Simulation

Modeling and Performance Analysis with Discrete-Event Simulation

Dr. Mesut Güneş
Chapter 4

Introduction to Network Simulators
Contents

- Network Simulator Tools
- ns-2
- OMNeT++
Network Simulator Tools
Network Simulator Tools

  - ns-2 is a discrete event simulator targeted at networking research.
  - ns-2 provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks.

- **OMNeT++** - http://www.omnetpp.org
  - OMNeT++ is an open-source, component-based simulation package built on C++ foundations. It offers a C++ simulation class library and GUI support (graphical network editing, animation).
Network Simulator Tools

- **SSFNet - http://www.ssfnet.org/**
  - SSF (Scalable Simulation Framework) is a standard for discrete-event simulation in Java and C++. Several SSF implementations and a large number of open-source protocol models and other components exist.

- **Parsec - http://may.cs.ucla.edu/projects/parsec/**
  - A C-based simulation language for sequential and parallel execution of discrete-event simulation models.

- **Scalable Networks (Qualnet) - http://www.scalable-networks.com**
  - Network simulator designed from the outset for maximum speed and scalability, with real-time simulation as an achievable goal.

- **OPNET Modeler - http://www.opnet.com/**
  - OPNET Modeler is a commercial tool for modeling and simulation of communications networks, devices, and protocols. It features graphical editors and animation.
Network Simulator Tools

- JiST / SWANS - http://jist.ece.cornell.edu/
  - “JiST is a high-performance discrete event simulation engine that runs over a standard Java virtual machine. It is a prototype of a new general-purpose approach to building discrete event simulators, called virtual machine-based simulation, that unifies the traditional systems and language-based simulator designs. The resulting simulation platform is surprisingly efficient. It out-performs existing highly optimized simulation runtimes both in time and memory consumption. For example, JiST has twice the raw event throughput of the highly optimized, C-based Parsec engine, and supports process-oriented simulation using a fraction of the memory.”
Network Simulator Tools

  - Tool for generation of realistic internet topologies, with export to several network simulators (ns2, SSFNet, OMNeT++).

  - Akaroa is a package for supporting the Multiple Replications In Parallel (MRIP) simulation technique to harness the computing power of a network of inexpensive workstations.
  - Integration exists with the ns2 and OMNeT++ simulators.
The Network Simulator, ns-2
ns-2

- **Simple model**
  - a discrete event simulator
- **Focused on modeling network protocols**
  - wired, wireless, satellite
  - TCP, UDP, multicast, unicast
  - web, telnet, ftp
  - ad hoc routing, sensor networks
  - infrastructure: stats, tracing, error models, etc.
- **Literature**
  - Project homepage: http://www.isi.edu/nsnam/
  - Ns manual: http://www.isi.edu/nsnam/ns/ns-documentation.html
ns-2 – Goal

- **Support networking research and education**
  - protocol design, traffic studies, etc.
  - protocol comparison

- **Provide a collaborative environment**
  - freely distributed, open source

- **Share code, protocols, models, etc.**
  - allow easy comparison of similar protocols
  - increase confidence in results

- **More people look at models in more situations**

- **Experts develop models**

- **Multiple levels of detail in one simulator**
ns2 – History

- Development began as REAL in 1989
- ns by Floyd and McCanne at LBL
- ns-2 by McCanne and the VINT project (LBL, PARC, UCB, USC/ISI)
- Currently maintained at USC/ISI
- In future ns-3 (http://www.nsnam.org/)
  - “The ns-3 project is developing a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. ns-3 is the next major revision of the ns-2 simulator. The acronym “nsnam” derives historically from the concatenation of ns (network simulator) and nam (network animator).”
ns-2 – Components

- **ns**: the simulator itself
- **nam**: the Network Animator
  - Visualize ns (or other) output
  - GUI input simple ns scenarios
- **Pre-processing**:  
  - Traffic and topology generators
- **Post-processing**:  
  - Simple trace analysis, often in Awk, Perl, Python, or Tcl
ns-2 – Models

- **Traffic models and applications**
  - Web, FTP, telnet, constant-bit rate, Real Audio

- **Transport protocols**
  - Unicast: TCP (Reno, Vegas, etc.), UDP
  - Multicast: SRM

- **Routing and queueing**
  - Wired routing, ad hoc routing and directed diffusion
  - Queueing protocols: RED, drop-tail, etc.

