BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS

former TECHNICAL UNIVERSITY OF BUDAPEST

Faculty of Electrical Engineering and Informatics

Department of Telecommunications

OMNeT++

Discrete Event Simulation System

Version 2.2

User Manual

---

by András Varga

Last updated: March 18, 2002

Document History

Date	Author	Change
2002/03/18	AV	Documented new ini file options about Envir plugins
2002/01/24	AV	Refinements on the Parsec chapter
2001/10/23	AV	Updated to reflect changes since 2.1 release (see include/ChangeLog)

Contents

Chapter 1

Introduction

1.1 What is OMNeT++?

OMNeT++ is an object-oriented modular discrete event simulator. The name itself stands for Objective Modular Network Testbed in C++. OMNeT++ has its distant roots in OMNeT, a simulator written in Object Pascal by dr. György Pongor.

The simulator can be used for modeling:

- communication protocols
- computer networks and traffic modeling
- multi-processor and distributed systems
- administrative systems
- ... any other system where the discrete event approach is suitable.

An OMNeT++ model consists of hierarchically nested modules. The depth of module nesting is not limited, which allows the user to reflect the logical structure of the actual system in the model structure. Modules communicate with message passing. Messages can contain arbitrarily complex data structures. Modules can send messages either directly to their destination or along a predefined path, through gates and connections.

Modules can have parameters which are used for three main purposes: to customize module behaviour; to create flexible model topologies (where parameters can specify the number of modules, connection structure etc); and for module communication, as shared variables.

Modules at the lowest level of the module hierarchy are to be provided by the user, and they contain the algorithms in the model. During simulation execution, simple modules appear to run in parallel, since they are implemented as coroutines (sometimes termed lightweight processes). To write simple modules, the user does not need to learn a new programming language, but he/she is assumed to have some knowledge of C++ programming.

OMNeT++ simulations can feature different user interfaces for different purposes: debugging, demonstration and batch execution. Advanced user interfaces make the inside of the model visible to the user, allow him/her to start/stop simulation execution and to intervene by changing variables/objects inside the model. This is very important in the development/debugging setPhase of the simulation project. User interfaces also facilitate demonstration of how a model works.

Since it was written in C++, the simulator is basically portable; it should run on most platforms with a C++ compiler. OMNeT++'s advanced user interfaces support X-window, DOS and are portable to Win3.1/Win95/WinNT.

OMNeT++ has been extended to execute the simulation in parallel. Any kind of synchronization mechanism can be used. One suitable synchronization mechanism is the statistical synchronization, for which OMNeT++ provides explicit support.

OMNeT++ Home Page on the Web:

http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

1.2 Where is OMNeT++ in the world of simulation tools?

There are numerous network simulation tools on the market today, both commercial and non-commercial. In this section I will try to give an overview by picking some of the most important or most representative ones in both categories and comparing them to OMNeT++: PARSEC, SMURPH, NS, Ptolemy, NetSim++, C++SIM, CLASS as non-commercial, and OPNET, COMNET III as commercial tools. (The OMNeT++ Home Page contains a list of Web sites with collections of references to network simulation tools where the reader can get a more complete list.) In the commercial category, OPNET is widely held to be the state of the art in network simulation. OMNeT++ is targeted at roughly the same segment of network simulation as OPNET.

Seven issues are examined to get an overview about the network simulation tools:

Detail Level. *Does the simulation tool have the necessary power to express details in the model?* In other words, can the user implement arbitrary new building blocks like in OMNeT++ or he is confined to the predefined blocks implemented by the supplier? Some tools like COMNET III are not programmable by the user to this extent therefore they cannot be compared to OMNeT++. Specialized network simulation tools like NS (for IP) and CLASS (for ATM) also rather fall into this category.

Available Models. *What protocol models are readily available for the simulation tool?* On this point, non-commercial simulation tools cannot compete with some commercial ones (especially OPNET) which have a large selection of ready-made protocol models. OMNeT++ is no exception.

Defining Network Topology. *How does the simulation tool support defining the network topology?* Is it possible to create some form of hierarchy (nesting) or only ”flat” topologies are supported? Network simulation tools naturally share the property that a model (network) consists of ”nodes” (blocks, entities, modules, etc.) connected by ”links” (channels, connections, etc.). Many commercial simulators have graphical editors to define the network; however, this is only a good solution if there is an alternative form of topology description (e.g. text file) which allows one to generate the topology by program. OPNET follows a unique way: the network topology is stored in a proprietary binary file format which can be generated (and read) by the graphical editor and C programs linked against a special library. On the other hand, most non-commercial simulation tools do not provide explicit support for topology description: one must program a ”driver entity” which will boot the model by creating the necessary nodes and interconnecting them (PARSEC, SMURPH, NS). Finally, a large part of the tools that do support explicit topology description supports only flat topologies (CLASS). OMNeT++ probably uses the most flexible method: it has a human-readable textual topology description format (the NED language) which is easy to create with any text-processing tool (perl, awk, etc.), and the same format is used by the graphical editor. It is also possible to create a ”driver entity” to build a network at run-time by program. OMNeT++ also supports submodule nesting.

Programming Model. *What is the programming model supported by the simulation environment?* Network simulators typically use either thread/coroutine-based programming (such as activity() in OMNeT++), or FSMs built upon a handleMessage()-like function. For example, OPNET, SMURPH and NetSim++ use FSMs (with underlying handleMessage()), PARSEC and C++SIM use threads. OMNeT++ supports both programming models; the author does not know of another simulation tool that does so.

