ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

07 - Simulation, Part II

Stefan Rührup

University of Freiburg
Computer Networks and Telematics

Summer 2009

CoNe Freiburg

IIF
INSTITUT FÜR INFORMATIK FREIBURG

Overview

- **In the first part of this chapter:**
  - Discrete-event simulation

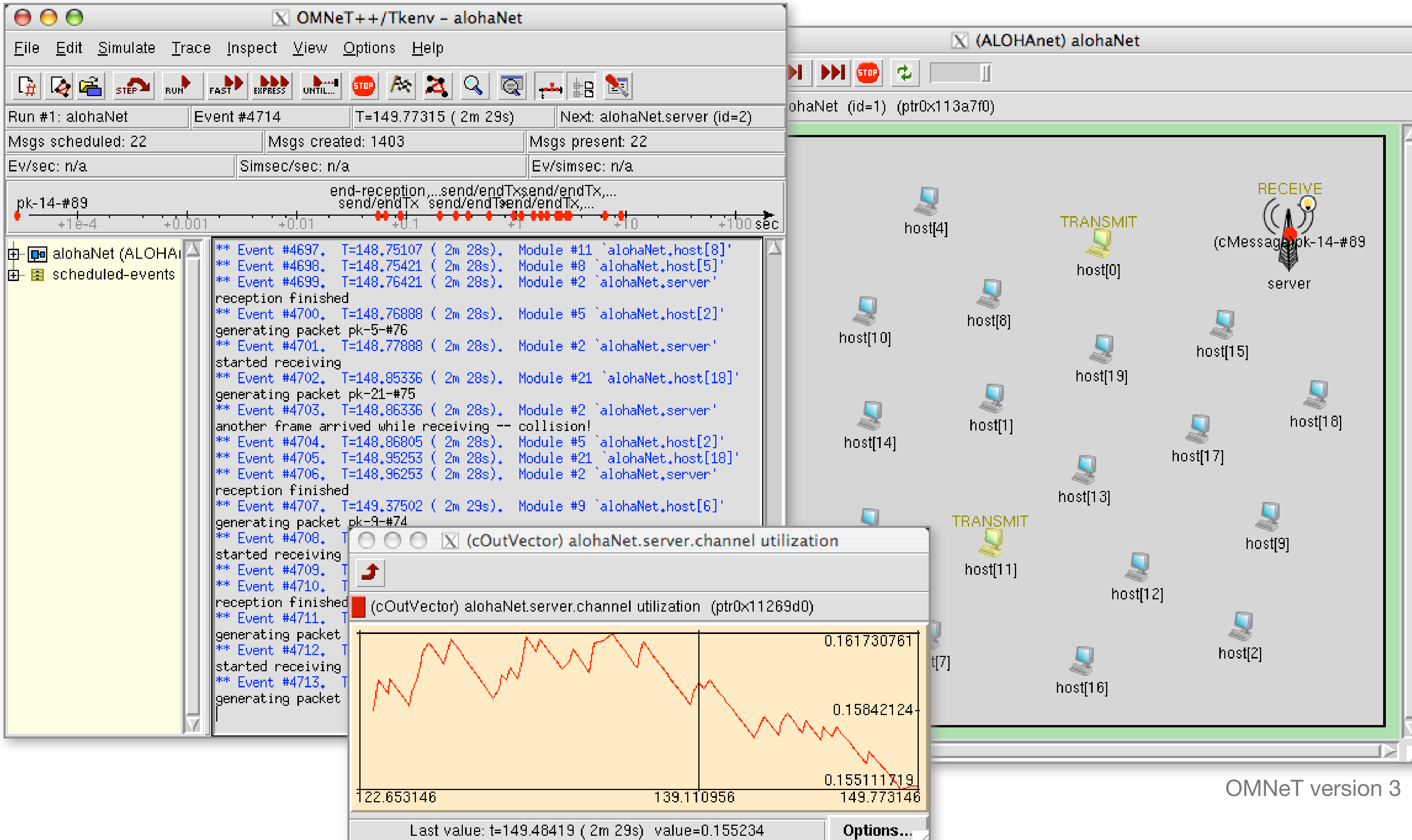
- **In this part:**
  - Network simulation
  - The network simulator OMNeT++
  - Simulation models for wireless networks

OMNeT++

- **The simulation environment OMNeT++**
  - Discrete event simulator
  - Component-based
  - Provides the basic tools to write simulations
    - simulation kernel (event processing)
    - utility classes (RNG, statistics collection)
  - Public-source (free for academic use)

- OMNeT++ is a **general purpose tool** and not specifically designed for network simulations. Components for network simulations are provided by **frameworks**.

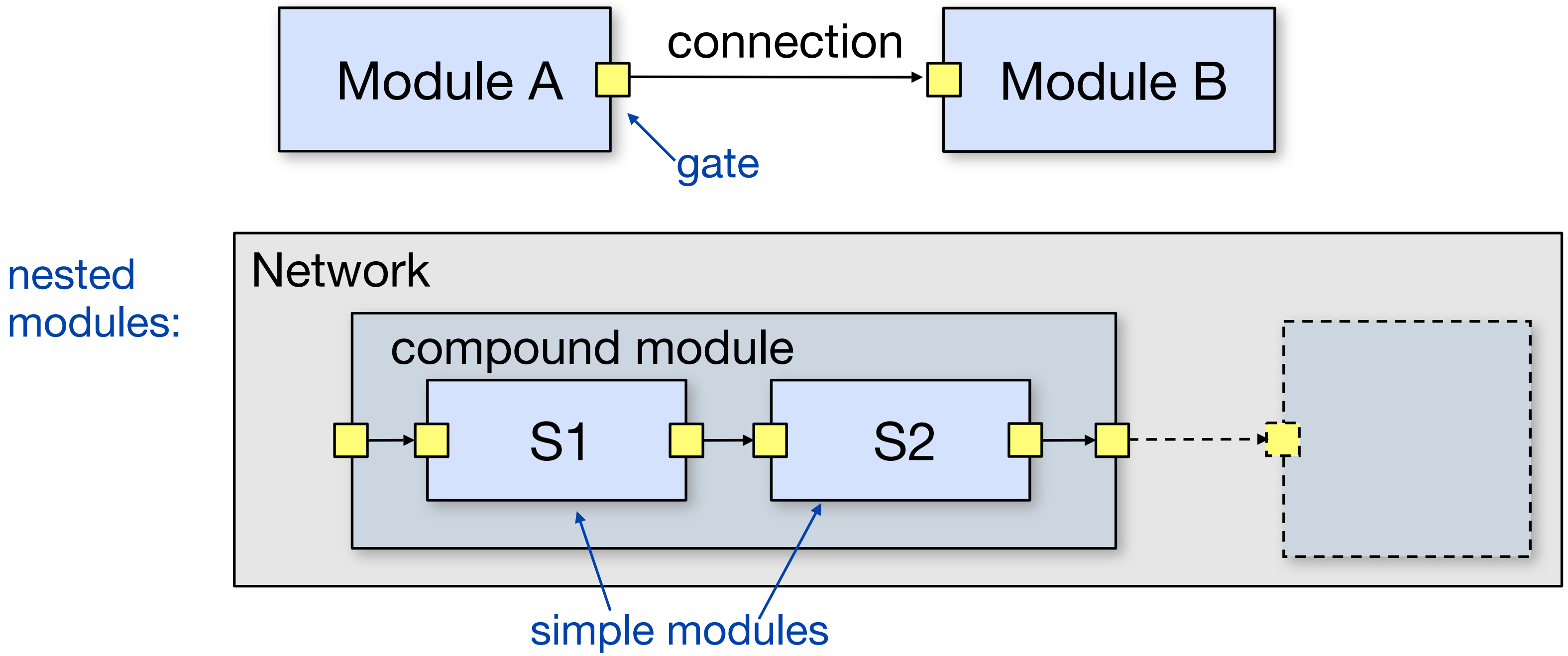
The User Interface



OMNeT version 3

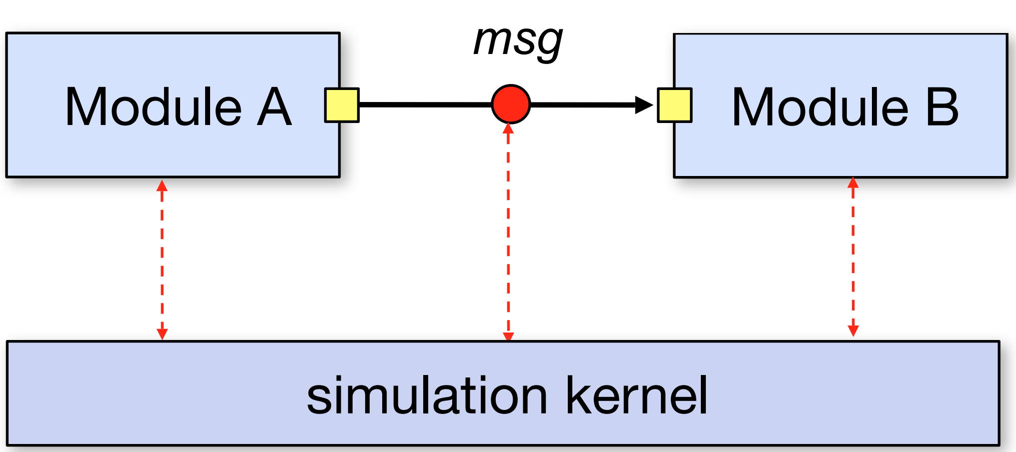
Basic Principles (1)

- A simulation model consists of **modules** (Modules are communicating FSMs)
- Modules communicate by passing messages over connections (links)



Basic Principles (2)

- Modules implement application-specific behaviour
- Modules are C++ objects
- Connections are defined using the NED (network topology description) language
- Modules communicate by exchanging messages. The reception of a message is an event



Why do we need gates?

