Simulative Evaluation of Internet Protocol Functions

Chapter 3
Introduction to OMNet++

(Acknowledgement: These slides have been prepared by H. Karl [Karl04])

Goals of this chapter

- This chapter introduces a simulation tool that allows to:
  - Specify connections between modules
  - Use an additional programming style for modules
  - Structure large simulation programs
  - Handle random numbers/variates for multiple purposes
  - Supports debugging
- In general, this chapter shows a typical example of a simulation tool

Overview

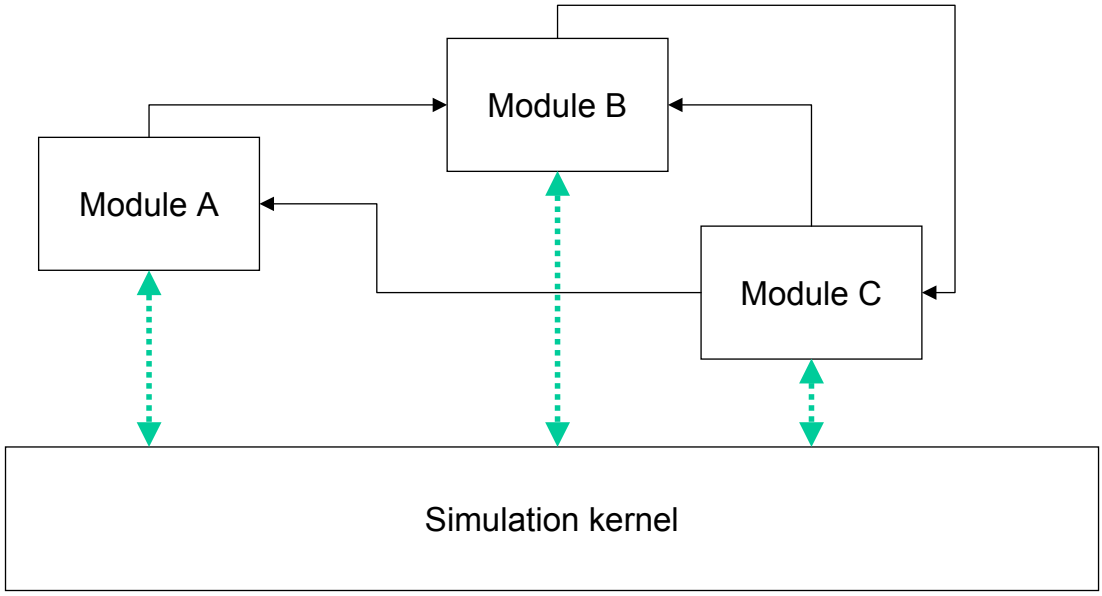
- *OMNeT++ - Some basic concepts*
- Specifying module connections/topology
- Process-based modules
- Multiple random streams
- Working with OMNeT++ simulation programs
- Some odds and ends

OMNet++

- Objective Modular Network Testbed in C++ - OMNet++ for short
  - http://www.omnetpp.org/
- General-purpose tool for discrete event simulations
- Object-oriented design
- General structure:
  - *Modules* implement application-specific functionality
  - Modules can be connected by *connections*
  - Modules communicate by exchanging *messages* via connections
  - Modules are implemented as C++ objects, using support functions from a simulation library
  - Topology of module connections is specified using an OMNeT++-specific language called NED

OMNeT++ - Structure

- Overall structure: Modules with connections + simulation kernel



Modules

- All application-specific functionality is put into modules
- Modules exchange messages; arrival of a message at a module is an event
- As OMNeT++ is object-oriented, all modules are instances of certain classes, representing “module types”
- These classes must be derived from a specific class, `cSimpleModule`, an abstract class which provides basic functionality for a module

Module example

```
#include "omnetpp.h"
class MyModule : public cSimpleModule
{
   // a macro that calls constructor and sets
   // up inheritance relationships:
   Module_Class_Members (MyModule,
                         cSimpleModule, 0)
   // user-specified functionality follows:
   … …
};
// announce this class as a module to OMNeT:
Define_Module (MyModule);
```


