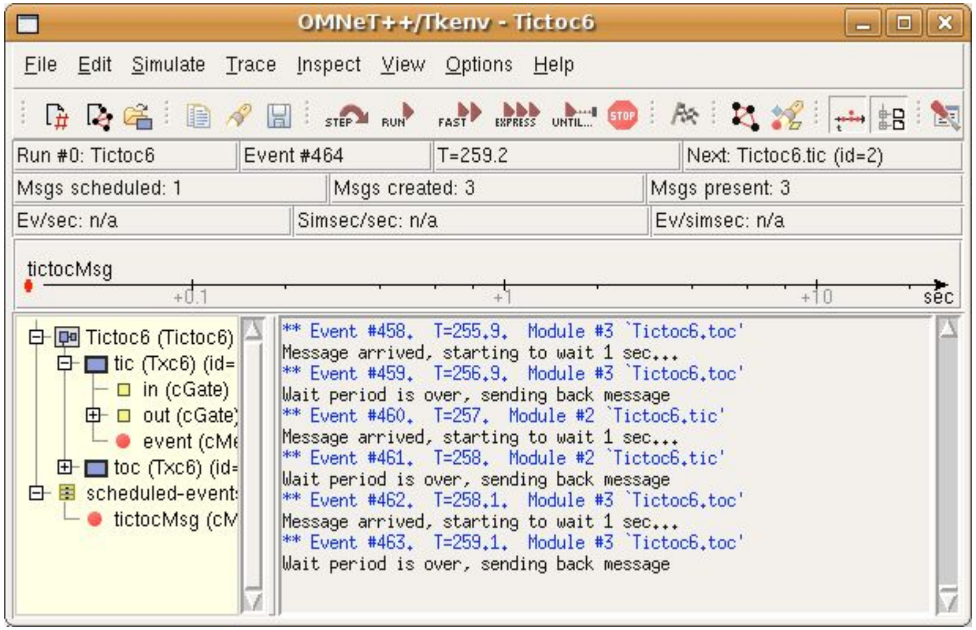
  `scheduleAt(simTime()+1.0, event);`

- handleMessage(): differentiate whether a new message has arrived via the input gate or the self-message came back (timer expired)

  `if (msg==event)` or `if (msg->isSelfMessage())`

Step 6: Modeling Processing Delay (2)

```
void Txc6::handleMessage(cMessage *msg) {

  if (msg==event) {
     EV << "Wait period is over, sending back message”;
     send(tictocMsg, "out");
     tictocMsg = NULL;
  } else {
     // If the message we received is not our self-message
     // be the tic-toc message arriving from our partner
     // 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);

  }
}
```



Enhancing the 2-node TicToc

- Step 2: Refining the graphics, and adding debugging output
- Step 3: Adding state variables
- Step 4: Adding parameters
- Step 5: Using inheritance
- Step 6: Modeling processing delay
- Step 7: Random numbers and parameters
- Step 8: Timeout, cancelling timers
- Step 9: Retransmitting the same message

Step 7: Random Numbers and Parameters

- Change the delay from 1s to a random value :

```
// The "delayTime" module parameter can be set to values like
// "exponential(5)" (tictoc7.ned, omnetpp.ini), and then here
// we'll get a different delay every time.
simtime_t delay = par("delayTime");

EV << "Message arrived, starting to wait " << delay << " secs...\n";
tictocMsg = msg;
scheduleAt(simTime()+delay, event);
```

- "Lose" (delete) the packet with a small (hardcoded) probability:

```
if (uniform(0,1) < 0.1) {
    EV << "\"Losing\" message\n";
    bubble("message lost");
    delete msg;
}
```

- Assign seed and the parameters in omnetpp.ini: [General]
  seed-0-mt=532569 # or any other 32-bit value
  Tictoc7.tic.delayTime = exponential(3s)
  Tictoc7.toc.delayTime = truncnormal(3s,1s)

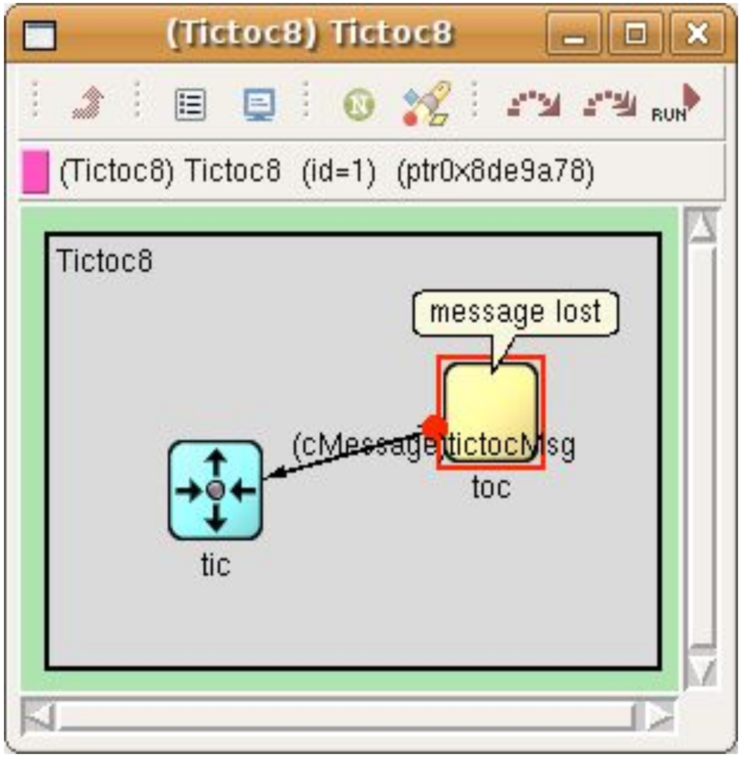
Enhancing the 2-node TicToc

- Step 2: Refining the graphics, and adding debugging output
- Step 3: Adding state variables
- Step 4: Adding parameters
- Step 5: Using inheritance
- Step 6: Modeling processing delay
- Step 7: Random numbers and parameters
- Step 8: Timeout, cancelling timers
- Step 9: Retransmitting the same message