- Gates are well-defined interfaces
- Functionality inside the module is independent of the connections
  → Modules can be treated as black boxes
  → Modules are exchangable (e.g. layer of a protocol stack)
- Modules send messages to outgoing gates
- ...and also directly to other modules (can be useful when simulating a wireless medium where connections are created dynamically)

How to Write a Simulation (1)

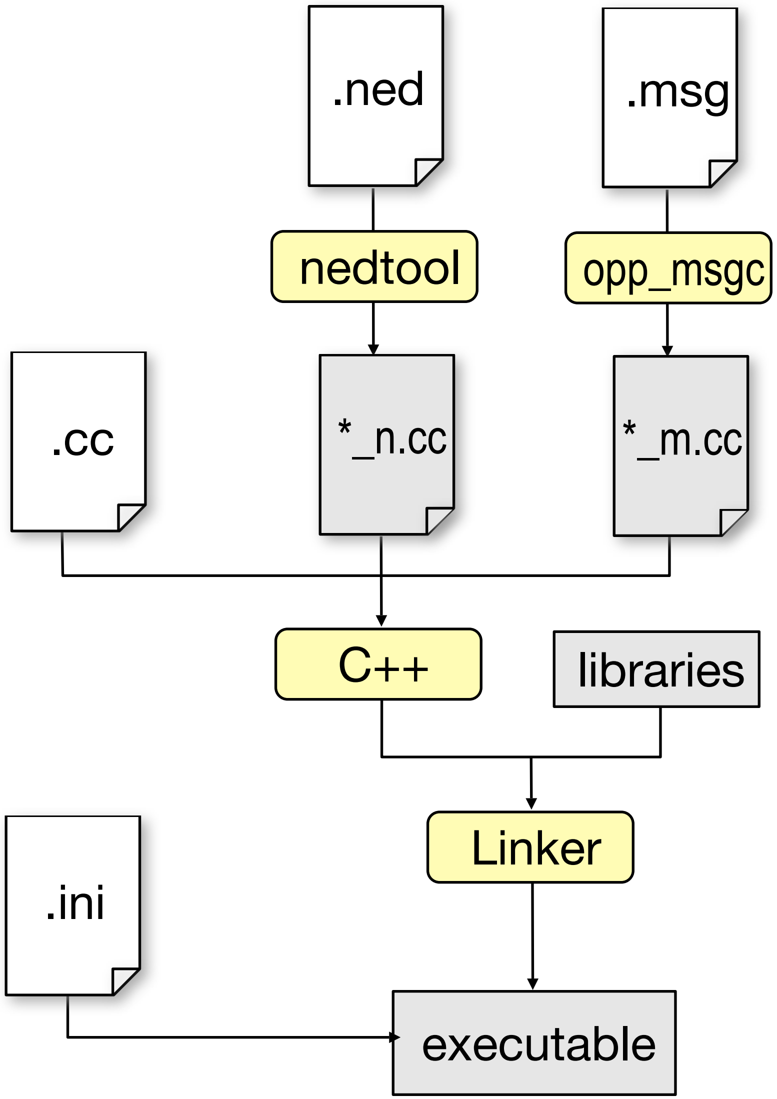
The general procedure:

- **Implementation**
  - Define module behaviour (event generation, event processing)
  - Define message format

- **Simulation setup:**
  - Define parameters
  - Define metrics to be observed during simulation

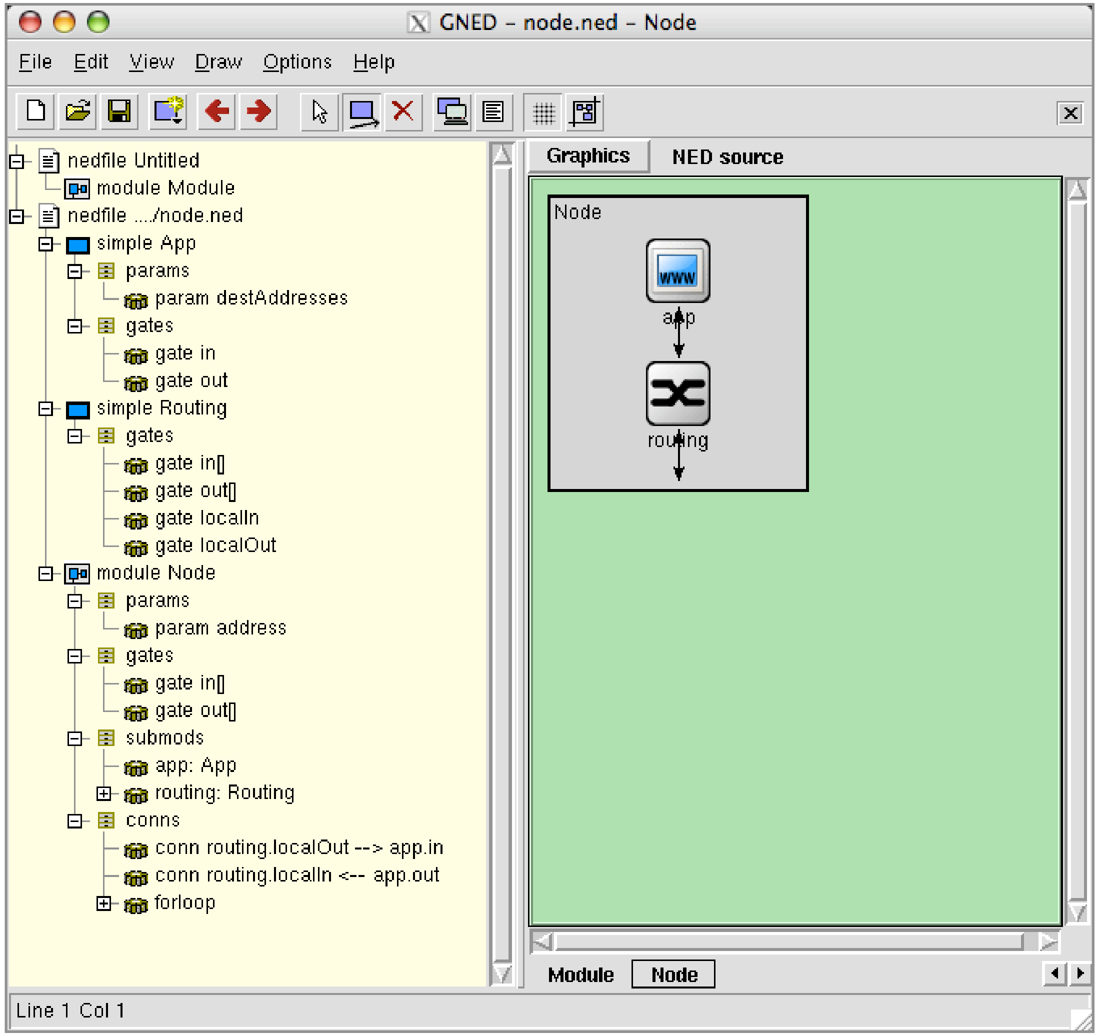
How to Write a Simulation (2)

1. Define modules and network topology (.ned)
2. Define messages (.msg)
3. Implement the behaviour of simple modules (.cc)
4. Compile the project
5. Define the parameters for the simulation (omnetpp.ini)



Step 1. Defining Modules

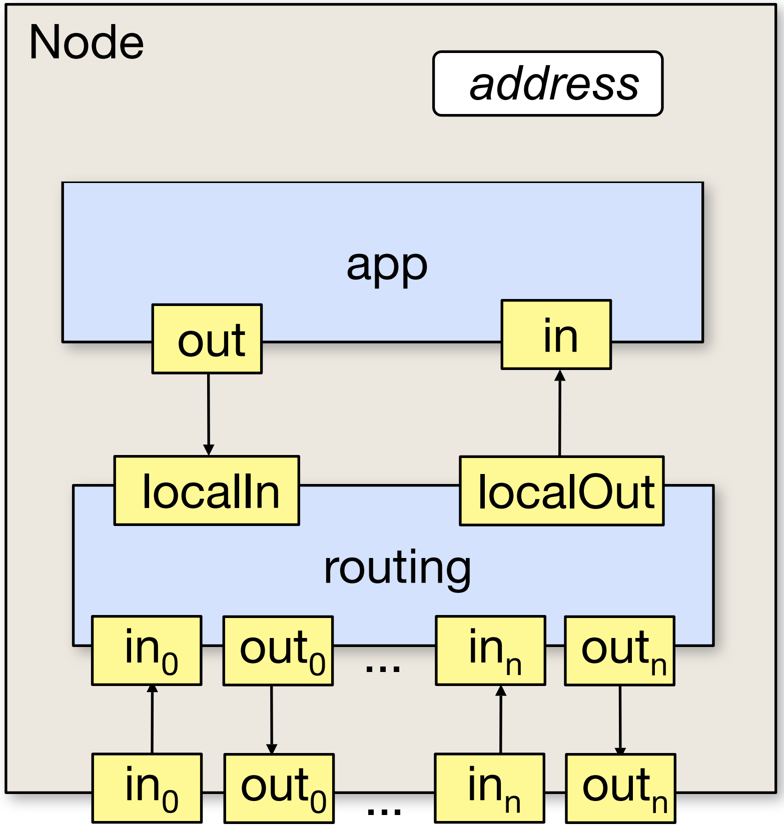
- Modules are defined using the NED language (OMNeT specific)
- GNED - a graphical editor for NED files
- Understanding the NED language is not that difficult...