Module inheritance

- User-defined module types can be used to derive new module types via standard inheritance techniques

```
#include "MyModule.h"
class MyDerivedModule : public MyModule
{
   Module_Class_Members (MyDerivedModule,
                         MyModule, 0);
   // and again user-specific methods
  // follow
};
// and again make this class known as a module
Define_Module (MyDerivedModule);
```


Hierarchical structure of modules

- Modules of different types can be aggregated together to form a new, larger module type: a *compound* module
- From outside a compound module, interior modules are not visible
- Compound modules behave just like simple modules
- Compound modules do not implement any functionality at all, only combine their constituent modules into a new module
- Derived from `cCompoundModule`
  - Both `cSimpleModule` and `cCompoundModule` are derived from `cModule`
- Enable hierarchical structuring of simulation programs

OMNeT++ basic event handling

- As OMNeT++ is a tool for discrete event simulation, management of events is a primary task
  - Including event loop, managing the future event set, executing the next-event time advance mechanism, etc.
  - Taken care of by the simulation library itself
- Events are generated by modules sending messages to other modules or to themselves (often interpreted as timeouts)
  - Arrival of a message is interpreted as an event
- Module implementations
  - Need not concern themselves with the management of events
  - Only have to implement functionality to process the arrival of messages, and
  - Have to send messages themselves (in general)

Module event handling

- Whenever a message arrives at a module, its `handleMessage()` method is invoked (completely analogous to our little simulation tool)
- `handleMessage()` is a virtual method provided by `cSimpleModule`, which a derived class has to override to implement some real functionality
- `handleMessage()` processes the arrived message, potentially sending new message(s) itself, and returns to the caller (the simulation library)
- To be able to access the arrived message (along with its data), `handleMessage` is passed a pointer to the message, commonly represented by a `cMessage` object
  - Prototype is hence: `void handleMessage (cMessage *)`

Processing messages in `handleMessage`

- Processing messages in general depends on the state of a particular module (e.g., is the server idle or busy?) and also manipulates the state
- Such state variables are part of the information/ knowledge of a module – hence, they are data members of the corresponding class
  - Besides actual state information, all kinds of data pertaining to a module can be stored as data members: parameters for a module, its name or identification (e.g. server number), statistics about metrics, timer values, etc.
  - They are created with the corresponding module
- In `handleMessage`, these variables can be accessed and modified

Initializing module data members

- Commonly, such data members are initialized in the constructor
- However, a module constructor is called during the setup of an OMNeT simulation, when some information might not yet be easily available (e.g., total number of nodes in a simulation, etc.)
- `cSimpleModule` provides the virtual method `initialize()` as a convenient place for setting such data members to well-defined values
- Additionally, `initialize()` can (and should) be used to generate some initial events
  - If no module would generate any events at all, no event would ever happen, and the simulation would be rather static

Shutting down modules

- As a counterpart to `initialize`, there needs to be some way to get data out of modules at the end of a simulation
  - E.g., statistics gathered about some interesting metrics
  - As not every module needs to know (or even should know) the stopping rule, it is not obvious to a module when to output this data
- `cSimpleModule` offers the `finish()` method as a convenient place
- At the end of a simulation run (determined by whatever mechanism), `finish()` of *all* modules is called by the simulation kernel
  - Allows modules to output statistics, perform clean up, …

How to generate events/messages

- So far, only the consumption/reception of events/messages is described
- In `handleMessage()`, a module can decide to
  - Send a message to some other module: an entire family of `send()`-like methods is available
  - Schedule an event to be delivered to itself: `scheduleAt()`
    - E.g., setting a timeout after a packet has been sent
  - Cancel an event that has before been scheduled with `scheduleAt()`: method `cancelEvent()` will delete the specified event from the future event set
    - E.g., canceling a timeout when a packet has been received
- Question: Why is there no possibility to receive messages within `handleMessage()`?