- **Physical media**
  - Wired (point-to-point, LANs),
  - Wireless (multiple propagation models), satellite
ns-2 – Installation and Documentation

- **Homepage:** [http://www.isi.edu/nsnam/ns/](http://www.isi.edu/nsnam/ns/)
  - Download ns-allinone
  - Includes Tcl, OTcl, TclCL, ns, nam, etc.

- **Mailing list:**
  - ns-users@isi.edu

- **Documentation**
  - Marc Gries tutorial
  - ns manual
ns-2 – Creating Event Scheduler

- **Create scheduler**
  - set ns [new Simulator]

- **Schedule event**
  - $ns at <time> <event>
  - <event>: any legitimate ns/tcl commands

- **Start scheduler**
  - $ns run
ns-2 – Creating a Network

- **Nodes**
  - set n0 [$ns node]
  - set n1 [$ns node]

- **Links & Queuing**
  - $ns duplex-link $n0 $n1 <bandwidth> <delay> <queue_type>
  - <queue_type>: DropTail, RED, CBQ, FQ, SFQ, DRR
ns-2 – Computing Routes

### Unicast
- `$ns rtproto <type>`
- `<type>`: Static, Session, DV, cost, multi-path

### Multicast
- `$ns multicast`
  - right after [new Simulator]
- `$ns mrtproto <type>`
- `<type>`: CtrMcast, DM, ST, BST
ns-2 – Traffic

- Simple two layers: transport and application
- Transport protocols:
  - TCP, UDP, etc.
- Applications: (agents)
  - ftp, telnet, etc.
ns-2 – Creating Connections

- **Source and sink**
  - set usrc [new Agent/UDP]
  - set udst [new Agent/NULL]

- **Connect them to nodes, then each other**
  - $ns$ attach-agent $n0$ $usrc$
  - $ns$ attach-agent $n1$ $udst$
  - $ns$ connect $usrc$ $udst$
ns-2 – Creating Connections

- **Source and sink**
  - set tsrc [new Agent/TCP]
  - set tdst [new Agent/TCPSink]

- **Connect to nodes and each other**
  - $ns$ attach-agent $n0$ $tsrc$
  - $ns$ attach-agent $n1$ $tdst$
  - $ns$ connect $tsrc$ $tdst$
ns-2 – Creating Traffic: On Top of TCP

- **FTP**
  - set ftp [new Application/FTP]
  - $ftp attach-agent $tsrc
  - $ns at <time> "$ftp start"

- **Telnet set**
  - telnet [new Application/Telnet]
  - $telnet attach-agent $tsrc
ns-2 – Creating Traffic: On Top of UDP

- **CBR**
  - set src [new Application/Traffic/CBR]

- **Exponential or Pareto on-off**
  - set src [new Application/Traffic/Exponential]
  - set src [new Application/Traffic/Pareto]
ns-2 – Creating Traffic: Trace Driven

- **Trace driven**
  - set tfile [new Tracefile]
  - $tfile filename <file>
  - set src [new Application/Traffic/Trace]
  - $src attach-tracefile $tfile

- **<file>**:
  - Binary format
  - inter-packet time (msec) and packet size (byte)
ns-2 – End-to-End Argument: File Transfer

- **Even if network guaranteed reliable delivery**
  - Need to provide end-to-end checks
  - e.g., network card may malfunction

- **If network is highly unreliable**
  - Adding some level of reliability helps performance, not correctness
  - Don’t try to achieve perfect reliability!
OMNeT++
Objective Modular Network Testbed in C++ (OMNeT++)
- General-purpose tool for discrete event simulations
- Object-oriented design

Literature
- OMNeT++ Community Site
  http://www.omnetpp.org
- User Manual
OMNeT++ – Goals

- The simulator can be used for:
  - traffic modeling of telecommunication networks
  - protocol modeling
  - modeling queueing networks
  - modeling multiprocessors and other distributed hardware systems
  - validating hardware architectures
  - evaluating performance aspects of complex software systems
  - ... modeling any other system where the discrete event approach is suitable.
OMNeT++ – Screenshot
An OMNeT++ model consists of hierarchically nested modules

**Simple Modules**
- Lowest level of the module hierarchy
- Simple modules contain the algorithms in the model
- The user implements the simple modules in C++
  - Using the OMNeT++ simulation class library