Step 8: Timeout, Canceling Timers

- Transform our model into a **stop-and-wait simulation**

```
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 *msg = new cMessage("tictocMsg");
      send(msg, "out");
      scheduleAt(simTime()+timeout, timeoutEvent);
   } else { // message arrived
      // Acknowledgement received -- delete the stored message
      // and cancel the timeout event.
      EV << "Timer cancelled.\n";
      cancelEvent(timeoutEvent);

      // Ready to send another one.
      cMessage *msg = new cMessage("tictocMsg");
      send(msg, "out");
      scheduleAt(simTime()+timeout,
      timeoutEvent);
   }
}
```



Enhancing the 2-node TicToc

- Step 2: Refining the graphics, and adding debugging output
- Step 3: Adding state variables
- Step 4: Adding parameters
- Step 5: Using inheritance
- Step 6: Modeling processing delay
- Step 7: Random numbers and parameters
- Step 8: Timeout, cancelling timers
- Step 9: Retransmitting the same message

Step 9: Retransmitting the Same Message

- Keep a copy of the original packet so that we can re-send it without the need to build it again:
  - Keep the original packet and send only copies of it
  - Delete the original when toc's acknowledgement arrives
  - Use message sequence number to verify the model
- generateNewMessage() and sendCopyOf()
  - avoid handleMessage() growing too large

Step 9: Retransmitting the Same Message (2)

```
cMessage *Tic9::generateNewMessage() {
   // Generate a message with a different name every time.
   char msgname[20];
   sprintf(msgname, "tic-%d", ++seq);
   cMessage *msg = new cMessage(msgname);
   return msg;
}

void Tic9::sendCopyOf(cMessage *msg) {
   // Duplicate message and send the copy.
   cMessage *copy = (cMessage *) msg->dup();
   send(copy, "out");
}
```

```
void Tic9::initialize()
{
    // Initialize variables.
    seq = 0;
    timeout = 1.0;
    timeoutEvent = new cMessage("timeoutEvent");

    // Generate and send initial message.
    EV << "Sending initial message\n";
    message = generateNewMessage();
    sendCopyOf(message);
    scheduleAt(simTime()+timeout, timeoutEvent);
}

void Tic9::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";
        sendCopyOf(message);
        scheduleAt(simTime()+timeout, timeoutEvent);
    }
    else // message arrived
    {
        // Acknowledgement received!
        EV << "Received: " << msg->getName() << "\n";
        delete msg;

        // Also delete the stored message and cancel the timeout event.
        EV << "Timer cancelled.\n";
        cancelEvent(timeoutEvent);
        delete message;

        // Ready to send another one.
        message = generateNewMessage();
        sendCopyOf(message);
        scheduleAt(simTime()+timeout, timeoutEvent);
    }
}
```

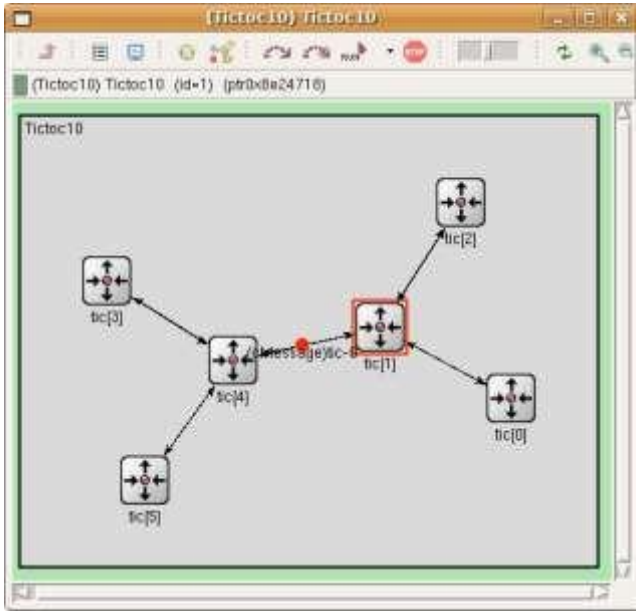
Turning it into a Real-world Network

- Step 10: More than two nodes
- Step 11: Channels and inner type definitions
- Step 12: Using two-way connections
- Step 13: Defining our message class

Step 10: More than Two Nodes

```
simple Txc10
{
    parameters:
        @display("i=block/routing");
    gates:
        input in[];  // declare in[] and out[] to be vector gates
        output out[];
}
```

```
network Tictoc10
{
    submodules:
        tic[6]: Txc10;
    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++;
}
```



Step 10: More than Two Nodes (2)

- tic[0] will generate the message to be sent around
  - done in initialize()
  - getIndex() function which returns the index of the module
- forwardMessage(): invoke from handleMessage() whenever a message arrives

```
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);
}
```

- When the message arrives at tic[3], its handleMessage() will delete the message

```
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);
    }
}
```

Step 11: Channels and Inner Type Definitions

- Network definition is getting quite complex and long, e.g., connections section

```
network Tictoc11
{
    types:
        channel Channel extends ned.DelayChannel {
            delay = 100ms;
        }
    submodules:
```

- **Note**: built-in DelayChannel (import ned.DelayChannel)

```
    connections:
        tic[0].out++ --> Channel --> tic[1].in++;
        tic[0].in++ <-- Channel <-- tic[1].out++;

        tic[1].out++ --> Channel --> tic[2].in++;
        tic[1].in++ <-- Channel <-- tic[2].out++;

        tic[1].out++ --> Channel --> tic[4].in++;
        tic[1].in++ <-- Channel <-- tic[4].out++;

        tic[3].out++ --> Channel --> tic[4].in++;
        tic[3].in++ <-- Channel <-- tic[4].out++;

        tic[4].out++ --> Channel --> tic[5].in++;
        tic[4].in++ <-- Channel <-- tic[5].out++;
}
```

Step 12: Using Two-way Connections

- Each node pair is connected with two connections => OMNeT++ 4 supports 2-way connections

```
simple Txc12
{
    parameters:
        @display("i=block/routing");
    gates:
        inout gate[];  // declare two way connections
}
```

- The new connections section would look like this:

```
    connections:
        tic[0].gate++ <--> Channel <--> tic[1].gate++;
        tic[1].gate++ <--> Channel <--> tic[2].gate++;
        tic[1].gate++ <--> Channel <--> tic[4].gate++;
        tic[3].gate++ <--> Channel <--> tic[4].gate++;
        tic[4].gate++ <--> Channel <--> tic[5].gate++;
}
```

Step 12: Using Two-way Connections (2)

- We have modified the gate names => some modifications to the C++ code.

```
void Txc12::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 `gate[]'.
    int n = gateSize("gate");
    int k = intuniform(0,n-1);

    EV << "Forwarding message " << msg << " on gate[" << k << "]\n";
    // $o and $i suffix is used to identify the input/output part of a two way gate
    send(msg, "gate$o", k);
}
```

- **Note:** The special $i and $o suffix after the gate name allows us to use the connection's two direction separately.