Step 1. Defining Modules (2)

Example:

```
module Node
    parameters:
        address : numeric;
    gates:
        in: in[];
        out: out[];
    submodules:
        app: App;
        routing: Routing;
            gatesizes:
                in[sizeof(in)],
                out[sizeof(out)];
    connections:
        routing.localOut --> app.in;
        routing.localIn <-- app.out;
        for i=0..sizeof(in)-1 do
            routing.out[i] --> out[i];
            routing.in[i] <-- in[i];
        endfor;
endmodule
```



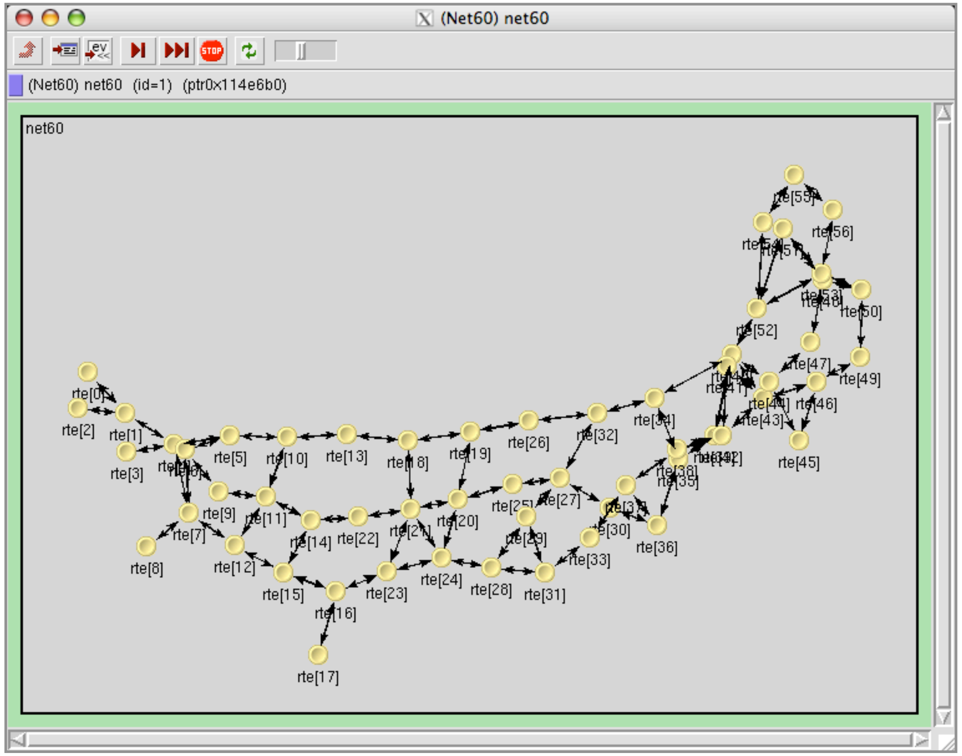
(see ../samples/routing)

Step 1. Defining a Network

Compound module containing the nodes:

```
import "node";

module Net60
    submodules:
        rte: Node[57];
            parameters:
                address = index;
    connections nocheck:
        rte[0].out++ --> rte[1].in++;
        rte[0].in++ <-- rte[1].out++;
        ...
        ...
        rte[0].out++ --> rte[1].in++;
        rte[0].in++ <-- rte[1].out++;
endmodule
```



Network definition:

```
network net60 : Net60
endnetwork
```

Step 2. Defining a Network (2)

- **nedtool creates C++ classes (if not loaded dynamically)**

node.ned → nedtool → node_n.cc

node.ned

```
module Node
    parameters:
        address : numeric;
    gates:
        in: in[];
        out: out[];
    submodules:
        app: App;
        routing: Routing;
            gatesizes:
                in[sizeof(in)],
                out[sizeof(out)];
    connections:
        routing.localOut --> app.in;
        routing.localIn <-- app.out;
        for i=0..sizeof(in)-1 do
            routing.out[i] --> out[i];
            routing.in[i] <-- in[i];
        endfor;
endmodule
```

node_n.cc

```
[...]

ModuleInterface(Node)
    // parameters:
    Parameter(address, ParType_Numeric)
    // gates:
    Gate(in[], GateDir_Input)
    Gate(out[], GateDir_Output)
EndInterface

Register_ModuleInterface(Node);

class Node : public cCompoundModule
{
  public:
    Node() : cCompoundModule() {}
  protected:
    virtual void doBuildInside();
};

Define_Module(Node);

[...]
```

Step 2. Defining Messages

- Messages are C++ classes and either of class cMessage or derived from this class
- Messages are handled in a module by the method `handleMessage(cMessage *msg)`
- Messages are sent to other modules by the method `send(cMessage *msg, const char *outGateName)`
- Timers are also realized by messages (self-messages)
- Messages can be defined in a MSG file. Example:

```
message Packet
{
  fields:
    int srcAddr;
    int destAddr;
    int hopCount;
}
```

Step 2. Defining Messages (2)

- **Define the fields in a .mgs file and let opp_msgc do the rest:**

packet.msg

```
message Packet
{
  fields:
    int srcAddr;
    int destAddr;
    int hopCount;
}
```

msgc

packet_m.h

```
class Packet : public cMessage
{
  protected:
    int srcAddr_var;
    int destAddr_var;
    int hopCount_var;

  public:
    Packet(const char *name=NULL, int kind=0);
    Packet(const Packet& other);
    virtual ~Packet();
    Packet& operator=(const Packet& other);
    virtual cPolymorphic *dup() const {
              return new Packet(*this);}
    virtual void netPack(cCommBuffer *b);
    virtual void netUnpack(cCommBuffer *b);

    virtual int getSrcAddr() const;
    virtual void setSrcAddr(int srcAddr_var);
    virtual int getDestAddr() const;
    virtual void setDestAddr(int destAddr_var);
    virtual int getHopCount() const;
    virtual void setHopCount(int hopCount_var);
};
```

getter and setter methods are automatically generated

Step 3. Module Implementation

- **Derive a class from cSimpleModule**

```
#include <omnetpp.h>
#include "packet_m.h"     ← include msg definitions

class Routing : public cSimpleModule {
  [...]
}

Define_Module(Routing);     ← register the module class
```

- **Redefine the methods (virtual methods of cModule)**
  - `initialize()` e.g., to define state variables
  - `handleMessage(cMessage *msg)`
  - `finish()` e.g., for statistics collection

Step 3. Event Handling

- Events are generated by sending messages from one module to other modules oder to itself
- Event handling is performed by `handleMessage(cMessage *msg)`
- Message processing depends on the state of a module, but also changes the state
- State variables are members of the module class
- Message sending (event generation) methods:
  - `send(cMessage* msg, int gateid)`
  - `scheduleAt(simtime_t t, cMessage* msg)`
  - `cancelEvent(cMessage* msg)`

Example of message handling

```
void Routing::handleMessage(cMessage *msg)
{
    Packet *pk = check_and_cast<Packet *>(msg);
    int destAddr = pk->getDestAddr();

    if (destAddr == myAddress)
    {
        ev << "local delivery of packet " << pk->name() << endl;
        send(pk,"localOut");
        return;
    }

    RoutingTable::iterator it = rtable.find(destAddr);
    if (it==rtable.end())
    {
        ev << "address " << destAddr << " unreachable, discarding packet "
           << pk->name() << endl;
        delete pk;
        return;
    }

    int outGate = (*it).second;
    ev << "forwarding packet " << pk->name() << " on gate id=" << outGate << endl;
    pk->setHopCount(pk->getHopCount()+1);

    send(pk, outGate);
}
```

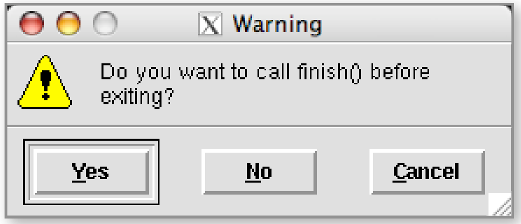
Step 3. Initialization and Finishing

- `initialize()` is the right place to initialize variables or create initial events, e.g.:

```
void AModule::initialize() {
 scheduleAt(simTime + 0.5, new cMessage);
}
```

- In the constructor not all information may be available at runtime (e.g. the total number of nodes)

- `finish()` is the counterpart to `initalize()`
- it is called at the end of the simulation



Step 4. Compiling the project

- **A Makefile can be created by opp_makemake**
  - from the source files in the current directory
  - with the necessary settings (compiler flags, libraries)

- **In the simplest case (one directory), call**
  ```
  opp_makemake -N
  make
  ```

  `-N` load NED files dynamically
  `-I` additional include directories (when using frameworks)
  `-u` specify user interface
  - `-u Tkenv` for the GUI (default)
  - `-u Cmdenv` for the command line