**Compound Modules**
- Module contains submodules, which can also contain submodules themselves
- Connects internal simple and compound modules

**The top level module is the system module**
OMNeT++ – Modules

- **Relationship of modules**
  - Modules communicate by passing messages to each another
  - Implement application-specific functionality
  - Connected by connections
  - Communication by exchanging messages via connections
  - Implemented as C++ objects
    - By using simulation library and in general C++ stuff
  - Topology of module connections are specified in the NED language
OMNeT++ – Parts of Simulation Programs

- **NED-Files**
  - OMNeT++ specific description language

- **Modules**
  - C++ Objects
    - Set of `blabla.cc` and `blabla.h` file
  - Describes behavior of components

- **File: omnetpp.ini**
  - Containing general settings for the execution of the simulation
OMNeT++

NETWORK DESCRIPTION LANGUAGE (NED)
Network Description language (NED)

- The topology of a model is specified using the NED language
- Files containing network descriptions generally have a .ned suffix

Elements of NED

- Channel definitions
- Simple module definitions
- Compound module definitions
- Connections
- Network definitions
NED – Channels

- Specifies a connection type of given characteristics
- Channel name can be used later in the NED description
  - To create connections with these parameters

Parameters
- delay
  - Propagation delay in (simulated) seconds
- error(rate)
  - Probability that a bit is incorrectly transmitted
- datarate
  - Channel bandwidth in bits per second [bps]
NED – Channels – Example

- **Syntax**

  channel ChannelName
  //...
  endchannel

- **Example**

  channel LeasedLine
  delay 0.0018 // sec
  error 1e-8
  datarate 128000 // bit per sec
  endchannel
NED – Simple Module

- **Simple modules are defined in NED file**
  - Simple modules are the basic building blocks for other (compound) modules.

- **Syntax**
  ```plaintext
  simple SimpleModuleName
  
  parameters:
  //...
  
  gates:
  //...
  
  endsimple
  ```
NED – Simple Module

- **Parameters**
  - Values that can be set from a compound module or outside the simulation program, e.g., in configuration files
  - Parameters can be accessed from C++ code using `cModule`'s method `par("name")`

- **Gates**
  - Gates are the connection points of modules.
  - OMNeT++ supports simplex (one-directional) connections
    - There are input and output gates.
  - Messages are sent through output gates and received through input gates.
Traffic generator as simple module

```plaintext
simple TrafficGen
parameters:
interArrivalTime,
numOfMessages   : const,
address         : string;
gates:
in:   from_upper_layer,
     from_physical_layer;
out:  to_upper_layer,
     to_physical_layer;
endsimple
```

Dr. Mesut Güneş

Chapter 4. Introduction to Network Simulators
NED – Simple Module – Gates

- **Gate vectors are supported**
  - A gate vector contains a number of single gates

- **Example**
  ```
  simple RoutingModule
  parameters: // ...
  gates:
    in: input[];
    out: output[];
  endsimple
  ```

- **The sizes of gate vectors are given later**
  - When the module is used as a building block of a compound module type
  - Every instance of the module can have gate vectors of different sizes
NED – Simple and Compound Module

**Compound Modules**
- Module contains submodules, which can also contain submodules themselves.
- Any module type (simple or compound module) can be used as a submodule.
- Like simple modules, compound modules can also have gates and parameters, and they can be used wherever simple modules can be used.
- Connects internal simple and compound modules

**The top level module is the system module**

```
+--------+               +--------+
| system |               | compound|
| module |               | module |
+--------+               +--------+
| simple |               | simple |
| module |               | module |
+--------+               +--------+
| simple |               | simple |
| module |               | module |
```

The top level module is the system module.
NED – Compound Module

- Composed of one or more submodules
- Any module type can be used as a submodule
  - Simple or compound module
- Compound modules can also have gates and parameters
  - Like simple modules
- To the outside: behave like any other modules
  - Must offer gates
- To the inside: composing modules must be able to communicate somehow
  - Their gates must be connected
NED – Compound Module – Syntax

- **Syntax**
  
  ```
  module CompoundModul
      parameters: ...
      gates: ...
      submodules: ...
      connections: 
  endmodule
  ```