Step 13: Defining our Message Class

- Destination address is no longer hardcoded tic[3] – add destination address to message
- Subclass cMessage: tictoc13.msg

```
message TicTocMsg13
    {  fields:
        int source;
        int destination;
        int hopCount =
        0;
    }
```

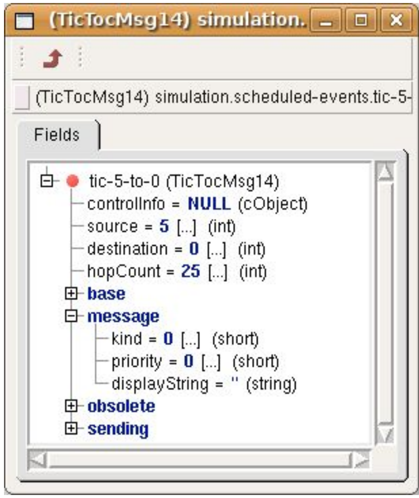
- opp_msgc is invoked and it generates tictoc13_m.h and tictoc13_m.cc
- Include tictoc13_m.h into our C++ code, and we can use TicTocMsg13 as any other class

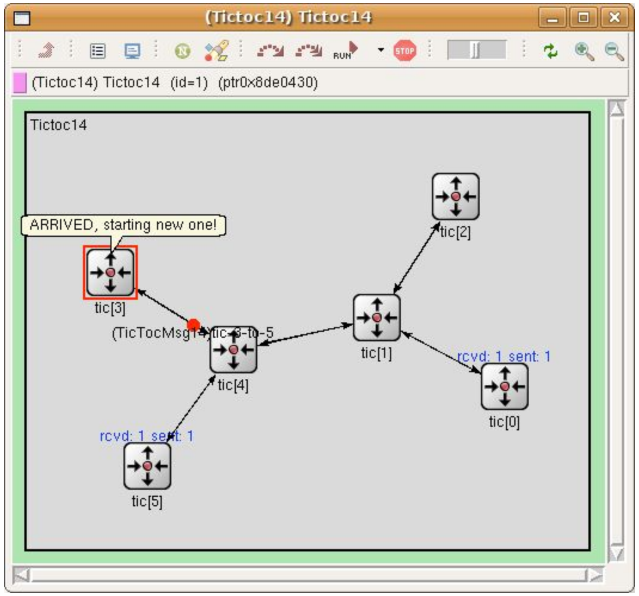
```
#include "tictoc13_m.h"

TicTocMsg13 *msg = new
TicTocMsg13(msgname);  msg->setSource(src);
msg->setDestination(dest);
return msg;
```

Step 13: Defining our Message Class (2)

- Use dynamic cast instead of plain C-style cast ((TicTocMsg13 *)msg) which is not safe

```
void Txc13::handleMessage(cMessage *msg) {
  TicTocMsg13 *ttmsg = check_and_cast<TicTocMsg13 *>(msg);

  if (ttmsg->getDestination()==getIndex()) {
    // Message arrived.
    EV << "Message " << ttmsg << " arrived after " << ttmsg->getHopCount() << " hops.\n";
    bubble("ARRIVED, starting new one!");
    delete ttmsg;

    // Generate another one.
    EV << "Generating another message: ";
    TicTocMsg13 *newmsg =
    generateMessage();  EV << newmsg << endl;
    forwardMessage(newmsg);
  } else {
    // We need to forward the message.
    forwardMessage(ttmsg);
  }
}
```