Example: Load generator

- Consider a simple load generator: send a packet, wait some time, send a packet, …
- Class declaration (e.g., loadgen.h)
  - Note that there are no data members (no state) needed for this example

```
#include "omnetpp.h"
class Generator : public cSimpleModule
{
   Module_Class_Members (Generator,
                         cSimpleModule, 0)
   virtual void initialize();
   virtual void handleMessage (cMessage *m);
};
Define_Module (Generator);
```


Example: Load generator

- Implementation (e.g., loadgen.cc)
- Initialization:

```
void Generator::initialize ()
{
   scheduleAt (simTime(), new cMessage);
}
```

  - First argument of scheduleAt is the time when the event should occur. Here: simTime(), which gives the current simulated time (akin to "now")
    → Event will occur immediately, after the `initialize` method has finished
  - Second argument is a pointer to the message to be delivered. Here, just a pointer to a dynamically created message is inserted
    → message has no data, serves as a plain event

Example: Load generator

- Handling the event

```
void Generator::handleMessage (cMessage *m)
{
   cMessage *pkt = new cMessage;
   send (pkt, "out");

   scheduleAt (simTime() + exponential(1.0), m);
}
```

  - First, another message is created (again without data) and sent "somewhere" (we will talk about the meaning of this shortly)
  - Second, the message (event) that triggered this `handleMessage` invocation is scheduled again, to occur at some time now plus an exponentially distributed time (with mean 1.0) in the future

Adding data to messages

- The load generator's message did not carry any data
- How to define messages that can carry data?
  - Rather: how to define „message types" ?
- Message types are defined using a small definition language
  - Definitions are put in *.msg files
  - C++-code is generated automatically, child class of cMessage
  - Each message type ultimately corresponds to a separate C++ class
- Example: Create a message type „customer"
  - Contains a field „paylod" of type integer
  - File: customer.msg

```
message customer {
{
        fields:
          int payload;
};
```


Message inheritance

- Message types can be inherited
- Example: VIP customers with priority

```
message VIPCustomer extends Customer
{
    fields:
      int priority;
};
```


Using messages classes in code

- Example customer

```
#include „customer_m.msg“
Customer newCustomer = new Customer
(„someCustomer“);
newCustomer->setPayload (42);
....
int pl = newCustomer->getPayload();
```

- Getter and setter methods are automatically generated for each field

Detecting message type

- All generated message classes are descendants of cMessage
- In handleMessage, dynamic cast can be used to detect which type of message has actually arrived

```
void dispatcher::handleMessage (cMessage *msg) {
    if (dynamic_cast<VIPCustomer *>(msg) != NULL) {
            VIPCustomer *vipMsg = (VIPCustomer *) msg;
            // do something for important customers
    } else if (dynamic_cast<Customer *>(msg) != NULL) {
            Customer *nMsg = (VIPCustomer *) msg;
            // normal customers ...
    }
    // ...
}
```


Remark: Complete FSM API in OMNeT++

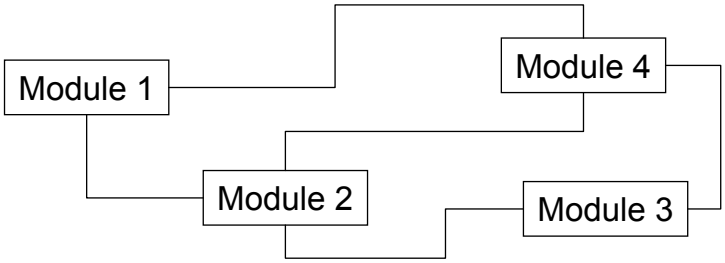
- Besides the simple mechanism to code a finite state machine oneself based on the `handleMessage()` method, OMNeT++ provides an additional API to simplify the programming of finite state machines
- Sets of states and actions for entering and leaving these states can be specified
- API is realized as a set of macros – certainly possible to do it oneself, just a help to structure the code
- Have a look at the manual

Overview

- OMNeT++ - Some basic concepts
- *Specifying module connections/topology*
- Process-based modules
- Multiple random streams
- Working with OMNeT++ simulation programs
- Some odds and ends