Debugging and Tracing Support. *What debugging or tracing facilities does the simulation tool offer?* Simulation programs are infamous for long debugging periods. C++-based simulation tools rarely offer much more than printf()-style debugging; often the simulation kernel is also capable of dumping

selected debug information on the standard output. Animation is also often supported, either off-line (record&playback) or in some client-server architecture, where the simulation program is the ”server” and it can be viewed using the ”client”. Off-line animation naturally lacks interactivity and is therefore little use in debugging. The client-server solution typically has limited power because the simulation and the viewer run as independent operating system processes, and the viewer has limited access to the simulation program's internals and/or it does not have enough control over the course of simulation execution. OPNET has a very good support for command-line debugging and provides both off-line and client-server style animation. NetSim++ and Ptolemy use the client-server method of animation. OMNeT++ goes a different way by linking the GUI library with the debugging/tracing capability into the simulation executable. This architecture enables the GUI to be very powerful: every user-created object is visible (and modifiable) in the GUI via inspector windows and the user has tight control over the execution. To the author's best knowledge, the tracing feature OMNeT++ provides is unique among the C++-based simulation tools.

Performance. *What performance can be expected from the simulation?* Simulation programs typically run for several hours. Probably the most important factor is the programming language; almost all network simulation tools are C/C++-based. Performance is a particularly interesting issue with OMNeT++ since the GUI debugging/tracing support involves some extra overhead in the simulation library. However, in a reported case, an OMNeT++ simulation was only 1.3 slower than its counterpart implemented in plain C (i.e. one containing very little administration overhead), which is a very good showing. A similar result was reported in a performance comparison with a PARSEC simulation.

Source Availability. *Is the simulation library available in source?* This is a trivial question but it immediately becomes important if one wants to examine or teach the internal workings of a simulation kernel, or one runs into trouble because some function in the simulation library has a bug and/or it is not documented well enough. In general it can be said that non-commercial tools (like OMNeT++) are open-source and commercial ones are not. This is also true for OPNET: the source for simulation kernel is not available (although the ready-made protocol models come with sources).

In conclusion, it can be said that OMNeT++ has enough features to make it a good alternative to most network simulation tools, and it has a strong potential to become one of the most widely used network simulation packages in academic and research environments. The most serious shortcoming is the lack of available protocol models, but since this is mostly due to the fact that it is a relatively new simulation tool, with the help of the OMNeT++ user community the situation is likely to become much better in the future.

1.3 Organization of this manual

The manual is organized around the following topics:

- The Chapters 1, 2 and 3 contain introductory material: some overview and an example simulation.
- The second group of Chapters, 4, 5 and 6 are the programming guide. They present the NED language, the simulation concepts and their implementation in OMNeT++, explain how to write simple modules and describe the class library.
- The following chapters, 7, 8 and 9 deal with practical issues like building and running simulations and analyzing results, and present the tools OMNeT++ has to support these tasks.
- Chapter 10 is devoted to the support for distributed execution. itemFinally, Chapter 11 explains the architecture and the internals of OMNeT++. This chapter will be useful to those who want to extend the capabilities of the simulator or want to embed it into a larger application.
- The first two Appendices, A and B, contain a comparison of OMNeT++ and two other important and well-known simulation tools, OPNET and PARSEC.
- Appendice C provides a reference of the NED language.

1.4 History (new)

The development of OMNeT++ started as a semester's programming assignment at the Technical University of Budapest (BME) in 1992. The assignment ("creation of an object-oriented discrete event simulation system in C++") was handed out by Prof. Dr György Pongor, and two students signed up: Ákos Kun and András Varga. The basis for the design was Mr. Pongor's existing simulation software written in Pascal, named OMNeT.

At that time, we wrote the code under Borland C++ 3.1. The idea of multiple runtime environments (a significant addition to the original OMNeT design) was there from the first moment; naturally, we used Turbo Vision (Borland's then successful character-based GUI) for the first 'graphical' user interface. In 1992, we submitted a paper about OMNeT++ to the student's annual university conference (named "TDK") and won first prize in the "Software" section. Later we also won 1st prize in the national "TDK". Then the idea came to port OMNeT++ to Unix (first for AIX on an RS/6000 with 16MB (!) RAM, later Linux), until all development was done in Linux and BC3.1 could no longer be supported.

Well, here's a brief list of events since then – maybe one time I'll make up my mind to enhance them to a whole story...

1994: XEnv (a GUI in pure MOTIF, superceded by Tkenv by now) was written as diploma work

1994: used OPNET for several simulation projects. OPNET features (and flaws) gave lots of ideas how to continue with OMNeT++.

1995: initial version of nedc was written by a group of exchange students from Delft

1996: initial version of PVM support was programmed by Zoltan Vass as diploma work

1997: started working on Tkenv

1997 Dec: added GNED

1997 Sept: web site set up, first public release

1997 Feb-1998 Sept: simulation projects for a small company in Hungary. We used a version of OMNeT++.

1998 March: added Plove

1998 June: animation implemented in GNED

1998 Sept-1999 May: work at MeTechnology (later Brokat) in Leipzig

2000 Jan: MSVC porting

2000 Sept: contributed model repository set up

2000: IP-suite created in Karlsruhe

2001 June: the CVS is hosted in Karlsruhe

...

1.5 Authors

OMNeT++ has been developed mostly by András Varga at the Technical University of Budapest, Department of Telecommunications (BME-HIT).