Step 13: Defining our Message Class (3)

```
TicTocMsg13 *Txc13::generateMessage()
{
    // Produce source and destination addresses.
    int src = getIndex();   // our module index
    int n = size();      // 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;
}

void Txc13::forwardMessage(TicTocMsg13 *msg)
{
    // Increment hop count.
    msg->setHopCount(msg->getHopCount()+1);

    // Same routing as before: random gate.
    int n = gateSize("gate");
    int k = intuniform(0,n-1);

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

Adding Statistics Collection

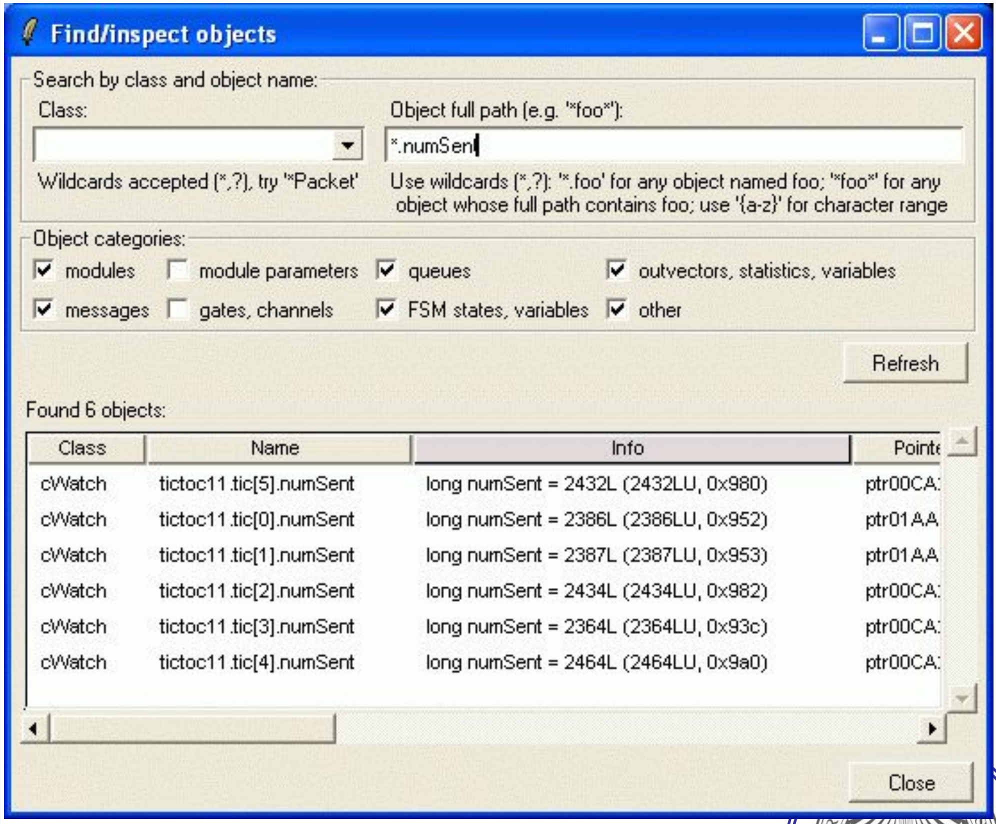
- Step 14: Displaying the number of packets sent/received

- Step 15: Adding statistics collection

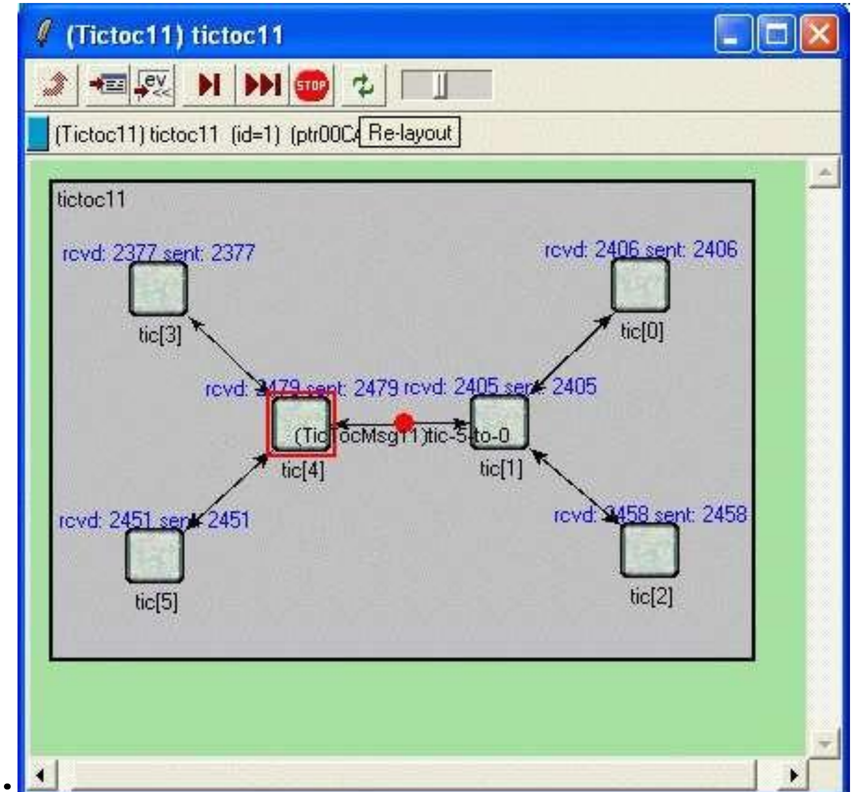
Step 14: Displaying the Number of Packets Sent/Received

- Add two counters to the module class: numSent and numReceived
- Set to zero and WATCH'ed in the initialize() method
- Use the Find/inspect objects dialog (Inspect menu; it is also on the toolbar)

```
void Txc14::initialize()
{
    // Initialize variables
    numSent = 0;
    numReceived = 0;
    WATCH(numSent);
    WATCH(numReceived);

    // Module 0 sends the first message
    if (getIndex()==0)
    {
        // Boot the process scheduling the initial message as a self-message.
        TicTocMsg14 *msg = generateMessage();
        scheduleAt(0.0, msg);
    }
}

void Txc14::handleMessage(cMessage *msg)
{
    TicTocMsg14 *ttmsg = check_and_cast<TicTocMsg14 *>(msg);

    if (ttmsg->getDestination()==getIndex())
    {
        // Message arrived
        int hopcount = ttmsg->getHopCount();
        EV << "Message " << ttmsg << " arrived after " << hopcount << " hops.\n";
        numReceived++;
        delete ttmsg;
        bubble("ARRIVED, starting new one!");

        // Generate another one.
        EV << "Generating another message: ";
        TicTocMsg14 *newmsg = generateMessage();
        EV << newmsg << endl;
        forwardMessage(newmsg);
        numSent++;

        if (ev.isGUI())
            updateDisplay();
    }
    else
    {
        // We need to forward the message.
        forwardMessage(ttmsg);
    }
}
```



- This info appears above the module icons using the t= display string tag




```
if (ev.isGUI())
   updateDisplay();

void Txc11::updateDisplay(){
   char buf[40];
   sprintf(buf, "rcvd: %ld sent: %ld", numReceived, numSent);
   getDisplayString().setTagArg("t",0,buf);
}
```

Step 15: Adding Statistics Collection

- OMNeT++ simulation kernel: omnetpp.ini

```
record-eventlog = true
```

- Example: average hopCount a message has to travel before reaching its destination
  - 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)
  - Calculate mean, standard deviation, minimum, maximum values per node, and write them into a file at the end of the simulation
  - Use off-line tools to analyse the output files

Step 15: Adding Statistics Collection (2)

- Output vector object (which will record the data into omnetpp.vec) and a histogram object (which also calculates mean, etc)

```
class Txc15 : public cSimpleModule {
  private:
    long numSent;
    long numReceived;
    cLongHistogram hopCountStats;
    cOutVector hopCountVector;
```

- Upon message arrival, update statistics within handleMessage()