Step 5. Setting Simulation Parameters

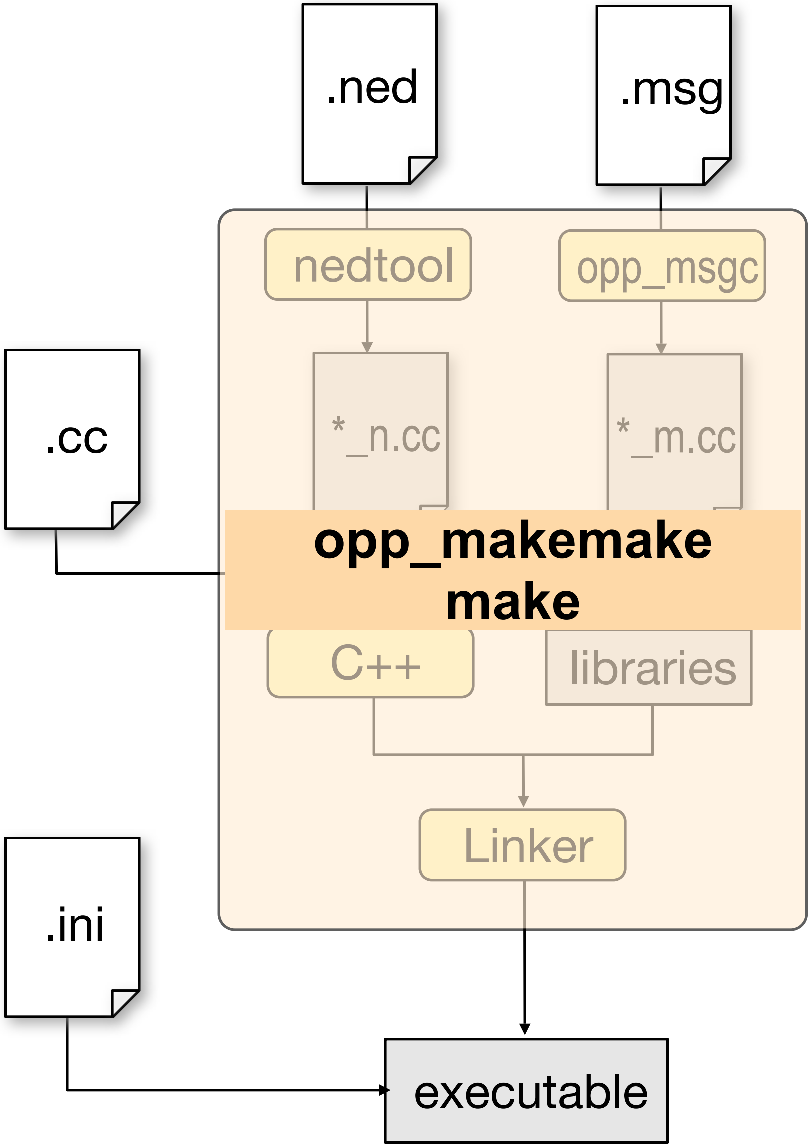
- **Create a file "omnetpp.ini"**
- **Contents:**
  - selection of the network
  - pre-loaded NED files
  - selection of the random number generator
  - parameters
- **Example:**

```
[General]
preload-ned-files=*.ned        ← load all NED files dynamically
network=net60

[Parameters]
net60.**.destAddresses="1 50"
      ↑
   wildcards
```

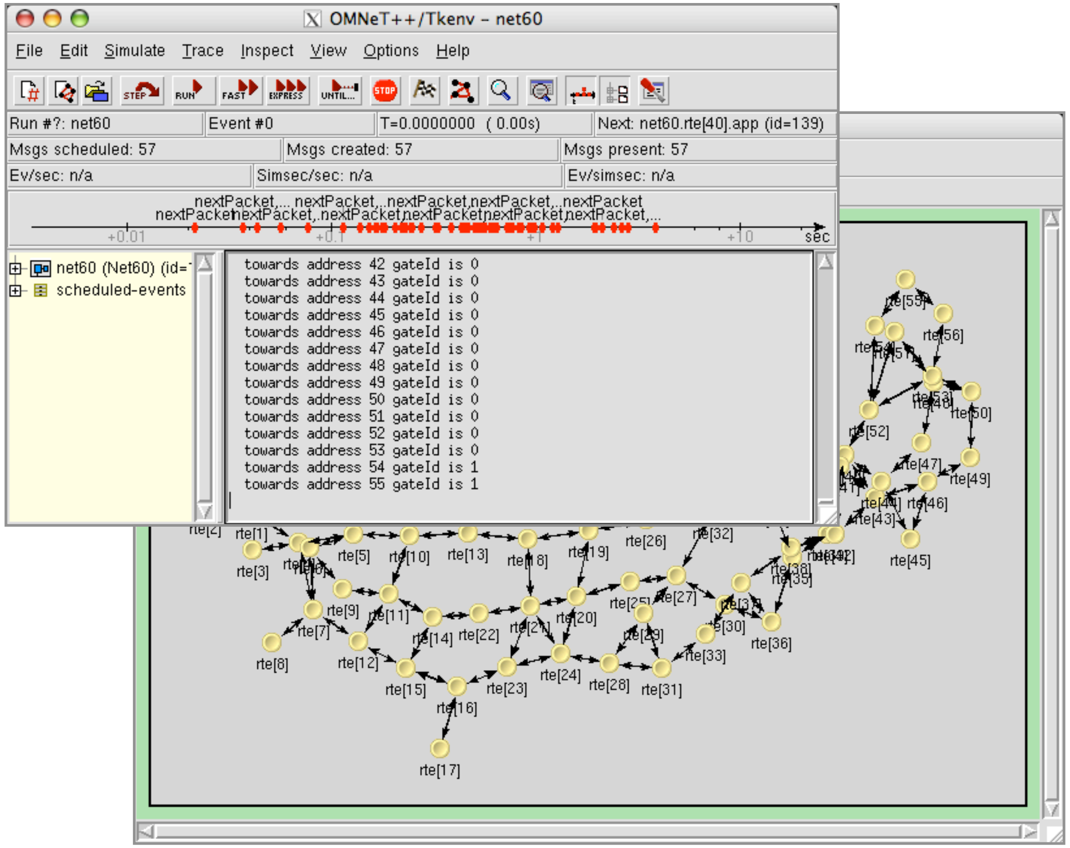
How to write a simulation

1. Define modules and network topology (.ned)
2. Define messages (.msg)
3. Implement the behaviour of simple modules (.cc)
4. Compile the project (Makefile)
5. Define the parameters for the simulation (omnetpp.ini)



Running Simulations

- Calling the executable starts the GUI (Tkenv) or the command line (Cmdenv) version



- Command-line switches for the executable:

`-f <inifile>`	specifies an ini file (default: omnetpp.ini)
`-r <runs>`	specifies the runs to be executed (e.g. 1,2,4-6)

Running Several Simulations

- Several runs started by a shell script

```
#! /bin/sh
for ((i=1; $i<50; i++)); do
  ./wireless -f runs.ini -r $i
done
```

← $i = run number

- Define parameters in the `[Run 1]`, `[Run 2]`,... sections of the ini file or define variable parameters in different ini files, e.g. `10nodes.ini`:

```
include universal.ini

[Parameters]
Wireless.n = 10
```

← inclusion of common settings

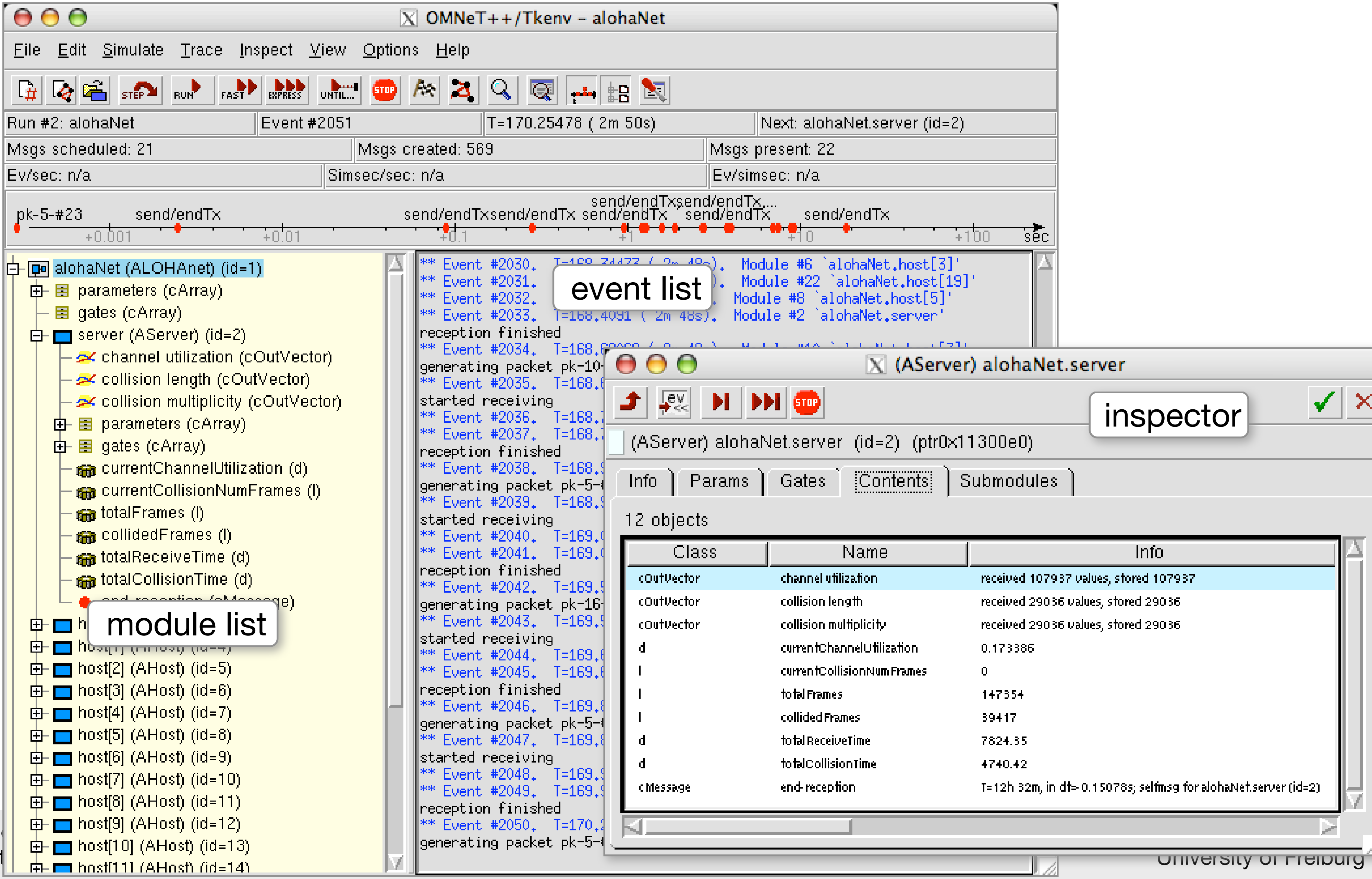
- Start the simulation for each ini file

```
#! /bin/tcsh
foreach f (*.ini)
  nice  +15 ./simulation -f $f >! $f:r.log
end
```