András Varga BME-HIT, andras@whale.hit.bme.hu

Since leaving the university in 1998, I've been doing the development in my free time.

Several people have worked for shorter periods (1..3 months) on different topics within OMNeT++. Credit for organizing this goes to Dr. György Pongor (BME-HIT, pongor@hit.bme.hu), my advisor at the University. Here is a more-or-less complete list of people:

Old NED compiler, 1992-93:
Ákos Kun BME

JAR compiler (now called NEDC), sample simulations; summer 1995:
Jan Heijmans TU Delft
Alex Paalvast TU Delft
Robert van der Leij TU Delft

New feaures, testing, new examples; fall 1995:
Maurits André TU Delft, M.J.A.Andre@twi.tudelft.nl
George van Montfort TU Delft, G.P.R.vanMontfort@twi.tudelft.nl
Gerard van de Weerd TU Delft, G.vandeweerd@twi.tudelft.nl

JAR (NEDC) support for distributed execution:
Gábor Lencse BME-HIT, lencse@hit.bme.hu

PVM support (as final project), spring 1996:
Zoltán Vass BME-HIT

P^2, k-split algorithms and more, from fall 1996:
Babak Fakhamzadeh TU Delft

We have to mention Dr. Leon Rothkranz from the Technical University of Delft whose work made it possible for the Delft students to come to Budapest in 1995.

Several bugfixes and valuable suggestions for improvements came from the user community of OMNeT++. It would be impossible to mention everyone here, and the list is constantly growing – instead, the README file contains acknowledgements to those I can remember.

Since the summer of 2001, the OMNeT++ sources are kept in the CVS server at the University of Karlsruhe. Credit for setting up and maintaining the CVS server goes to Ulrich Kaage.

The starting point of this manual was the 1995 report of Jan Heijmans, Alex Paalvast and Robert van der Leij.

Chapter 2

Overview

2.1 Modeling concepts

OMNeT++ provides efficient tools for the user to describe the structure of the actual system. Some of the main features are:

- hierarchically nested modules
- modules are instances of module types
- modules communicate with messages through channels
- flexible module parameters
- topology description language

2.1.1 Hierarchical modules

An OMNeT++ model consists of hierarchically nested modules which communicate with messages. OMNeT++ models are often referred to as *networks*. The top level module is the *system module*. The system module contains *submodules*, which can also contain submodules themselves (Fig. 2.1). The depth of module nesting is not limited; this allows the user to reflect the logical structure of the actual system in the model structure.

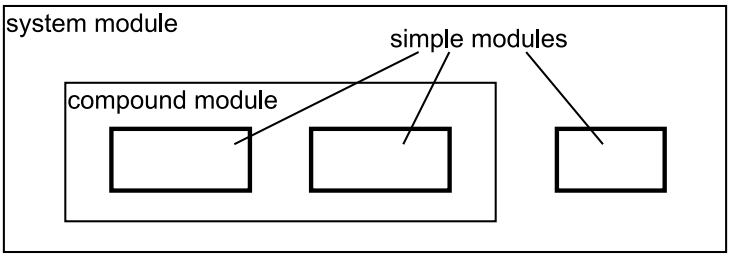

Figure 2.1: Simple and compound modules

Modules that contain submodules are termed *compound modules*, as opposed *simple modules* which are at the 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.

2.1.2 Module types

Both simple and compound modules are instances of *module types*. While describing the model, the user defines module types; instances of these module types serve as components for more complex module types. Finally, the user creates the system module as an instance of a previously defined module type; all modules of the network are instantiated as submodules and sub-submodules of the system module.

When a module type is used as a building block, there is no distinction whether it is a simple or a compound module. This allows the user to split a simple module into several simple modules embedded into a compound module, or vica versa, aggregate the functionality of a compound module into a single simple module, without affecting existing users of the module type.

Module types can be stored in files separately from the place of their actual usage. This means that the user can group existing module types and create *component libraries*. This feature will be discussed later, in Chapter 8.

2.1.3 Messages, gates, links

Modules communicate by exchanging *messages*. In an actual simulation, messages can represent frames or packets in a computer network, jobs or customers in a queuing network or other types of mobile entities. Messages can contain arbitrarily complex data structures. Simple modules can send messages either directly to their destination or along a predefined path, through gates and connections.

The "local simulation time" of a module advances when the module receives a message. The message can arrive from another module or from the same module (*self-messages* are used to implement timers).

Gates are the input and output interfaces of modules; messages are sent out through output gates and arrive through input gates.

Each *connection* (also called *link*) is created within a single level of the module hierarchy: within a compound module, one can connect the corresponding gates of two submodules, or a gate of one submodule and a gate of the compound module (Fig. 2.2).

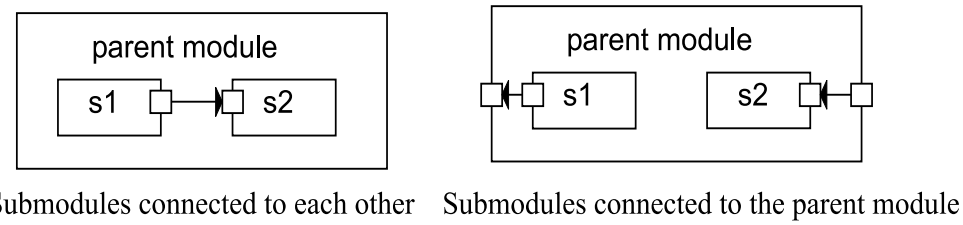

Figure 2.2: Connections

Due to the hierarchical structure of the model, messages typically travel through a series of connections, to start and arrive in simple modules. Such series of connections that go from simple module to simple module are called *routes*. Compound modules act as 'cardboard boxes' in the model, transparently relaying messages between their inside and their outside world.