Specifying module connections

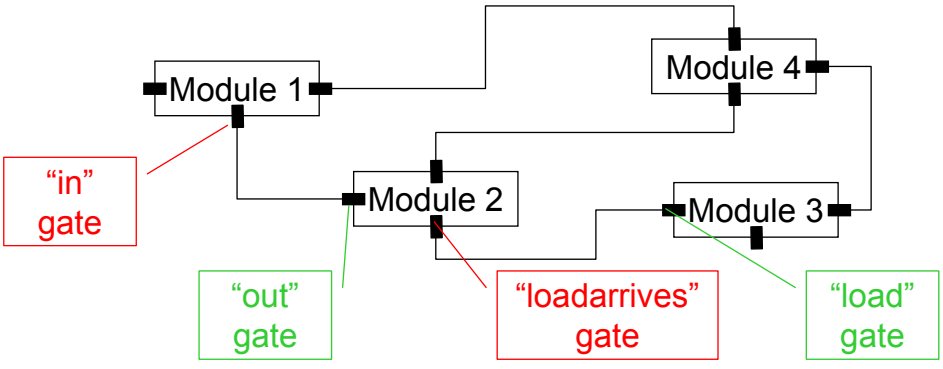
- Is there a nice way to specify which module is connected to which other module?
- Basic idea: somehow specify connections between modules



- But: How can a module distinguish which connection leads to which module?

Gates as connection points for modules

- Additional construct: Module can have (an arbitrary number of) gates
  - Gates are identified by a number or by an index in a named array of gates
  - Gates are unidirectional: either an in-gate or an out-gate
- A connection simply connects an out-gate to an in-gate



Why use gates?

- Adding gates is just another level of complexity
- Why not have the module send directly to the peer module?
  - Reusability: a module should be useful in many contexts, without referring explicitly to other modules directly
  - Gates encapsulate the knowledge “where to” within the module
- Why not send directly to a connection?
  - Similar reason: connections can change, and do not have a real identity of their own (only specified by the two modules they are connecting)
- Gates turn modules into black boxes with well defined interfaces/”service access points with protocols”

Communicating using gates

- Modules can send messages directly to a specific out gate: `send (pkt, "myoutgate1");`
- Modules can find out on which in-gate a message has arrived
  - `cMessage` objects represent not only the message as such, but also meta information about the particular message
  - E.g., `cMessage` provides a method
    `cGate *arrivalGate()`
    which returns a pointer to the gate at which the message arrived
  - `cGate` is a class representing gates in OMNeT++
    - In `send`, the string name of a gate will implicitly be converted to the corresponding pointer

How to specify gates and their connections?

- So far, we have not seen any constructions in the class definition that would specify a gate or a connection between gates
- In fact, OMNet++ uses a separate language to specify topology of an entire network of modules: the NEtwork Description language (NED)
- For each simulation program, a NED file is required
- It describes
  - Channels (to be used as connections between modules)
  - Simple modules (declarations, to be implemented as a C++ class)
  - Compound modules (discussed later)
  - Connections between modules within compound modules

NED channel definition

- Channels represent types of connections
- Parameterized by delay, error rate (uniformly distributed), and data rate
- Example:

```
channel DialUpConnection
    delay normal (0.004, 0.0018)
    error 0.00001
    datarate 14400
endchannel
```

- `DialUpConnection` can later be used as a connection type
- Note: delay is parameterized with a normal distribution, not a specific value -> for every connection of this type, a new delay is randomly chosen from a normal distribution with these parameters
- General concept of NED: where a value is legal, a random distribution is also acceptable

NED simple module definitions – Parameters

- Simple modules are defined in NED file by their
  - Parameters
  - Gates
- Parameters of simple modules
  - Values that can be set from outside the simulation program, e.g., in configuration files
  - Parameters can be easily accessed from the C++ code using `cModule`'s method `par("parametername")`
- Example:

```
simple LoadGenerator
    paramers:
          interarrival_time : numeric const;
    gates: …
endsimple
```

- Using the parameter in `LoadGenerator::someMethod()`:

```
    float intarrtime = par("interarrival_time");
```