Random Number Generators

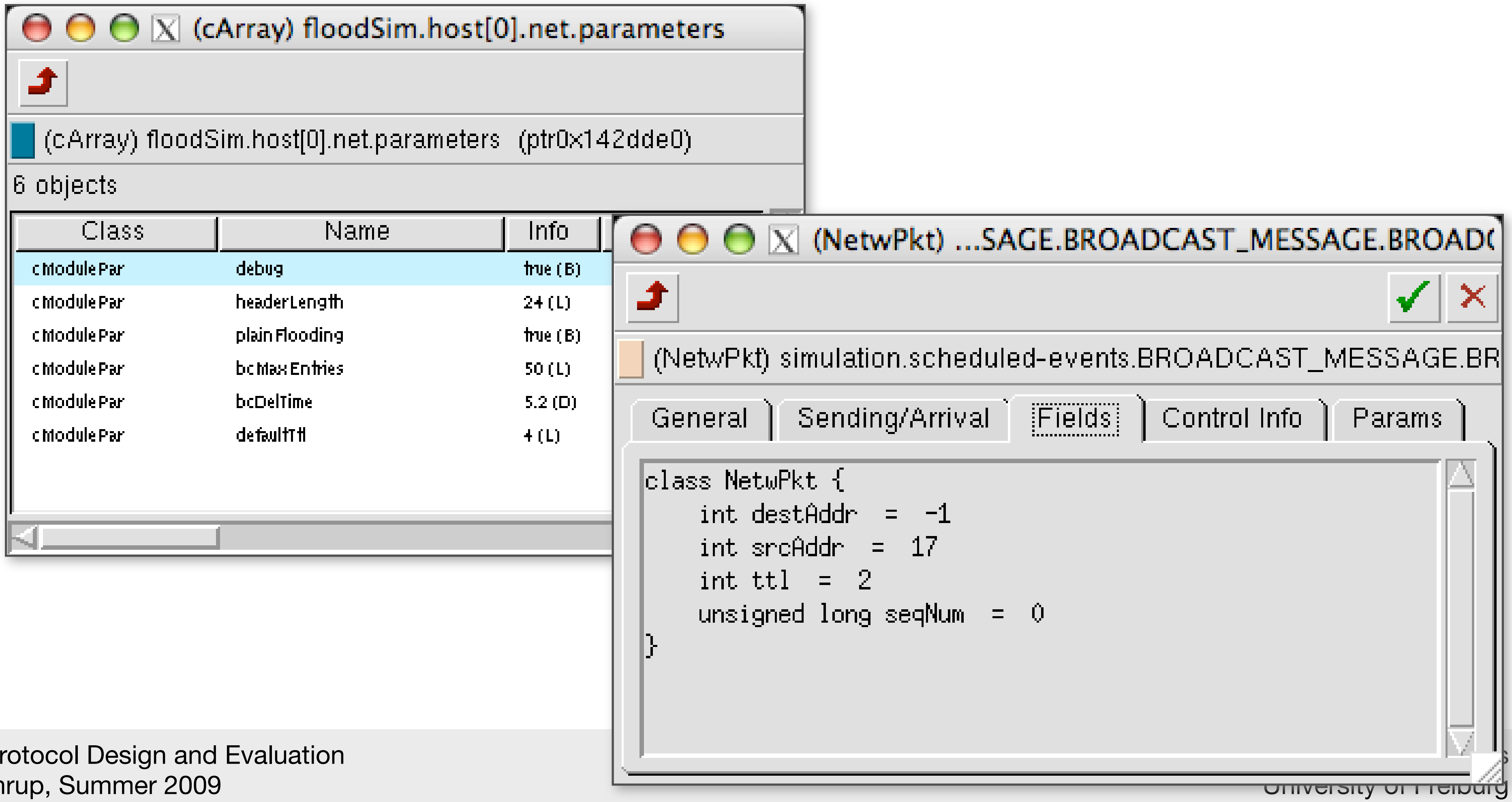
- Standard RNG: Mersenne Twister with a period of $2^{19937} - 1$
- Several predefined distributions (uniform, exponential, normal, Pareto, ...)
- The number of RNGs is set in the ini file
  (multiple RNGs to avoid unwanted correlation)
- Seeds are automatically selected
  (based on RNG number and run number)
- RNGs can be mapped to modules
  e.g. the default RNG for the channel module is RNG No. 1

GUI Features



Inspectors

- Members of module classes derived from cObject (e.g., cArray, cMessage) are displayed in the inspector:



Statistics collection

- Record scalar statistics in the finish() method (→ .sca file)

```
AServer::finish() {
  recordScalar("channel utilization",currentChannelUtilization);
}
```

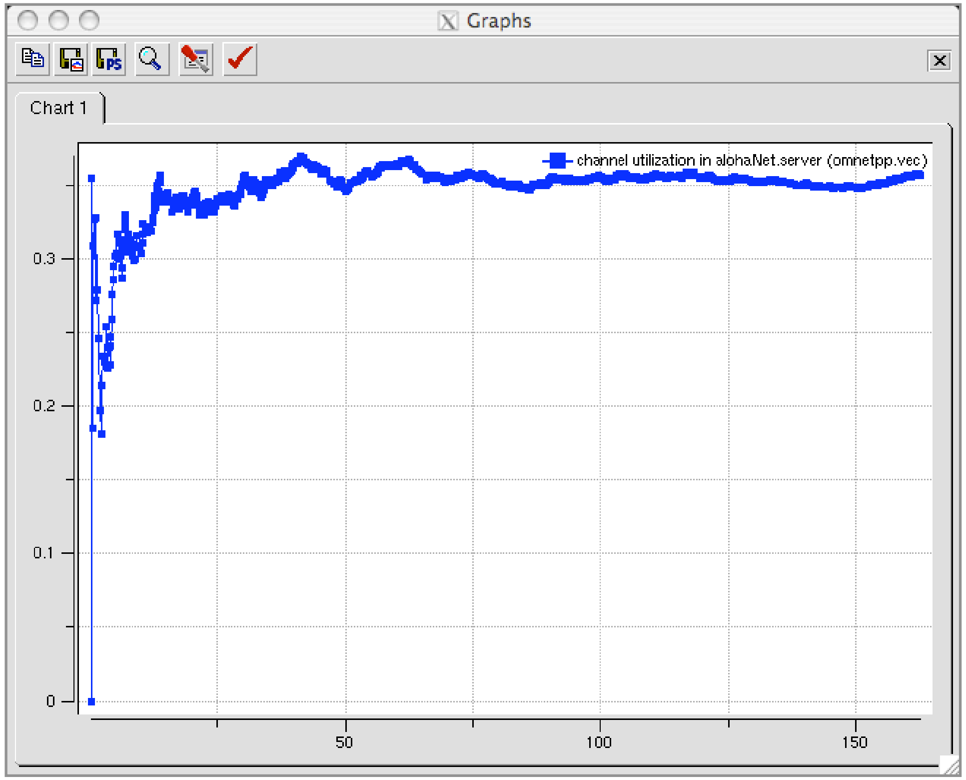
- Record output vectors (→ .vec file)

```
AServer::initialize() {
  cOutVector channelUtilizationVector;
}

AServer::handleMessage(cMessage* msg) {
  channelUtilizationVector.record(currentChannelUtilization);
}
```

Statistics evaluation

- Scalar values of different runs are appended to the .sca file
- Scalar files (.sca) can be processed by the scalars tool
- Vector files (.vec) can be processed by the plove tool