2.1.4 Link characteristics

Connections can be assigned three parameters which facilitate the modeling of communication networks, but can be useful for other models too:

- propagation delay (sec)
- bit error rate (errors/bit)

- data rate (bits/sec)

Each of these parameters is optional. One can specify link parameters individually for each connection, or define link types (also called *channel types*) once and use them throughout the whole model.

The *propagation delay* is the amount of time the arrival of the message is delayed by when it travels through the channel. Propagation delay is specified in seconds.

The *bit error rate* has influence on the transmission of messages through the channel. The bit error rate is the probability that a bit is incorrectly transmitted. Thus, the probability that a message of *n* bits length is transferred correctly is:

$P(\textit{sent message received properly}) = (1 - \textit{ber})^n$
where *ber* = bit error rate and *n* = number of bits in message.

The message has an error flag which is set in case of transmission errors.

The *data rate* is specified in bits/second, and it is used for transmission delay calculation. The sending time of the message normally corresponds to the transmission of the first bit, and the arrival time of the message corresponds to the reception of the last bit (Fig. 2.3).

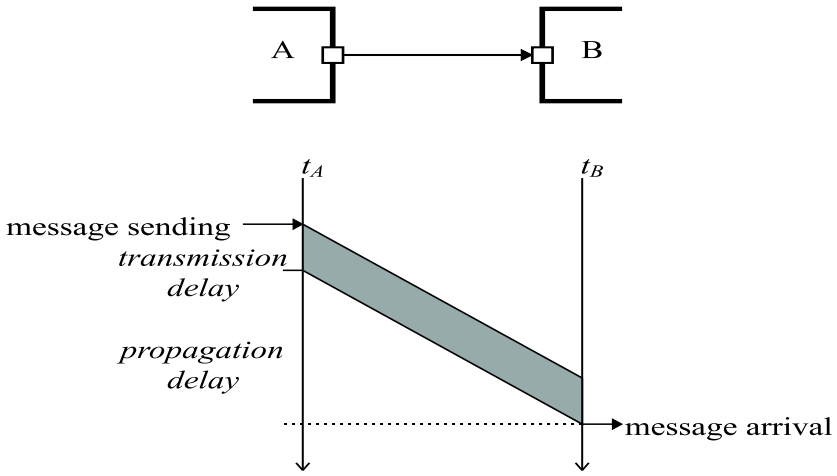

Figure 2.3: Message transmission

The above model is not applicable for modeling some protocols like Token Ring and FDDI where the stations repeat the bits of a frame that arrives on the ring immediately, without waiting for the whole frame to arrive; in other words, frames ”flow through” the stations, being delayed only a few bits. If you want to model such networks, the data rate modeling feature of OMNeT++ cannot be used.

If a message travels along a route, through successive links and compound modules, the model behaves as if each module waited until the last bit of the message arrives and only start its transmission then (Fig. 2.4).

Since the above effect is usually not the desired one, typically you will want to assign data rate to only one connection in the route.

2.1.5 Parameters

Modules can have parameters. Parameters are used for three purposes:

1. to parameterize module topology

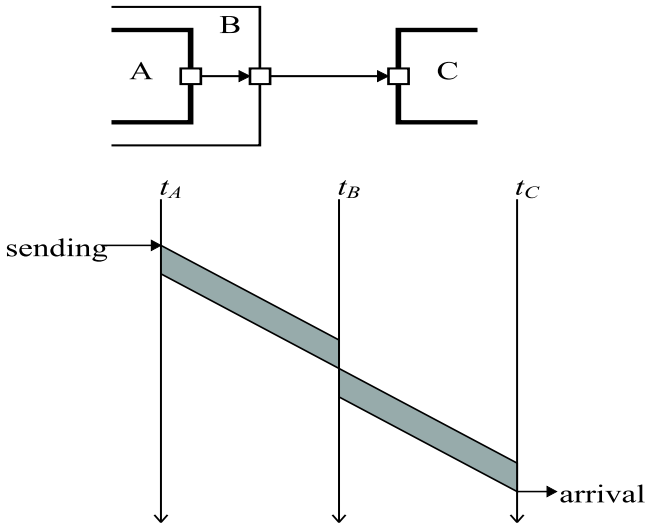

Figure 2.4: Message sending over multiple channels

2. to customize simple module behaviour
3. for module communication, as shared variables

Parameters can take string, numeric or pointer values; numeric values include expressions using other parameters and calling C functions, random variables from different distributions, and values input interactively by the user.

Numeric-valued parameters can be used to construct topologies in a flexible way. Within a compound module, parameters can define the number of submodules, number of gates, and the way the internal connections are made.

Compound modules can pass parameters or expressions of parameters to their submodules. Parameter passing can be done by value or by reference.

During simulation execution, if a module changes the value of a parameter taken by reference, the changed value propagates to other modules. This effect can be used to tune the model or as a second means of module communication. Pointer-valued parameters can be used to implement shared memory.

2.1.6 Topology description method

The user defines the structure of the model in NED language descriptions (Network Description).The NED language will be discussed in detail in Chapter 4.

2.2 Programming the algorithms

The simple modules of a model contain the algorithms as C++ functions. The full flexibility and power of the programming language can be used, supported by the OMNeT++ simulation class library.