NED simple module definitions – Gates

- In NED, the gates of simple modules are defined as well, as either in or out gates
- Example

```
simple DataLinkProtocol
    parameters: …
    gates:
          in:   from_upper_layer,
                from_physical_layer;
          out:  to_upper_layer,
                to_physical_layer;
endsimple
```


NED simple module definitions – Gates

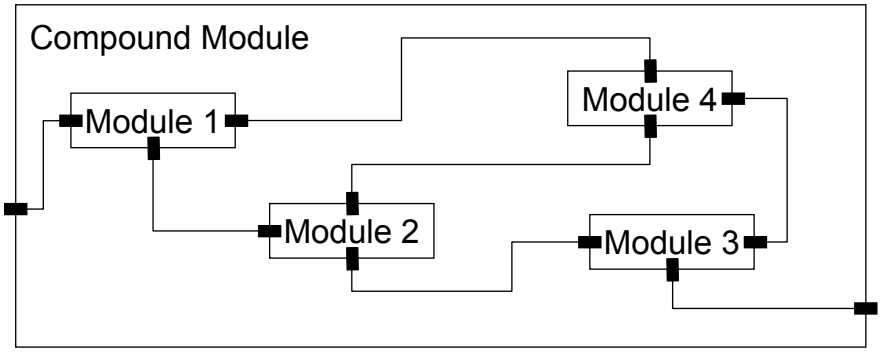
- Gates can also be defined as arrays:

```
simple RoutingModule
    parameters: …
    gates:
        in: input[];
        out: output[];
endsimple
```

- Size of the vectors need not be defined immediately but can be supplied later
  - Size can be different for multiple instances of the same type
  - Example: different `RoutingModules` have different numbers of in and out links

NED compound modules

- Compound modules consist of one or more submodules
- 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
- Relating outside and inside: gates of the compound module are connected to (some) gates of (some) of the composing modules



NED compound modules – Syntax

```
module SomeCompoundModul
    parameters:        …
    gates:             …
    submodules: …
    connections:
endmodule
```

- Parameters of compound modules are similar to simple modules
  - Can be used to set parameters of contained modules
  - Can be used to compute connections (see below)
- Gates of compound modules are identical to simple module gates

NED compound modules – Submodules

- Submodules section of a compound modules defines which modules (and their module types) constitute the compound module
- For parameterized module types, the parameters have to be provided
  - General parameters as well as sizes of gate vectors (if any)
- Submodules can be written as vectors of modules

```
module BigCompound
  parameters:
    num_of_submods: const;
  submodules:
    manyparts: Node[num_o_submods/2];
endmodule
```

- Module type of submodules need not be specified explicitly, can be left as a parameter

NED compound modules – Connections

- Specify connections between gates
  - Of the compound modules to gates of its constituent modules
  - Of the constituent modules themselves
- Connections can not “cross border lines” of modules
- Connections can be endowed with parameters (delay, error, bandwidth) or channel types

NED compound modules – Connections

- Simple programming constructs (for loops, if conditions) allow to construct complicated topologies of connections
  - Example: Normally, all gates must be connected after program has initialized. Sometimes, however, only partially connectivity is desired -> “nocheck” primitive

```
module Stochastic:
  connections nocheck:
    for i=0 .. 9 do
      Sender.outgate[i] → Receiver[i].ingate if
                          uniform(0,1) < 0.3;
    endfor
```


Relating NED file to C++ classes

- Compound modules do not have a corresponding C++ class at all
- Simple modules of name X are implemented by a C++ class of name X
  - Recall the macro `Define_Module (X)` after the definition of a C++ class
  - This macro couples the class to the NED module type
  - Usually put in file X.h and X.cc, but that is not required