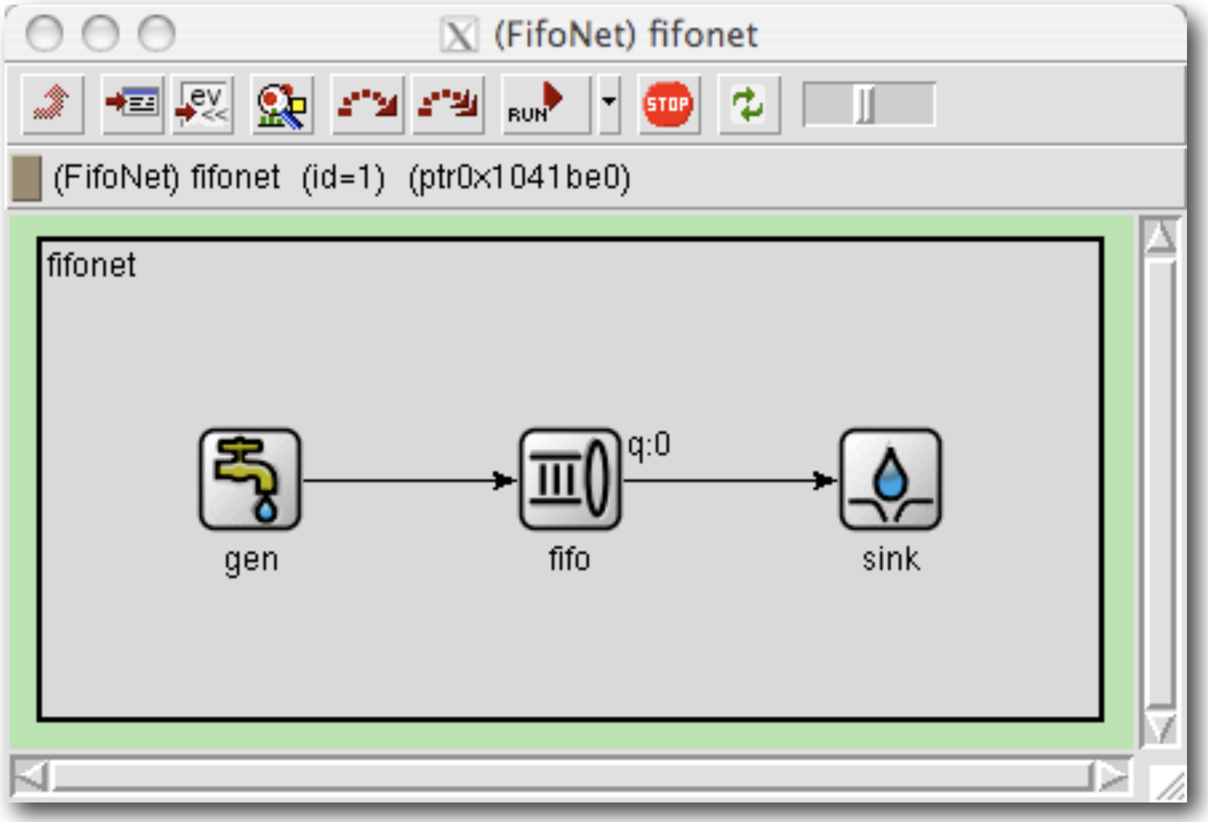
Resources

- **www.omnetpp.org**

- **OMNeT++ User Community Wiki: www.omnetpp.org/pmwiki**
  - Installation instructions
  - description of frameworks

- **Documentation ("doc" directory of the distribution):**
  - User Manual (html or PDF, 230 pages)
  - API documentation (html, doxygen/neddoc)

Example: Simulating a Queue



see <omnet>/samples/fifo

The Network

```
module FifoNet
    submodules:
        gen: FFGenerator;
            display: "p=89,100;i=block/source";
        fifo: FFBitFifo;
            display: "p=209,100;i=block/queue;q=queue";
        sink: FFSink;
            display: "p=329,100;i=block/sink";
    connections:
        gen.out --> fifo.in;
        fifo.out --> sink.in;
endmodule
```

The load generator

```
void FFGenerator::initialize()
{
    sendMessageEvent = new cMessage("sendMessageEvent");
    scheduleAt(0.0, sendMessageEvent);
}

void FFGenerator::handleMessage(cMessage *msg)
{
    ASSERT(msg==sendMessageEvent);

    cMessage *m = new cMessage("job");
    m->setLength(par("msgLength"));
    send(m, "out");

    scheduleAt(simTime()+(double)par("sendIaTime"),
               sendMessageEvent);
}
```

parameter from the omnetpp.ini file

```
[Run 1]
description="low job arrival rate"
**.gen.sendIaTime = exponential(0.1)
```

The Queue

```
void FFAbstractFifo::handleMessage(cMessage *msg)
{
    if (msg==endServiceMsg) {    timer event (job is finished)
        endService( msgServiced );
        if (queue.empty()) {
            msgServiced = NULL;    delete pointer to current job
        } else {
            msgServiced = (cMessage *) queue.pop();
            simtime_t serviceTime = startService( msgServiced );
            scheduleAt( simTime()+serviceTime, endServiceMsg );
        }
    }        else... incoming message (job), not a timer
    else if (!msgServiced) {
        arrival( msg );
        msgServiced = msg;    store current job
        simtime_t serviceTime = startService( msgServiced );
        scheduleAt( simTime()+serviceTime, endServiceMsg );

    }        set timer for finishing current job
    else {                                              timer event
        arrival( msg );
        queue.insert( msg );
    }
}
```

The Sink

```
void FFSink::handleMessage(cMessage *msg)
{
    double d = simTime()-msg->creationTime();
    ev << "Received " << msg->name() << ", queueing time: "
       << d << "sec" << endl;
    qstats.collect( d );
    qtime.record( d );
    delete msg;
}
```

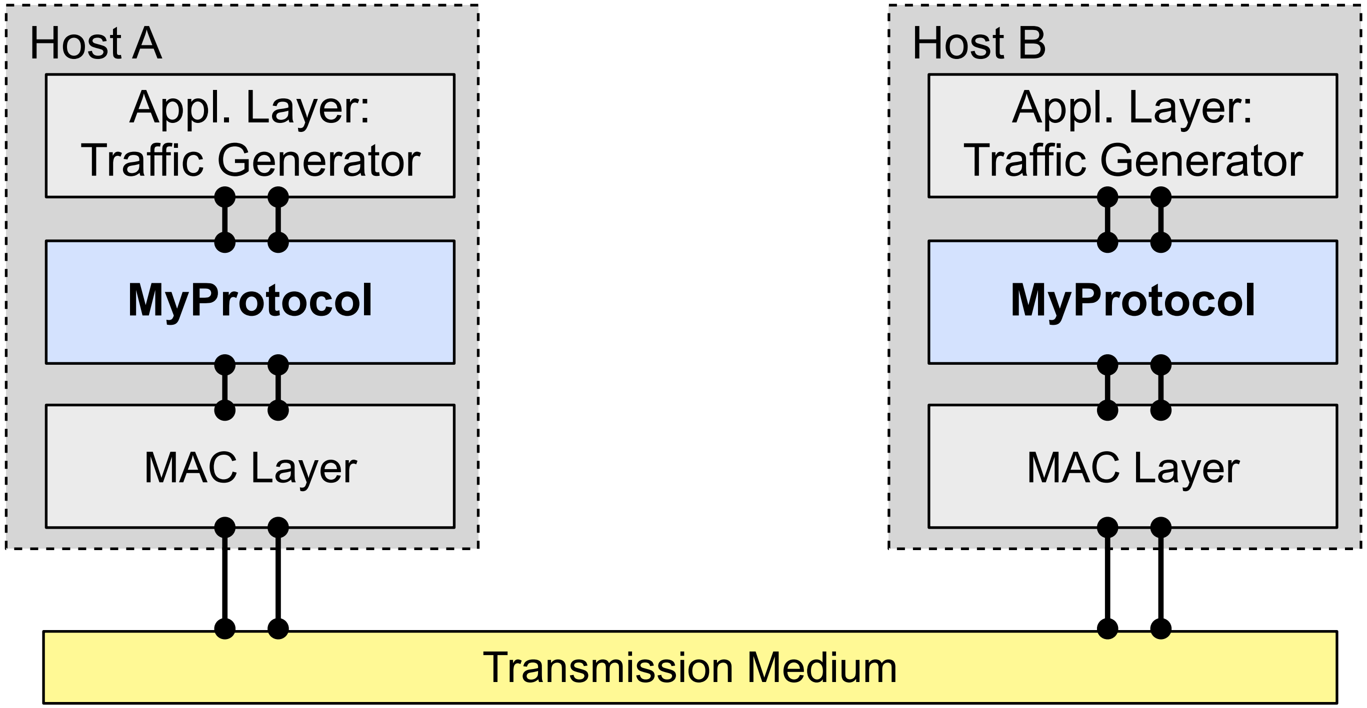
Simulation of Wireless Networks

- **Characteristics of wireless networks**
  - Wireless links: packet errors, packet loss, delay
  - Mobility: links are not permanent

- **Required:**
  - Distinct channel model
  - Mobility model
  - In mobile scenarios: dynamic link management

Wireless Channel Simulation

- **Channel model includes various effects of wireless transmissions**



Wireless Channel Simulation

- **Wireless transmission**
  - Radio-wave propagation: calculating the received signal strength
    - based on large-scale path loss, small-scale and multipath fading
  - Interference: calculating packet loss
    - Signal-to-noise/interference ratio

[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Radio-wave Propagation

- **Impact on radio-wave propagation:**
  - *Attenuation* by distance
  - *Reflection* on obstacles
  - *Diffraction* by obstacles with sharp edges
  - *Scattering* by objects which are small compared to the wavelength

[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Towards Propagation Models

- **Effects that can be observed**
  - large-scale path loss
    - attenuation with increased distance
  - small-scale fading
    - rapid changes in signal strength (while time and distance variation is small)
    - random frequency modulation (Doppler shifts on multipath signals)
    - Echoes by multipath propagation delays

- Propagation models try to capture (some of) these effects ...