```
hopCountVector.record(hopcount);
hopCountStats.collect(hopcount);
```

- hopCountVector.record() call writes the data into Tictoc15-0.vec (will be deleted each time the simulation is restarted)

Step 15: Adding Statistics Collection (2)

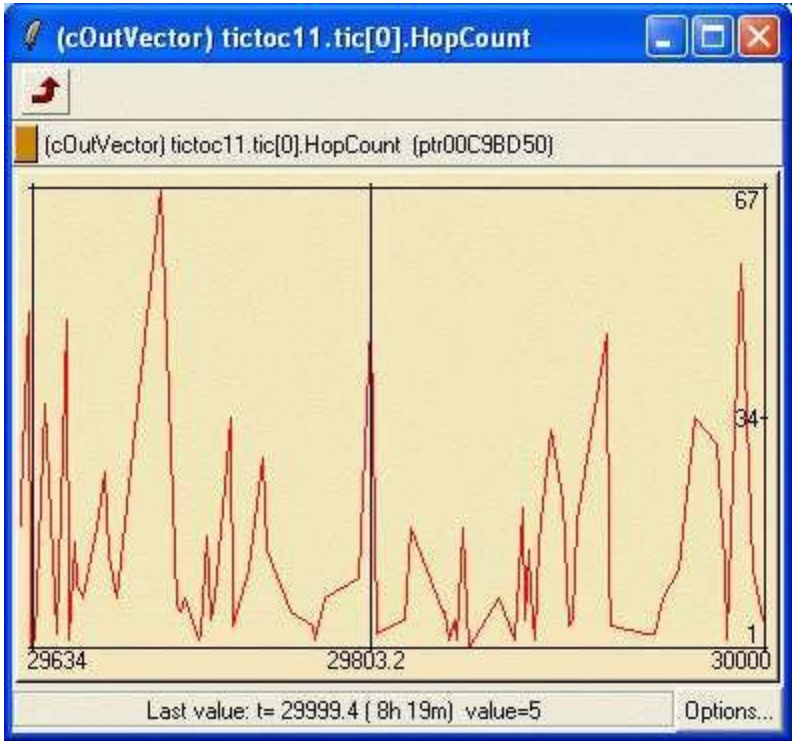
- Scalar data (histogram object) have to be recorded manually, in the finish() function

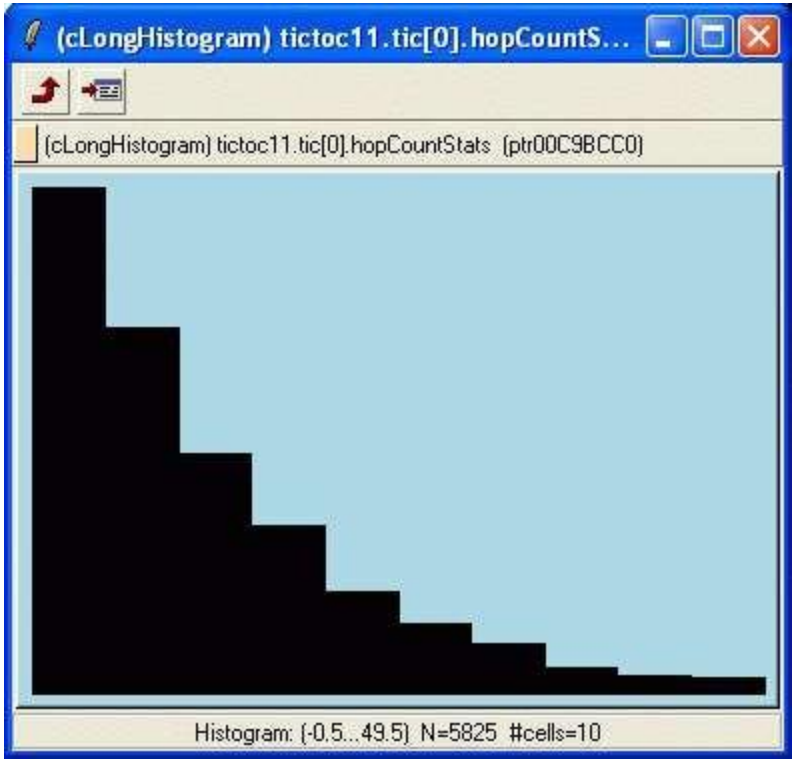
```
void Txc15::finish() {
    // This function is called by OMNeT++ at the end of the
    simulation.  EV << "Sent: " << numSent << endl;
    EV << "Received: " << numReceived << endl;

    EV << "Hop count, min:    " << hopCountStats.getMin() << endl;
    EV << "Hop count, max:    " << hopCountStats.getMax() << endl;
    EV << "Hop count, mean:   " << hopCountStats.getMean() << endl;
    EV << "Hop count, stddev: " << hopCountStats.getStddev() <<
                                   endl;

    recordScalar("#sent", numSent);
    recordScalar("#received",numReceived);
    hopCountStats.recordAs("hop count");
}
```

- recordScalar() calls in the code below write into the Tictoc15-0.sca file
- Tictoc15-0.sca not deleted between simulation runs; new data are just appended – allows to jointly analyze data from several simulation runs

Step 15: Adding Statistics Collection (3)

You can also view the data during simulation. In the **module inspector's Contents page** you'll find the hopCountStats and hopCountVector objects, and you can open their inspectors (double-click). They will be initially empty -- run the simulation in Fast (or even Express) mode to get enough data to be displayed. After a while you'll get something like this:





Step 16: Adding Statistics Collection (4)

- OMNeT++ 5 provides an additional mechanism to record values and events
  - Any model can emit 'signals' that can carry a value or an object
  - The model writer just have to decide what signals to emit, what data to attach to them and when to emit them
- The end user can attach 'listeners' to these signals that can process or record these data items
  - This way the model code does not have to contain any code that is specific to the statistics collection
  - The end user can freely add additional statistics without even looking into the C++ code
- We can safely remove all statistic related variables from our module
  - No need for the cOutVector and cLongHistogram classes either
  - Need only a single signal that carries the hopCount of the message at the time of message arrival at the destination

Step 16: Adding Statistics Collection (5)

- Define our signal: arrivalSignal as identifier

```
class Txc16 : public cSimpleModule
{
  private:
        simsignal_t arrivalSignal;

  protected:
```

- We must register all signals before using them

```
void Txc16::initialize()
{
    arrivalSignal = registerSignal("arrival");
    // Module 0 sends the first message
    if (getIndex()==0)
```

- Emit our signal, when the message has arrived to the destination node. finish() method can be deleted!