- Parameters and gates for compound modules are declared and work in the same way as with simple modules.
### Compound module with parameter

```verbatim
module Router
  parameters:
    packetsPerSecond : numeric,
    bufferSize : numeric,
    numOfPorts : const;
  gates:
    in: inputPort[];
    out: outputPort[];
  submodules: //...
  connections: //...
endmodule
```
NED – Compound Module – Submodules

- Defined in the “submodules:” section of a compound module declaration
- Identified by names
- Instances of a module type, either simple or compound
- Assign values to their parameters
- Specify the size of the gate vectors

Syntax

```plaintext
module CompoundModule
  submodules:
    submodule1: ModuleType1
      parameters: //...
      gatesizes: //...
    submodule2: ModuleType2
      parameters: //...
      gatesizes: //...

endmodule
```
NED – Compound Module – Submodules

- It is possible to create an array of submodules (a module vector).
- Example

```plaintext
module CompoundModule
  parameters:
    size: const;
  submodules:
    submod1: Node[3] //...
    submod2: Node[size] //...
    submod3: Node[2*size+1] //...
endmodule
```
NED – Connections

- **In compound module definition**
- **Specifies how the gates of the compound module and its immediate sub-modules are connected**
- **Only one-to-one connections are supported**
- **A connection**
  - May have attributes (delay, bit error rate or data rate)
  - Or use a named channel

**Example**

```plaintext
module CompoundModule
  parameters: //...
gates: //...
submodules: //...
connections:
  node1.output --> node2.input;
  node1.input <-- node2.output;
  sender.outGate --> rec.inGate;
  sender.inGate <-- Fiber <--
  rec.outGate;
//...
endmodule
```

![Diagram](parent_module.png)
NED – Network Definition

- Module declarations (compound and simple module declarations) just define module types.
- Network definition to get a simulation model
- Syntax is similar to that of a submodule declaration
- Only module types without gates can be used in network definitions
- Assign values to submodule parameters

```plaintext
network wirelessLAN: WirelessLAN
parameters:
    numUsers=10,
    httpTraffic=true,
    ftpTraffic=true,
    distanceFromHub=truncnormal(100,60);
endnetwork
```
OMNeT++

IMPLEMENTATION OF MODULES
C++ Classes

- cMessage
- cSimpleModule
C++ Classes – cMessage

- OMNeT++ uses messages to represent events
- Event represented by an instance of cMessage class
  - Or one of its subclasses
- Messages are sent from one module to another
  - This means that the place where the “event will occur” is the message's destination module
- Events like “timeout expired” are implemented by the module sending a message to itself

- Future Event Set (FES)
  - Events are inserted into the FES
  - Events are processed in strict timestamp order
### C++ Classes – cMessage

- The message class in OMNeT++.
- Represents events, messages, packets or other entities in a simulation

#### Creating a message
- `cMessage *msg = new cMessage();`
- `cMessage *msg = new cMessage("MessageName");`

#### Some methods
- `msg->setKind(kind);`
- `msg->setLength(length);`
- `msg->setByteLength(lengthInBytes);`
- `msg->setPriority(priority);`
- `msg->setBitError(err);`
- `msg->setTimestamp();`
- `msg->setTimestamp(simtime);`
C++ Classes – cSimpleModule

- **Simple modules of name** `MyModule` **implemented by a C++ class of name** `MyModule`
  - Subclassing the `cSimpleModule` class

- **Call the macro** `Define_Module(X)` **after the definition of a C++ class**
  - This macro couples the class to the NED module type

- **Compound modules do not have a corresponding C++ class at all**
### Member functions

- `void initialize()`
- `void activity()`
- `void handleMessage(cMessage *msg)`
- `void finish()`

#### initialize

- OMNeT++ calls the `initialize()` functions of all modules at start time.

#### finish

- Called when the simulation terminates successfully, e.g., for recording of statistics collected during simulation run.
C++ Classes – cSimpleModule

- **handleMessage() and activity() functions**
  - Called during event processing.
  - User implements the model behavior in these functions.
  - `handleMessage()` and `activity()` implement different event processing strategies:
  - For each simple module, the user has to redefine **exactly one** of these functions.