[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Propagation models

- **Elementary models:**
  - Free-space propagation model
  - Two-ray ground reflection model
  - Shadowing model
- **Empirical models**
  - Outdoor models (main effect: ground reflection)
  - Indoor models (obstacle-dependent path loss)
- **Raytracing**

Free Space Propagation

- Line-of-sight without obstacles
- Received signal strength in distance d:

$$P_r(d) = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2 d^2 L}$$

  - Transmission power P_t
  - Transmitter gain G_t, receiver gain G_r
  - Wavelength L
  - Speed of light

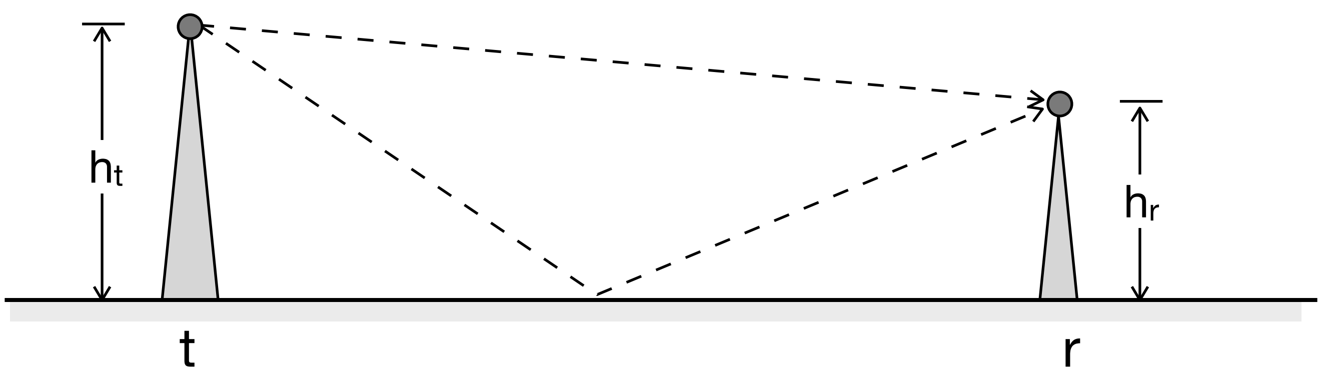
[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Two-way Ground Reflection

- Attenuation of the direct path signal by a reflected signal:

$$P_r(d) = \frac{P_t G_t G_r {h_t}^2 {h_r}^2 \lambda^2}{d^4 L}$$

h_t, h_r = height of sender and receiver



[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Log-distance Path Loss

- Generalization of the previous models: Path loss is proportional to distance with some exponent

$$\mathsf{PL}_{\mathsf{dB}}(\mathsf{d}) \propto \left(\frac{\mathsf{d}}{\mathsf{d}_0}\right)^{\beta}$$

- In dB (logarithmic scale):

$$\mathsf{PL}(\mathsf{d})_{[\mathsf{dB}]} = \mathsf{PL}(\mathsf{d}_0) + 10\,\beta\,\log\left(\frac{\mathsf{d}}{\mathsf{d}_0}\right)$$

- Reference path loss at d_0 through measurements or free space model.

[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Path Loss Exponents

- Empirical results for different environments

Environment	Path loss exponent
Free space	2
Urban area cellular radio	2.7 - 3.5
Urban area cellular with shadowing	3 - 5
Indoor, line-of-sight	1.6 - 1.8
Indoor obstructed	4 - 6
Indoor, factories, obstructed	2 - 3

- There is a significant difference between line-of-sight and non-LoS connections!

[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Log-normal Shadowing (1)

- The log-normal shadowing model includes path loss and Gaussian fading

$$\begin{aligned} PL'(d)_{[dB]} &= PL(d) + X_\sigma \\ &= \underbrace{PL(d_0) + 10\,\beta \log\left(\frac{d}{d_0}\right)}_{\text{mean path loss}} + \underset{\substack{\uparrow \\ \text{random variation}}}{X_\sigma} \end{aligned}$$

- PL'(d) is a random variable with normal distribution
- Receiver signal strength: P_r'(d) = P_t - PL'(d)

[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Log-normal Shadowing (2)

- The Q-function (tail probability of a normal distribution) can be used to determine the probability of a succesful reception, i.e. signal strength above receiver threshold γ.
- Definition of the Q-function:

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} \exp\left(-\frac{u^2}{2}\right) du$$

- Probability of successful reception:

$$\Pr[P_r'(d) > \gamma] = Q\left(\frac{\gamma - P_r(d)}{\sigma}\right)$$

[T.S. Rappaport: Wireless Communications Principles and Practice, 2/e]

Mobility Models

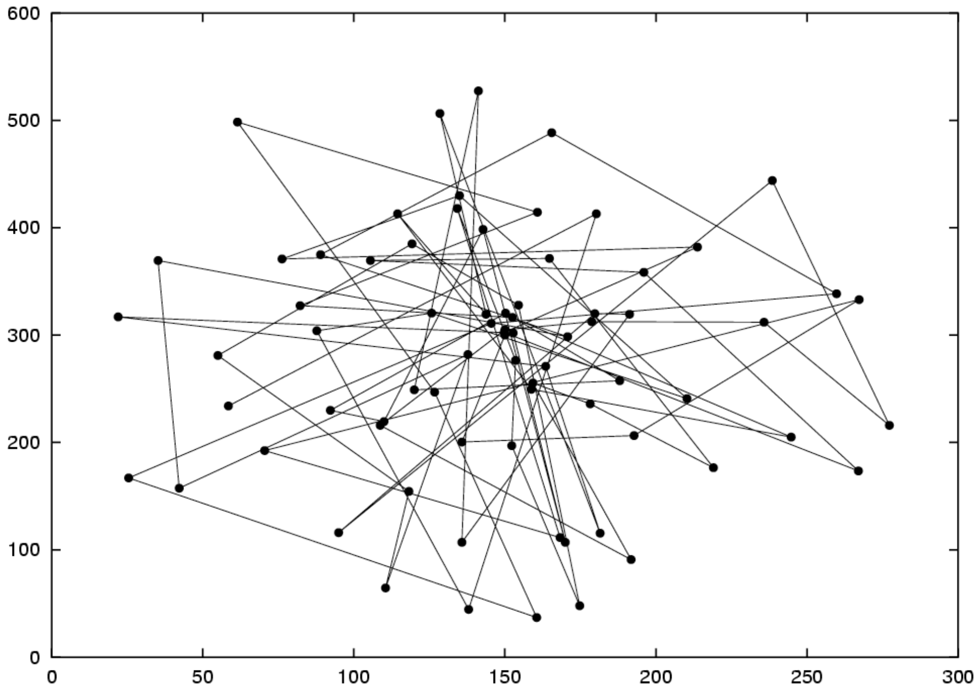
- Determine movement of network nodes

- Deterministic models: mobility traces
- Random models: random choice of direction, speed, ...

- Level of detail
  - Microscopic
  - Macroscopic
  - Aggregated behaviour (fluid flow)

Brownian Motion, Random Walk

- **Brownian Motion (microscopic view)**
  - Velocity and movement direction are chosen randomly and uniformly from $[v_{min},v_{max}]$ and $[0,\pi]$



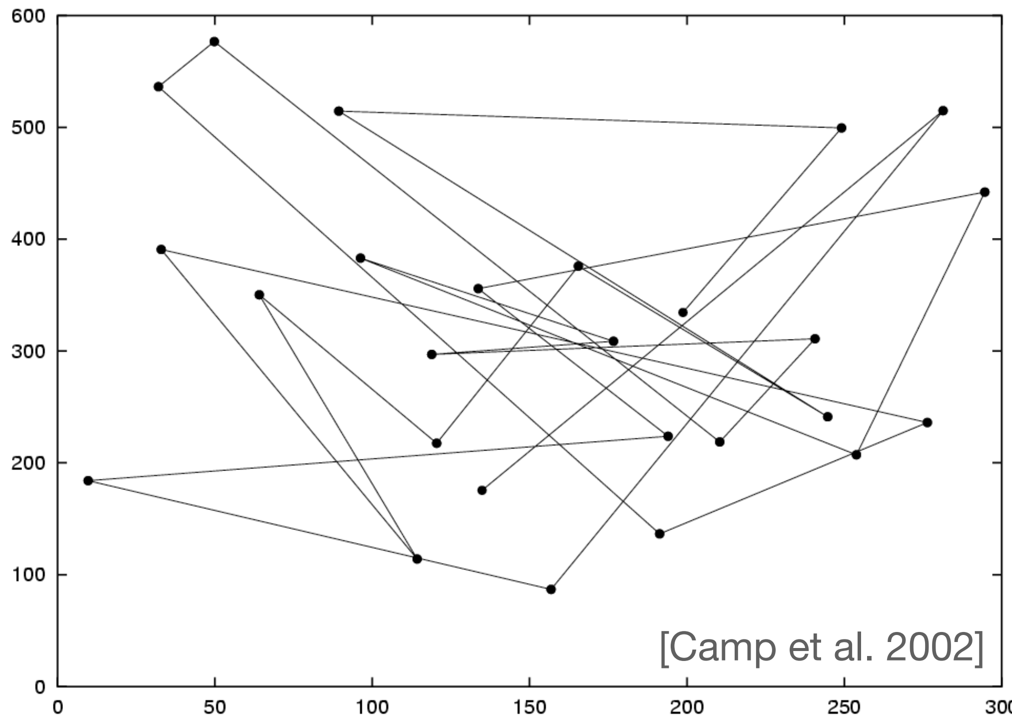
[Camp et al. 2002]

- **Random Walk**
  - macroscopib view
  - e.g. for cellular networks
  - random choice of next cell (among neighboring ones)
  - sojourn probability

Random Waypoint Mobility Model

[Johnson, Maltz 1996]

- Movement towards a randomly chosen target point
- Velocity randomly and uniformly from $[v_{min}, v_{max}]$
- Wait time if target is reached