OMNeT++ supports a process-style description method for describing activities. During simulation execution, simple module functions appear to run in parallel, because they are implemented as coroutines (also termed lightweight processes). Coroutines were chosen because they allow an intuitive description of the algorithm and they can also serve as a good basis for implementing other description methods like state-transition diagrams or Petri nets.

OMNeT++ has a consistent object-oriented design. One can freely use OOP concepts (inheritance, polymorphism etc) to extend the functionality of the simulator.

Elements of the simulation (messages, modules, queues etc.) are represented as objects. These classes are part of the simulation class library:

- modules, gates, connections etc.
- parameters
- messages
- container classes (e.g. queue, array)
- data collection classes
- statistic and distribution estimation classes (histograms, P^2 algorithm for calculating quantiles etc.)
- transient detection and result accuracy detection classes

The objects are designed so that they can efficiently work together, creating a powerful framework for simulation programming.

2.2.1 Creating simple modules

Each simple module type is implemented with a C++ class. Simple module classes are derived from a simple module base class, by redefining the virtual function that contains the algorithm. The user can add other member functions to the class to split up a complex algorithm; he can also add data members to the class.

It is also possible to derive new simple module classes from existing ones. For example, if one wants to experiment with retransmission timeout schemes in a transport protocol, he can implement the protocol in one class, create a virtual function for the retransmission algorithm and then derive a family of classes that implement concrete schemes. This concept is further supported by the fact that in the network description, actual module types can be parameters.

2.2.2 Object mechanisms

The use of smart container classes allows the user to build *aggregate data structures*. For example, one can add any number of objects to a message object as parameters. Since the added objects can contain further objects, complex data structures can be built.

There is an efficient *ownership* mechanism built in. The user can specify an owner for each object; then, the owner object will have the responsibility of destroying that object. Most of the time, the ownership mechanism works transparently; ownership only needs to be explicitly managed when the user wants to do something non-typical.

The *forEach* mechanism allows one to enumerate the objects inside a container object in a uniform way and do some operation on them. This feature which makes it possible to handle many objects together. (The forEach feature is extensively used by the user interfaces with debugging capability and the snapshot mechanism; see later.)

2.2.3 Derive new classes

It most cases, the functionality offered by the OMNeT++ classes is enough for the user. But if it is needed, one can derive new classes from the existing ones or create entirely new classes. For flexibility, several member functions are declared virtual. When the user creates new classes, certain rules need to be kept so that the object can fully work together with other objects.

2.2.4 Self-describing objects to ease debugging

The class library is designed so that objects can give textual information about themselves. This makes it possible to peek into a running simulation program: through an appropriate user interface, one can examine (and modify) the internal data structures of a running simulation. This feature helps the user to get some insight what is happening inside the model and get hands-on experience.

A unique feature called *snapshot* allows the user to dump the contents of the simulation model or a part of it into a text file. The file will contain textual reports about every object; this can be of invaluable help at times of debugging. Ordinary variables can also be made to appear in the snapshot file. Snapshot creations can be scheduled from within the simulation program or done from the user interface.

2.3 Using OMNeT++

2.3.1 Building and running simulations

This section gives some idea how to work with OMNeT++ in practice: issues like model files, compiling and running simulations are discussed.

An OMNeT++ model consists of the following parts:

- NED language topology description(s) which describe the module structure with parameters, gates etc. They are files with .ned suffix. NED files can be written with any text editor or using the GNED graphical editor.
- Simple modules sources. They are C++ files, with .h/.cc suffix.

The simulation system provides the following components:

- Simulation kernel. This contains the code that manages the simulation and the simulation class library. It is written in C++, compiled and put together to form a library (a file with .a or .lib extension)
- User interfaces . OMNeT++ user interfaces are used with simulation execution, to facilitate debugging, demonstration, or batch execution of simulations. There are several user interfaces, written in C++, compiled and put together into libraries (.a or .lib files).

Simulation programs are built from the above components. First, the NED files are compiled into C++ source code, using the NEDC compiler which is part of OMNeT++. Then all C++ sources are compiled and linked with the simulation kernel and a user interface to form a simulation executable.

Running the simulation and analyzing the results

The simulation executable is a standalone program[1]; thus, it can be run on other machines without OMNeT++ or the model files being present. When the program is started, it reads in a configuration file (usually called omnetpp.ini); it contains settings that control how the simulation is run, values for model parameters, etc. The configuration file can also prescribe several simulation runs; in the simplest case, they will be executed by the simulation program one after another.

The output of the simulation is written into data files: output vector files, output scalar files, and possibly the user's own output files. OMNeT++ provides a GUI tool named Plove to view and plot the contents of output vector files. But it is not expected that someone will process the result files using OMNeT++ alone: output files are text files in a format which (maybe after some preprocessing using sed, awk or perl) can be read into math packages like Matlab or its free equivalent Octave, or imported into spreadsheets like Excel. All these external programs have rich functionality for statistical analysis and visualization, and OMNeT++

[1] as long as it is linked statically

does not try to duplicate their efforts. This manual briefly describes some data plotting programs and how to use them with OMNeT++.

User interfaces

The primary purpose of user interfaces is to make the inside of the model visible to the user, to start/stop simulation execution, and possibly allow the user intervene by changing variables/objects inside the model. This is very important in the development/debugging phase of the simulation project. Just as important, a hands-on experience allows the user to get a 'feel' about the model's behaviour. A nice graphical user interface can also be used to demonstrate how the model works internally.