```
void Txc16::handleMessage(cMessage *msg)
{
    TicTocMsg16 *ttmsg = check_and_cast<TicTocMsg16 *>(msg);

    if (ttmsg->getDestination()==getIndex())
    {
        // Message arrived
        int hopcount = ttmsg->getHopCount();
        // send a signal
        emit(arrivalSignal, hopcount);

        EV << "Message " << ttmsg << " arrived after " << hopcount << " hops.\n";
```

Step 16: Statistics Collection w/o Modifying Your Model

- Declare signals in the NED file

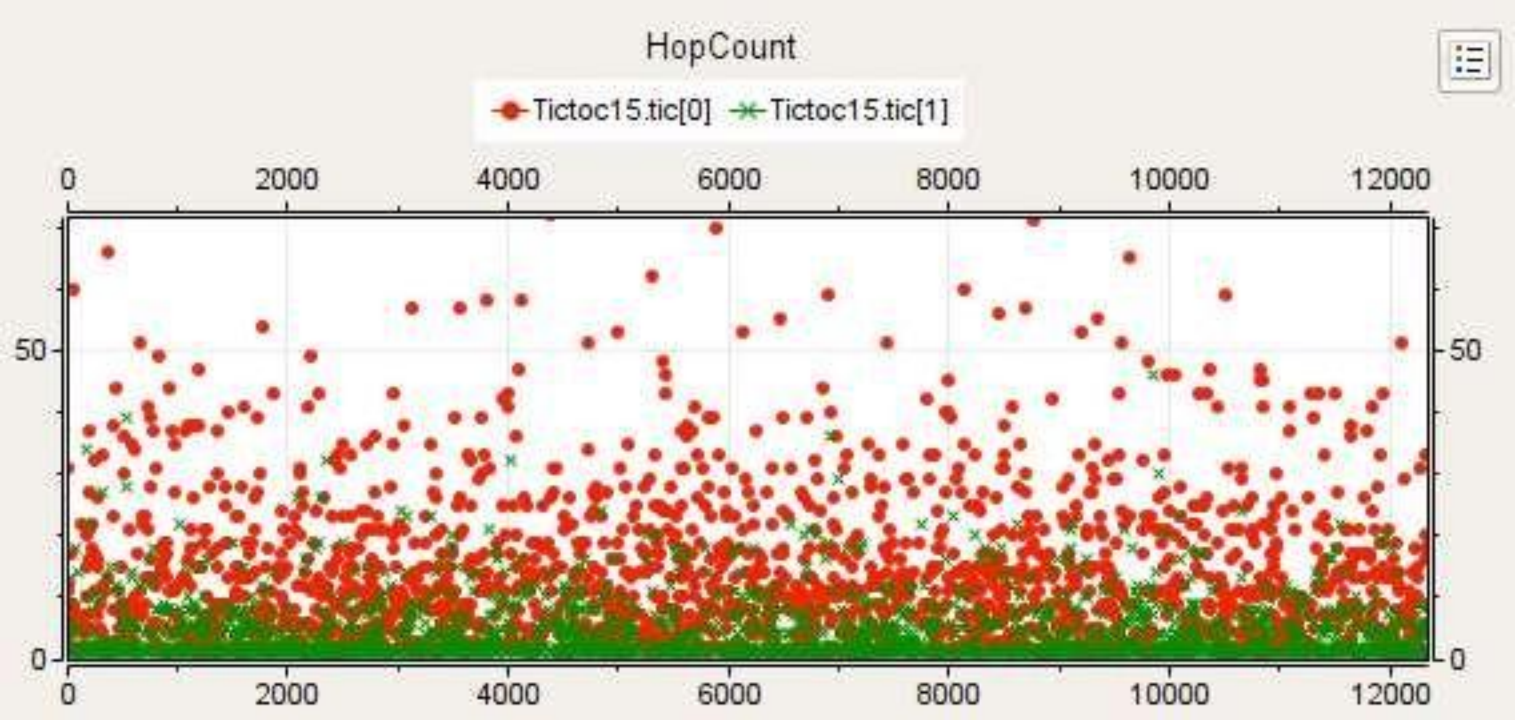
```
simple Txc16
{
    parameters:
        @signal[arrival](type="int");
        @statistic[hopCount](title="hop count"; source="arrival"; record=vector,stats; interpolationmode=none);

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

- Configuration via ini file

```
[Config Tictoc16]
network = Tictoc16
**.tic[1].hopCount.result-recording-modes = +histogram
**.tic[0..2].hopCount.result-recording-modes = -vector
```