Relating NED file to C++ classes

- A module type can be implemented by different classes which share the same interface
  - Define_Module_Like (X, Y): class X implements NED module Y's interface
  - Example: Different types of MAC layers all sharing the same interface
    - `Define_Module_Like (Ethernet_MAC, General_MAC);`
    - `Define_Module_Like (TokenRing_MAC, General_MAC);`
    - `Define_Module_Like (FDDI_MAC, General_MAC);`
  - Such a module type can not directly be used within a compound module, however, it can be used as a placeholder to be instantiated with the actual type of the submodule from some initialization mechanism
  - Example:

```
submodules:
  mac : mac_type like General_MAC
```

  - mac_type can later be assigned any of `Ethernet_MAC, TokenRing_MAC, FDDI_MAC`

Overview

- OMNeT++ - Some basic concepts
- Specifying module connections/topology
- *Process-based modules*
- Multiple random streams
- Working with OMNeT++ simulation programs
- Some odds and ends

Reconsider the load generator module

- The load generator module has been implemented as a finite state machine:
  - Upon receipt of a message (where the content of the message is irrelevant), generate a “load” message to be delivered immediately and a self message to be delivered some time later
  - The self message only serves to trigger a new cycle
- What the load generator actually does:
  - Repeatedly wait some random time, then send a message
  - Modeling such a process as a finite state machine is somewhat awkward
- Would it not be nice to be able to explicitly model such processes that have a distinctive flow of control?

Processes in a simulation

- Such a process should be able to
  - Receive messages
  - Manipulate inner state
  - Send messages
  - And *wait* for some arbitrary amount of time
    - Example: To model the time necessary to analyze a message, a process-based simulation would just wait
- During such waiting, the process would not be able to react to any messages
  - In contrast to a finite-state simulation, which is “always” able to receive messages, as the processing of messages and performing state transmission takes *zero* simulated time

Using processes in OMNeT++

- OMNet++ supports both finite state/event-based simulation as well as process-based simulation
- A module can implement a single process
- Instead of implementing `handleMessage`, a process-based module class implements the `activity` method
  - Strictly exclusive: a module is either event- or process-based
  - Different modules can use different paradigms (one of OMNeT’s main advantages!)

Process-based modules

- Within `activity`, a module can
  - receive messages (different functions available)
  - send messages (different functions available)
  - wait – to suspend its own execution for a specified amount of simulated time
  - scheduleAt – module sends a message to itself
  - cancelEvent – delete an event scheduled with scheduleAt
  - end – terminate its own execution
- Not available in `handleMessage`
  - receive – useless, as `handleMessage` already is the reaction to the reception of a message
  - wait – processing event takes no time for a finite state machine
  - end – due to implementation issues

Process-based modules

- Typical structure of activity: Infinite loop, containing at least a single wait or receive call
  - What if both are absent?
- Local state of a module can be put into data members of the class as in the event-based case, here, an alternative exists
  - `activity` is run as a coroutine, having its own stack, for each module instance
  - Think of `activity` as a thread running in parallel with all other parts of the simulation
  - Hence, `activity` has local variables which maintain value even across receive and wait calls
  - Local module state can be stored in local variables of `activity`

Process-based modules

- Running activity as a coroutine requires to set aside memory for the stack
  - Stack is specified in the constructor for a module, called by the `Module_Class_Members` macro:

```
class ProcessModule : public cSimpleModule {
 public:
   Module_Class_Members (ProcessModule, cSimpleModule, 8192)
```

  - Specifying a stack in this macro distinguishes between an event- and process based module implementation
- `initialize()` is not necessary, can be done at the start of activity
- `finish()` is required to output statistics at end of simulation

Process-based modules – typical setup

```
class MyProcessBasedModule : public cSimpleModule {
   variables for statistics gathering
   activity();
   finish();
}
MyProcessBasedModule::activity () {
   declare local variable to hold state, initialize them
   initialize statistics gathering variables
   while (true)
   { … (usually send, receive, wait, …}
}
MyProcessBasedModule::finish() {
   record statistics data into a file
}
```


Comparing event and process style

- Advantages of process-based style
  - initialize() is not required
  - State can be stored in local variables of activity()
  - Process-based style can be natural programming model
- Advantages of event-based style
  - Lower memory overhead (no separate stack required)
  - Faster: switching to coroutines takes longer than just calling a method (handleMessage)