The same simulation model can be executed with different user interfaces, without any change in the model files themselves. The user would test and debug the simulation with a powerful graphical user interface, and finally run it with a simple and fast user interface that supports batch execution.

Component libraries

Module types can be stored in files separately from the place of their actual usage. This means that the user can group existing module types and create component libraries.

Universal standalone simulation programs

A simulation executable can store several independent models that use the same set of simple modules. The user can specify in the configuration file which model he/she wants to run. This allows one to build one large executable that contains several simulation models, and distribute it as a standalone simulation tool. The flexibility of the topology description language also supports this approach.

2.3.2 What is what in the directories

To help you navigate among files in the OMNeT++ distribution, here's a list what you can find in the different directories.

The omnetpp directory contains the following subdirectories.

The simulation system itself:

omnetpp/	OMNeT++ root directory
bin/	OMNeT++ executables (GNED, nedc, etc.)
include/	header files for simulation models
lib/	library files
bitmaps/	icons that can be used in network graphics
doc/	manual (PDF), readme, license, etc.
html/	manual in HTML
api/	API reference in HTML
src/	OMNeT++ sources
nedc/	NED compiler
sim/	simulation kernel
std/	files for non-distributed execution
pvm/	files for distributed execution over PVM
mpi/	files for distributed execution using MPI
envir/	common code for user interfaces
cmdenv/	command-line user interface
tkenv/	Tcl/Tk-based user interface
gned/	graphical NED editor
plove/	output vector analyzer and plotting tool
utils/	makefile-autocreator etc

There is a tutorial, contributed by Nick van Foreest

tutorial/	the tutorial document
queues/	sample simulation that supports the tutorial
doc_src/	the Latex sources for the tutorial doc

Sample simulations are within the samples directory. Each of the sample directories contain a network description (.ned file) and corresponding simple module code (.h, .cc files). Makefiles are included.

samples/	directories for sample simulations
nim/	a simple two-player game
hcube/	hypercube network with deflection routing
token/	Token-Ring network
fddi/	an accurate FDDI MAC simulation
hist/	demo of the histogram classes
dyna/	dynamic module creation (client-server network)
pvmex/	demonstrates distributed execution
fifo1/	single-server queue
fifo2/	another implementation of a single-server queue
demo/	several sim. models in a single executable

The contrib directory contains material from the OMNeT++ community

contrib/	directory for contributed material
octave/	Octave scripts for result processing
emacs/	NED syntax highlight for Emacs

You may also find additional directories like msvc/, which contains integration components for Microsoft Visual C++, etc.

Chapter 3

An Example: The NIM Game

This chapter contains a full example program that can give you some basic idea of using the simulator. An enhanced version of the NIM example can be found among the sample programs.

Nim is an ancient game with two players and a bunch of sticks. The players take turns, removing 1, 2, 3 or 4 sticks from the heap of sticks at each turn. The one who takes the last stick is the loser.

Of course, building a model of the Nim game is not much of a simulation project, but it nicely demonstrates the modeling approach used by OMNeT++.

Describing the model consists of two phases:

- topology description
- defining the operation of components

3.1 Topology

The game can be modelled in OMNeT++ as a network with three modules: the ”game” (a manager module) and two players. The modules will communicate by exchanging messages. The ”game” module keeps the current number of tokens and organizes the game; in each turn, the player modules receives the number of tokens from the Game module and sends back its move.

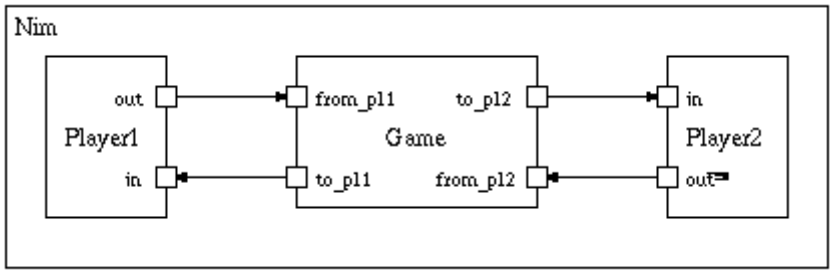

Figure 3.1: Module structure for the Nim game.

Player1, Player2 and Game are simple modules (e.g. they have no submodules.) Each module is an instance of a module type. We’ll need a module type to represent the Game module; let’s call it Game too.

We can implement two kinds of players: SmartPlayer, which knows the winning algorithm, and SimplePlayer, which simply takes a random number of sticks. In our example, Player1 will be a SmartPlayer and

Player2 will be a SimplePlayer.

The enclosing module, Nim is a compound module (it has submodules). It is also defined as a module type of which one instance is created, the system module.

Modules have input and output gates (the tiny boxes labeled in, out, from_player1, etc. in the figure). An input and an output gate can be connected: connections (or links) are shown as in the figure as arrows. During the simulation, modules communicate by sending messages through the connections.

The user defines the topology of the network in NED files.

We placed the model description in two files; the first file defines the simple module types and the second one the system module.