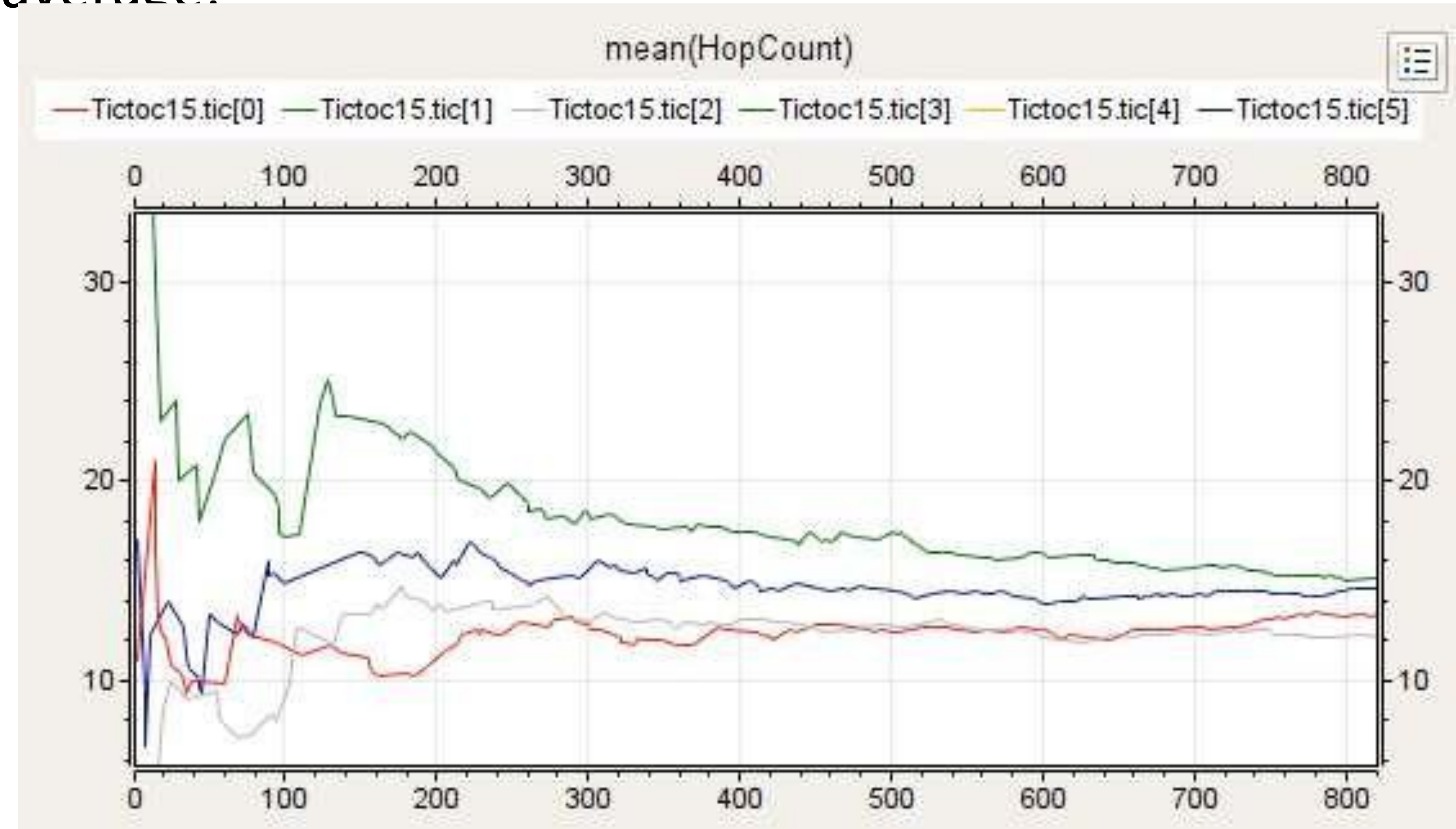
Visualizing Results

- OMNet++ IDE can be used to analyze the results.
- Our last model records the hopCount of a message each time the message reaches its destination.
- The following plot shows these vectors for nodes 0 and 1.



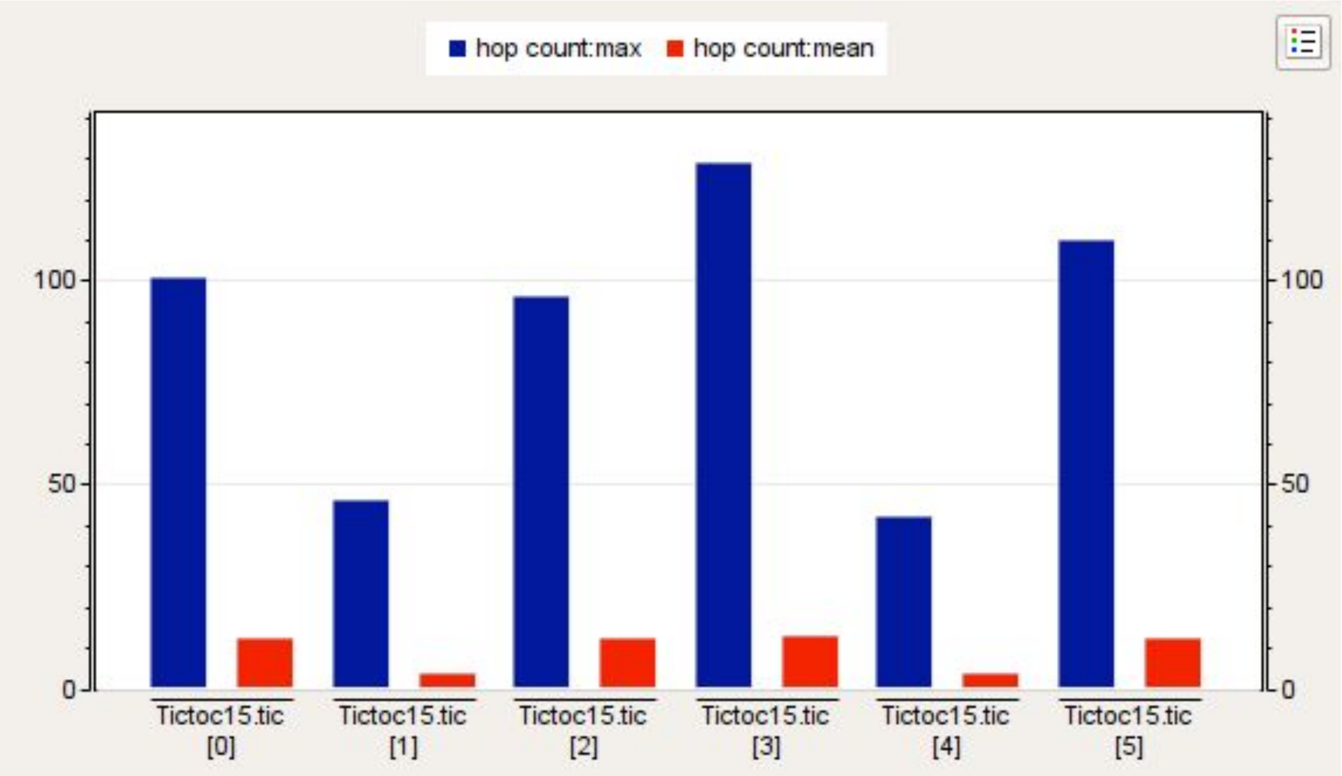
Visualizing Results (2)

- If we apply a mean operation we can see how the hopCount in the different nodes converge to an average:



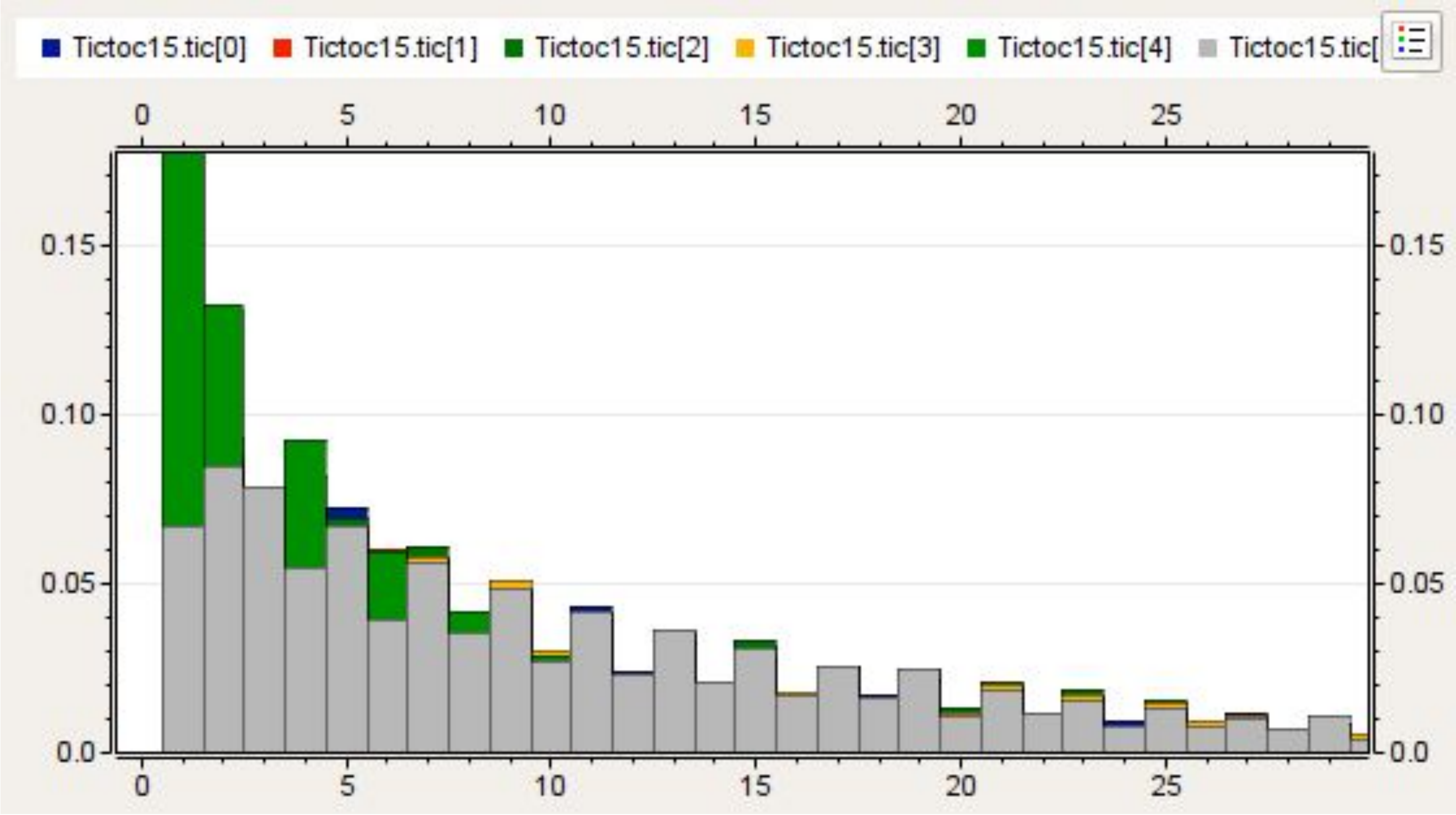
Visualizing Results (3)

- Mean and the maximum of the hopCount of the messages for each destination node
  - based on the scalar data recorded at the end of the simulation.



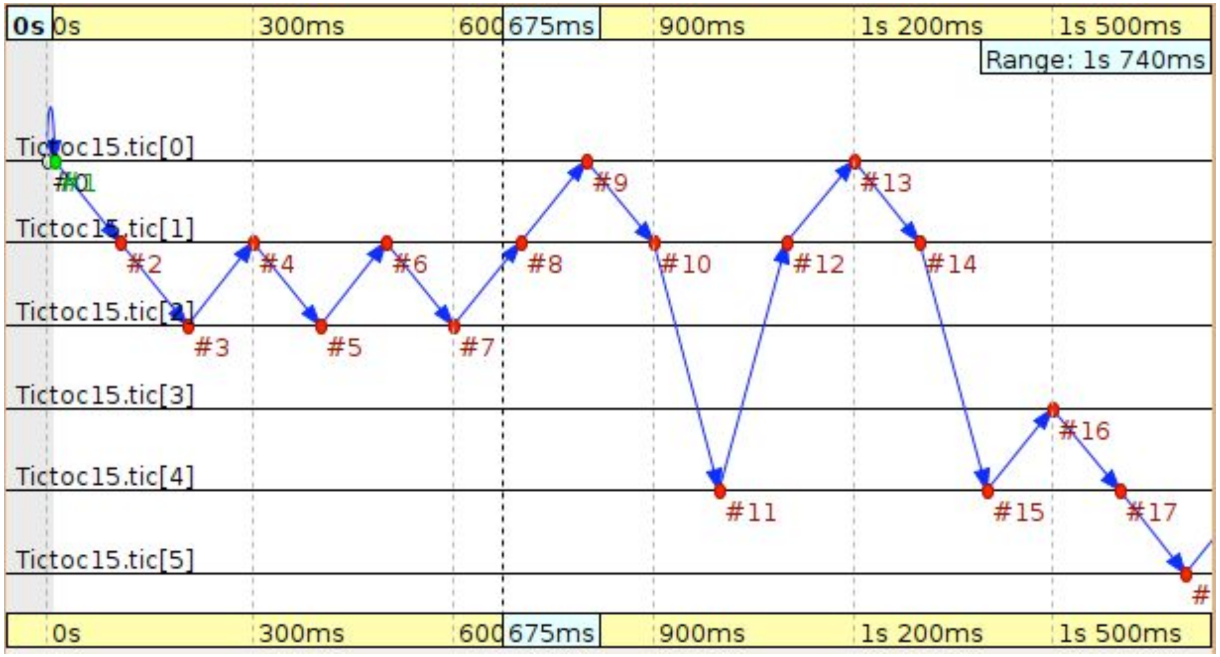
Visualizing Results (4)

- Histogram of hopCount's distribution



Visualizing Results (5)

- OMNeT++ simulation kernel can record the message exchanges during the simulation => event log file => analyzed with Sequence Chart tool
- E.g., message is routed between the different nodes in the network



Useful Concepts for the Homework

Manipulating parameters from C++

- cModule* c = getModuleByPath("myModule");
- double x = ((double) c->par("myPar"));

Conclusions

- OMNeT++: discrete event simulation system
- OMNeT++ is a
  - public-source,
  - component-based,
  - modular and open-architecture simulation environment
  - with strong GUI support and
  - an embeddable simulation kernel
- OMNeT++ 5.2 IDE (Eclipse)
  - http://www.omnetpp.org/doc/omnetpp/UserGuide.pdf
- http://www.omnetpp.org/documentation