Comparing event and process style

- Event-based is usually better if
  - Module has little or no state (e.g., data sinks)
  - Module has large state space, where many arbitrary transitions between any two states exist, i.e., there is no clear succession from one state to a successor state – typical for communication protocols
- Rule of thumb:
  - If `activity` looks like a loop which only switches on the message type, convert it to `handleMessage`

Overview

- OMNeT++ - Some basic concepts
- Specifying module connections/topology
- Process-based modules
- *Multiple random streams*
- Working with OMNeT++ simulation programs
- Some odds and ends

Using multiple random streams

- As mentioned already, different random number streams must be used for
  - Different simulation runs
  - Different sources of randomness within a simulation

  to avoid unwanted correlation
- OMNeT++ provides 32 independent random number generators (by default, can be extended)
- Most simply, one generator can be accessed with
  - intrand() – produces an integer between 1..INTRAND_MAX-1
  - randseed(x) – set seed of first generator to x

Using multiple random streams

- To access another one of the 32 available generators, use:
  - `genk_intrand (k)` – random number from generator k
  - `genk_randseed (k, x)` – set seed of generator k to x
- To obtain double randoms between 0 and 1:
  - `dblrand()` and `genk_dblrand()`
- To obtain numbers from certain distributions
  - `double genk_uniform (…)`
  - `double genk_intuniform (…)`
  - `double genk_exponential (…)`
  - `double genk_normal (…)`
  - `double genk_truncnormal (…)`
- Additional distributions can be implemented, and, when registered using Register_Function, even used in NED expressions

Choosing seeds

- Choosing good seeds for RNGs is a difficult problem
- seedtool can be used to generate sufficient number of seeds
- Example: four runs of a simulation that need two different RNGs, each needing at most 10,000,000 random numbers
  - `seedtool g 1 10000000 8` will generate the required eight seed values for streams that are 10,000,000 values apart
  - Details see manual
- How to easily use these seed values in simulations runs will be described later

Overview

- OMNeT++ - Some basic concepts
- Specifying module connections/topology
- Process-based modules
- Multiple random streams
- *Working with OMNeT++ simulation programs*
  - *Building and running*
  - *Debugging support*
  - *Collecting and displaying measurements*
- Some odds and ends

Parts of Omnet programs

- An Omnet program consists of a collection of modules
  - Set of `bla.cc` and `bla.h` file for each class `bla`
- A `*.ned` file is required to specify network connectivity
  - Strictly speaking, not really required: a program can generate the entire network (modules+connections) dynamically – check the manual on how to do this
- Additionally, the file `omnetpp.ini` is required, containing general settings about the execution of the simulation
  - More details later
- How to turn this into an executable?

Bulding an Omnet program

- Usually, a makefile is required, listing dependencies between separate files and instructions about libraries to include in a program
- Omnet provides a little tool `makemake` to generate a makefile
  - Usage: call `makemake` (or `opp_nmakemake`, respectively)
  - Call `make depend` to generate dependencies on .h-files
  - `make` will build the program
- `makemake` collects all `*.cc` files in the current directory and includes them in the program-to-be-built
  - Required libraries and theirs paths are also bound to the program
  - Possible to select between statically and dynamically linked programs

Running an Omnet program

- Omnet programs can be run under two user interfaces: command-line oriented or graphical user interface
  - Selected as a parameter to `makemake`
- Graphical user interface
  - Represents modules and connections, messages traveling along connections
  - Single-step from event to event or run the program with different animation speeds
  - Inspectors for most objects (e.g., double-click on modules)
- Command-line interface
  - Rather uncomfortable, use it to run a program at maximum speed after it has been debugged with the graphical interface

omnetpp.ini

- Initialization file, contains several sections
  - General settings: warning levels, names for output files, seed selection, limits on the simulation and simulated time, maximum memory usage, using parallel execution, etc.
  - Environment-specific settings
    - Command-line: which runs to execute (see below), level of verbosity
    - Graphical environment: speed and level of detail of animation, etc.
  - Parameters: any parameters that were unspecified in the ned-file can be set here
    - Any parameter left unset will be requested from the user at program startup
  - Runs: the same program can be executed multiple times with different parameter settings – a *run*
  - Possible to specify when to start/stop taking measurements