Problems of the RWP Model

- Parameters of the Random Waypoint Model: min/max speed and pause time.
- What we expect: Average speed is $(v_{min} + v_{max})/2$
- This is **wrong**!
- Reasons:
  - The next waypoint and thus the travel distance is chosen independently of the speed. A lower speed causes a lower average speed, because the node travels a longer time with low speed
  - The longer the simulation runs, the more time is spent in slower trips

[Yoon, Liu, Noble: “Random Waypoint Considered Harmful”, INFOCOM 2003]

Gauss-Markov Mobility Model

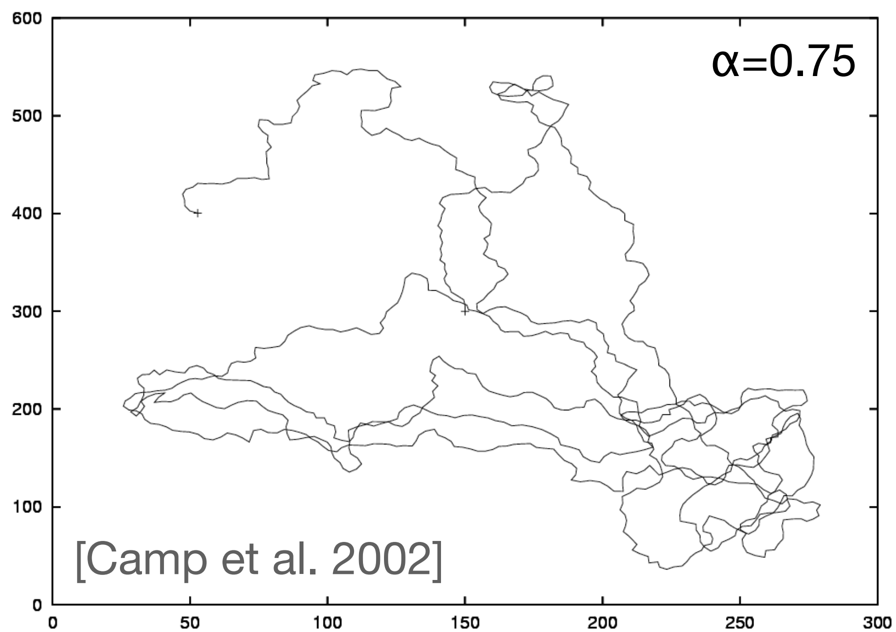
[Liang, Haas 1999]

- Tuning parameter for randomness
- Velocity: $v_n = \alpha v_{n-1} + (1-\alpha)\bar{v} + \sqrt{1-\alpha^2}\, v_{x_{n-1}}$
- Direction: $d_n = \alpha d_{n-1} + (1-\alpha)\bar{d} + \sqrt{1-\alpha^2}\, d_{x_{n-1}}$

tuning factor

mean

random variable
gaussian distribution



Simulation of Wireless Networks with OMNeT++

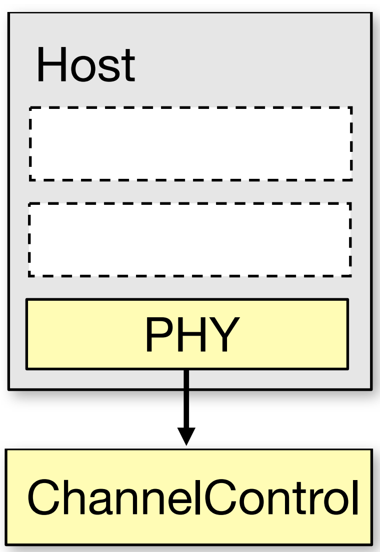
- Modules for wireless network simulations are provided by frameworks:
  - **Mobility Framework** (mobility-fw.sourceforge.net, wiki.github.com/mobility-fw/mf-opp4)
    - Support for node mobility and a wireless medium (dynamic connection management)
    - Modules for 802.11, CSMA
  - **INET Framework** (inet.omnetpp.org)
    - IP, TCP/UDP, OSPF, RIP
    - Ethernet, 802.11, PPP, ...
    - Support for wireless protocols and mobility (based on the Mobility Framework)

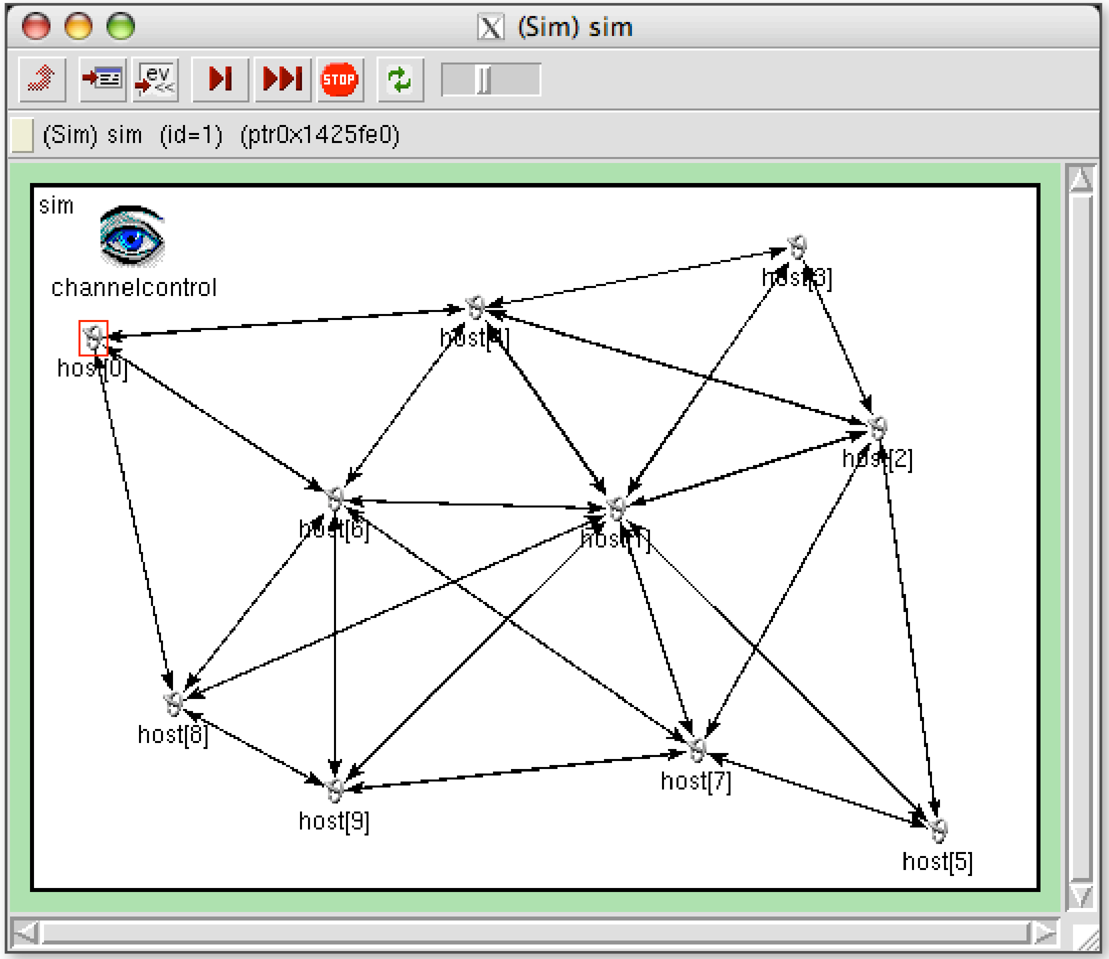
Mobility Framework

- **Simulation of the wireless medium:**
  - Module ChannelControl
  - Dynamic link assignment: Gates and connections are created dynamically, if a node is added or moves
  - Path loss and SIR calculation (based on distance)
- **Mobility**
  - Nodes have (geographical) positions
  - Various mobility models (subclasses of the BasicMobility module)
  - The position is changed every update interval (triggered by self-messages)

ChannelControl

- Module for the simulation of the wireless medium
- Links between the nodes are created dynamically
- PHY is connected to ChannelControl





On the Pitfalls of Simulation (1)

- **Simulating Internet Protocols:**
  - Complexity of the Internet topology: How to create realistic models?
  - Diversity of bandwidth, routers, protocols, ...
  - Other sources of traffic (traffic diversity: UDP, TCP, ...)
  - Load patterns: How to model the application layer?

On the Pitfalls of Simulation (2)

- **Simulating Wireless Networks:**
  - Too many effects on radio propagation to be considered in a simulation model.
  - Environment models: Significant differences between direct line-of-sight and non line-of-sight transmission
  - Mobility models: What is a realistic mobility pattern? Some models have unwanted side effects on the simulation results.

On the Pitfalls of Simulation (3)

- **In general:**
  - Inappropriate model abstraction
  - Bad pseudo random number generators, bad seed selection
  - Simulation time not sufficient
  - Inappropriate aggregation of statistical data

Simulation Practice

- Current simulators offer a lot of environmental models and protocols which increase the complexity
- There is a trend towards leaner models:

Problem	Current practice	New advice
Model complexity	Complex realistic modeling	Preserve simplicity where possible. Focus on effects that have an impact on the protocol behaviour
Simulation parameters	Scenarios with complex parameters	Start with a simple model, add more parameters later
Simulation procedure	Build complex simulation model and perform simulations	Parallel advance of modeling, simulation and protocol design

[I. Stojmenovic: Simulations in Wireless Sensor and Ad Hoc Networks, IEEE Communications Magazine, Dec. 2008]