The first file (NED keywords are typed in boldface):

```
//-------------------------------------------------------
// file: nim_mod.ned
// Simple modules in nim.ned
//-------------------------------------------------------


// Declaration of simple module type Game.

simple Game
    parameters:
         num_sticks, // initial number of sticks
         first_move; // 1=Player1, 2=Player2

    gates:
        in:
             from_player1, // input and output gates
             from_player2; // for connecting to Player1/Player2
        out:
             to_player1,
             to_player2;
endsimple


// Now the declarations of the two simple module types.
// Any one of the two types can be Player1 or Player2.

// A player that makes random moves
simple SimplePlayer
    gates:
        in: in; // gates for connecting to Game
        out: out;
endsimple

// A player who knows the winning algorithm
simple SmartPlayer
    gates:
        in: in; // gates for connecting to Game
        out: out;
endsimple
```

The second file:

```
//--------------------------------------------------------------
// file: nim.ned
// Nim compound module + system module
//--------------------------------------------------------------

import "nim_mod";

module Nim
    submodules:
        game: Game
            parameters:
                num_sticks = intuniform(21, 31),
                first_move = intuniform(1, 2);
        player1: SmartPlayer;
        player2: SimplePlayer;
    connections:
        player1.out --> game.from_player1,
        player1.in <-- game.to_player1,
        player2.out --> game.from_player2,
        player2.in <-- game.to_player2;
endmodule

// system module creation
network
    nim: Nim
endnetwork
```

3.2 Simple modules

The module types SmartPlayer, SimplePlayer and Game are implemented in C++, using the OMNeT++ library classes and functions.

Each simple module type is derived from the C++ class cSimpleModule, with its activity() member function redefined. The activity() functions of all simple modules in the network are executed as coroutines, so they appear as if they were running in parallel. Messages are instances of the class cMessage.

We present here the C++ sources of the SmartPlayer and Game module types.

The SmartPlayer first introduces himself by sending its name to the Game module. Then it enters an infinite loop; with each iteration, it receives a message from Game with the number of sticks left, it calculates its move and sends back a message containing the move.

Here's the source:

```
#include <stdio.h>
#include <string.h>
#include <time.h>

#include "omnetpp.h"

// derive SmartPlayer from cSimpleModule
class SmartPlayer : public cSimpleModule
{
    Module_Class_Members( SmartPlayer, cSimpleModule, 8192)
    // this is a macro; it expands to constructor definition etc.
```

```
    // 8192 is the size for the coroutine stack (in bytes)

virtual void activity();
    // this redefined virtual function holds the algorithm
};

// register the simple module class to OMNeT++
Define_Module( SmartPlayer );

// define operations of SmartPlayer
void SmartPlayer::activity()
{
    int move;

    // initialization phase: send module type to Game module
    // create a message with the name "SmartPlayer" and send it to Game

    cMessage *msg = new cMessage("SmartPlayer");
    send(msg, "out");

    // infinite loop to process moves;
    // simulation will be terminated by Game

    for (;;)
    {
        // messages have several fields; here, we'll use the message
        // kind member to store the number of sticks
        cMessage *msgin = receive(); // receive message from Game
        int num_sticks = msgin->kind(); // extract message kind (an int)
                                        // it hold the number of sticks
                                        // still on the stack
        delete msgin;                   // dispose of the message

        move = (num_sticks + 4) % 5; // calculate move
        if (move == 0)                  // we cannot take zero
            move = 1;                   // seems like we going to lose

        ev << "Taking " << move         // some debug output. The ev
           << " out of " << num\_sticks // object represents the user
           << " sticks.\n";             // interface of the simulator

        cMessage *msgout = new cMessage;// create empty message
        msgout->setKind( move );        // use message kind as storage
                                        // for move
        send( msgout, "out"); // send the message to Game
    }
}
```

The Game module first waits for a message from both players and extracts the message names that are also the players' names. Then it enters a loop, with the player_to_move variable alternating between 1 and 2. With each iteration, it sends out a message with the current number of sticks to the corresponding player and gets back the number of sticks taken by that player. When the sticks are out, the module announces the winner and ends the simulation.

The source:

```
//--------------------------------------------------------------
// file: game.cc
// (part of NIM - an OMNeT++ demo simulation)
//--------------------------------------------------------------

#include <stdio.h>
#include <string.h>

#include "omnetpp.h"

// derive Game from cSimpleModule
class Game : public cSimpleModule
{
    Module_Class_Members(Game,cSimpleModule,8192)
      // this is a macro; it expands to constructor definition etc.
      // 8192 is the size for the coroutine stack (in bytes)

    virtual void activity();
      // this redefined virtual function holds the algorithm
};

// register the simple module class to OMNeT++
Define_Module( Game );

// operation of Game:
void Game::activity()
{
    // strings to store player names; player[0] is unused
    char player[3][32];

    // read parameter values
    int num_sticks = par("num_sticks");
    int player_to_move = par("first_move");

    // waiting for players to tell their names
    for (int i=0; i<2; i++)
    {
        cMessage *msg = receive();
        if (msg->arrivedOn("from_player1"))
            strcpy( player[1], msg->name());
        else
            strcpy( player[2], msg->name());
        delete msg;
    }

    // ev represents the user interface of the simulator
    ev << "Let the game begin!\n";
    ev << "Player 1: " << player[1] << "   Player 2: " << player[2]
       << "\n\n";

    do
    {
```

```
        ev << "Sticks left: " << num_sticks << "\n";
        ev << "Player " << player_to_move << " ("
           << player[player_to_move] << ") to move.\n";

        cMessage *msg = new cMessage("", num_sticks);
                        // num\_sticks will be the msg kind

        if (player_to_move == 1)
            send(msg, "to_player1");
        else
            send(msg, "to_player2");

        msg = receive();
        int sticks_taken = msg->kind();
        delete msg;

        num_sticks -= sticks_taken;

        ev << "Player " << player_to_move << " ("
           << player[player_to_move] << ") took "
           << sticks_taken << " stick(s).\n";

        player_to_move = 3 - player_to_move;
    }
    while (num_sticks>0);

    ev << "\nPlayer " << player_to_move << " ("
       << player[player_to_move] << ") won!\n";

    endSimulation();
}
```

3.3 Running the simulation

Once the source files are ready, one needs to compile and link them into a simulation executable. One can specify the user interface to be linked.