Debugging aids

- Printf-style debugging is supported with the `ev` object, using normal << I/O operator
  - Can be collected in different ways for compound modules
  - Do never use `printf`, `cout` etc. as this will conflict with the graphical environment (will appear in xterm from which program is started)
  - `setPhase()` to set title of windows
- Watches
  - A watch can be declared for primitive variables
  - Watched variables can be inspected/changed in the GUI and output into a snapshot
  - Syntax: `int i; WATCH (i);`

Debugging aids

- Snapshots
  - Dump status of the entire (or selected parts of) simulation into a text file (default: `omnetpp.sna`)
  - Modules, queues, message queues, watched variables, etc. can be included
- Breakpoints
  - In `activity()`, breakpoints can be set by calling function breakpoint, execution will be suspended
  - Only available if user interface supports debugging
- Warnings can be disabled
- Stack usage can be checked
  - Can be substantial with coroutines/`activity()` as they need a lot of space on the stack

Collecting measurements

- Omnet provides several classes to collect results
  - `cStdDev` collects samples, computes mean, standard deviation, number of samples, min, max
    - `cWeightedStdDev`: similar, but weighted, e.g. to compute time-averaged statistics
  - `cLongHistogram`/`cDoubleHistogram`: additionally store an approximated density
- These classes also provide hooks to interact with classes for transient detection and accuracy estimation of results
  - Details later
- Some other classes – read the manual!

Extracting measurements

- Two main supporting mechanisms are offered:
  - Outputting scalar measurements at the end of a simulation run
  - Outputting vector-like measurements during a simulation
- Scalar measurements
  - `recordScalar ("bla", value)` writes an entry into omnetpp.sca
  - `recordStats ("bla", statobject)` writes an entire statistics collection object into `omentpp.sca`
  - Usually done in `finish()` methods of a simple module
- Vector measurements
  - Class `cOutVector` provides functionality
  - Create an object e.g. `value_vector` of this class, along with a name
  - Call `value_vector.record(value)` to write an entry into `omnetpp.vec`
  - Generates a single line in this file
  - Note: vector file is deleted at the beginning of each run

Visualizing measurements

- General remark:
  - Make SURE you can generate visualizations automatically
  - You are going to run many many different simulation experiments with identical result formats
  - You do not want to point-and-click every time the same sequence of commands
  - Hence, visualizations MUST be batch-able!
- Main keywords: perl and gnuplot
  - Or equivalent tools, but nothing GUI-point-and-click-ish!
- One nice intermediate tool provided by Omnet: `plove`
  - Knows how to “read in” Omnet’s vector output
  - Acts as a wrapper around awk and gnuplot
  - Interactively edit the way the graph looks, then safe a script that will recreate the graph when applied to data
- And even better:
  - add some little script code that generates LaTeX-wrappers for the figures as well

Some odds and ends

- Above all: READ THE MANUAL!
- Omnet has an extensive simulation library
  - Contains container classes like queues (`cQueue`) and other useful stuff
- The way messages have been described here is rather inefficient
  - Handling `cPar` objects to manipulate data of a message is processing intensive
  - Subclassing of `cMessage` can provide immense speedup
- Look at coding conventions and tips for speeding up the simulation in the manual
- Omnet can make use of PVM-based parallel execution

Conclusions

- OMNeT++ is an extensive discrete event simulation system
  - Cleanly structure object-oriented design
  - Provides access to both event- and process-based programming style
  - A lot of support functionality
  - Does
- Experience is needed to make best use of its potential
  - But has a comparatively smooth learning curve (other tools are much worse)

Additional References

[Karl04] H. Karl. *Praxis der Simulation.* course slides, Technische Universität Berlin.

[Var04] A. Varga. *OMNeT++: Object-Oriented Discrete Event Simulator.* http://www.omnetpp.org/