- **handleMessage()**
  - Called by the simulation kernel when the module receives a message

- **activity()**
  - Coroutine-based solution which implements the process interaction approach
C++ Classes – cSimpleModule – Example

```cpp
// file: HelloModule.cc
#include <omnetpp.h>

class HelloModule : public cSimpleModule
{
    protected:
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);
};

// register module class with OMNeT++
Define_Module(HelloModule);

void HelloModule::initialize()
{
    ev << "Hello World!\n";
}

void HelloModule::handleMessage(cMessage *msg)
{
    delete msg; // just discard everything we receive
}
```
C++ Classes – cSimpleModule

- **Member function for sending messages**
  - `send()` family of functions
    - to send messages to other modules
  - `scheduleAt()`
    - to schedule an event (the module “sends a message to itself”)
  - `cancelEvent()`
    - to delete an event scheduled with `scheduleAt()`
C++ Classes – cSimpleModule – Sending messages

- Message objects can be sent through an output gate
- Using one of the following functions
  - `send(cMessage *msg, const char *gateName, int index=0);`
    - `gateName` is the name of the gate in NED file
  - `send(cMessage *msg, int gateId);`
  - `send(cMessage *msg, cGate *gate);`

- Example
  - `send(msg, "outGate");`
  - `send(msg, "outGates", i); // send via outGates[i]`
C++ Classes – cSimpleModule – Self-messages

- Implement timers, or schedule events that occur at some point in the future
- The message would be delivered to the simple module at a later point of time
  - Through `handleMessage()`
  - Module can call `isSelfMessage()` to determine if it is a self-message

- Scheduling an event
  - `scheduleAt(absoluteTime, msg);`
  - `scheduleAt(simtime() + delta, msg);`
OMNeT++

AN EXAMPLE
Example

- The following example shows a useful simple module implementation.
- It demonstrates several of the discussed concepts:
  - constructor, initialize and destructor conventions
  - using messages for timers
  - accessing module parameters
  - recording statistics at the end of the simulation
  - documenting the programmer's assumptions using ASSERT()
Example

```c++
// file: FFGenerator.h
#include <omnetpp.h>

/**
 * Generates messages or jobs; see NED file for more info.
 */
class FFGenerator : public cSimpleModule {

  private:
    cMessage *sendMessageEvent;
    long numSent;

  public:
    FFGenerator();
    virtual ~FFGenerator();

  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};
```
Example

```cpp
// file: FFGenerator.cc
#include "FFGenerator.cc"

// Register module class with OMNeT++
Define_Module(FFGenerator);

FFGenerator::FFGenerator()
{
    sendMessageEvent = NULL;
}

void FFGenerator::initialize()
{
    numSent = 0;
    sendMessageEvent = new cMessage("sendMessageEvent");
    scheduleAt(0.0, sendMessageEvent);
}
```
Example

```cpp
void FFGenerator::handleMessage(cMessage *msg) {
    ASSERT(msg==sendMessageEvent);
    cMessage *m = new cMessage("packet");
    m->setLength(par("msgLength"));
    send(m, "out");
    numSent++;

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

void FFGenerator::finish(){
    recordScalar("packets sent", numSent);
}

FFGenerator::~FFGenerator(){
    cancelAndDelete(sendMessageEvent);
}
```
// file: FFGenerator.ned

class FFGenerator

parameters:
    sendIaTime: numeric;

endclass
Example

- Direct communication of two nodes

```
simple Node
  gates:
    in: inPort;
    out: outPort;
endsimple

module Network
  submodules:
    nodeA: Node;
    nodeB: Node;
  connections:
    nodeA.outPort --> nodeB.inPort;
    nodeA.inPort <-- nodeB.outPort;
endmodule
```
Example

- Communication over a channel

```verilog
module Network
  submodules:
    nodeA: Node;
    nodeB: Node;
  connections:
    nodeA.outPort --> AChannel --> nodeB.inPort;
    nodeA.inPort <-- AChannel <-- nodeB.outPort;
endmodule
```

```
channel AChannel
  delay 0.0015
  error 0.000001
  datarate 1000000
endsimple
```
OMNeT++

BUILDING SIMULATION PROGRAMS
Running the Simulation

- **Linux**
  - opp_makemake -f (generate Makefiles)
  - make depend
  - make
  - ./X

- **Windows (Console)**
  - opp_nmakemake
  - nmake -f Makefile.vc
Summary

- Discussed some network simulation tools
- **ns-2** is one of the most used network simulators
  - Contains many protocol and application components
  - Widely accepted
- **OMNeT++** is an extensive discrete event simulation system
  - Cleanly structured object-oriented design
  - Provides access to both event- and process-based programming style
  - A lot of support functionality