Before running the simulation, one can put parameter values and all sorts of other settings into an initialization file that will be read when the simulation program starts:

```
;---------------------
; file: omnetpp.ini
;---------------------

[General]
network = nim
random-seed = 3
ini-warnings = false

[Cmdenv]
module-messages = yes
verbose-simulation = no
```

Suppose we link the NIM simulation with the command line user interface. We get the executable nim (nim.exe under Windows). When we run it, we'll get the following screen output:

```
% ./nim
```

Or:

```
C:\OMNETPP\SAMPLES\NIM> nim.exe

OMNeT++ Discrete Simulation, TUB Dept. of Telecommunications, 1990-97

Preparing for Run #1...

Setting up network 'nim'...

Running simulation...
Let the game begin!
Player 1: SmartPlayer Player 2: SimplePlayer

Sticks left: 29
Player 2 (SimplePlayer) to move.
SimplePlayer is taking 2 out of 29 sticks.
Player 2 (SimplePlayer) took 2 stick(s).
Sticks left: 27
Player 1 (SmartPlayer) to move.
SmartPlayer is taking 1 out of 27 sticks.
Player 1 (SmartPlayer) took 1 stick(s).
Sticks left: 26
[...]
Sticks left: 5
Player 1 (SmartPlayer) to move.
SmartPlayer is taking 4 out of 5 sticks.
Player 1 (SmartPlayer) took 4 stick(s).
Sticks left: 1
Player 2 (SimplePlayer) to move.
SimplePlayer is taking 1 out of 1 sticks.
Player 2 (SimplePlayer) took 1 stick(s).

Player 1 (SmartPlayer) won!
<!> Module nim.game: Simulation stopped with endSimulation().

End run of OMNeT++
```

3.4 Other examples

An enhanced version of the NIM example can be found among the sample programs. It adds a third, interactive player and derives specific player types from a Player abstract class. It also adds the possibility that actual types for player1 and player2 can be specified in the ini file or interactively entered by the user at the beginning of the simulation.

Nim does not show very much of how complex algorithms like communication protocols can be implemented in OMNeT++. To have an idea about that, look at the Token Ring example. It is also extensively commented, though you may need to peep into the user manual to fully understand it.

Other programs in the example manual are Dyna and FDDI. Dyna models a simple client-server network and demonstrates dynamic module creation. The FDDI example is an accurate FDDI MAC simulation which was written on the basis of the ANSI standard.

The following table summarizes the sample simulations:

NAME	TOPIC	DEMONSTRATES
nim	a simple two-player game	module inheritance module type as parameter
hcube	hypercube network with deflection routing	hypercube topology with dimension as parameter topology templates output vectors
token	Token Ring network	ring topology with the number of nodes as parameter using cQueue `wait()` and the putaside-queue output vectors
fifo1	single-server queue	simple module inheritance decomposing `activityf()` into several functions using simple statistics and output vectors printing stack usage info to help optimize memory consumption using finish()
fifo2	another fifo implementation	using `handleMessage()` decomposing `handleMessage()` into several functions the FSM macros simple module inheritance using simple statistics and output vectors using `finish()`
fddi	FDDI MAC simulation	using statistics classes and many other features
hist	demo of the histogram classes	collecting observations into statistics objects saving statistics objects to file and restoring them using the inspect.lst file in Tkenv
dyna	a client-server network	dynamic module creation using `WATCH()` star topology with the number of modules as parameters
pvmex	distributed execution	distributed execution
demo	tour of OMNeT++ samples	shows how to link several sim. models into one executable

Chapter 4

The NED Language

4.1 NED overview

The description of model topology is given in the NED language. The NED language supports modular description of a network. This means that a network description consists of a number of component descriptions (channels, simple/compound module types). The channels, simple modules and compound modules of one network description can be used in another network description. As a consequence, the NED language makes it possible for the user to build his own libraries of network descriptions.

Files containing network descriptions generally have a .ned suffix. Network descriptions are not used directly: they are translated into C++ code by the NEDC compiler, then compiled by the C++ compiler and linked into the simulation executable.

The EBNF description of the language can be found in the appendix.

4.1.1 Components of a NED description

A NED description can contain the following components, in arbitrary number or order:

- import statements
- channel definitions
- simple and compound module declarations
- system module declarations

The rest of this chapter discusses each of these types in detail.

4.1.2 Reserved words

The writer of the network description has to take care that no reserved words are used for names. The reserved words of the NED language are:

```
import include channel endchannel simple endsimple module endmodule
error delay datarate const parameters gates submodules connections
gatesizes on if machines for do endfor network endnetwork nocheck
ref ancestor true false like input numeric string bool char
```

4.1.3 Case sensitivity

The network description and all identifiers in it are case sensitive.

4.2 The import statement

Example:

import ”tkn_mod”, ”tkn2_mod”;

The import statement (the include keyword is also recognized for backwards compatibility) is used to import declarations from other network description files. After importing a network description, one can use the components (channels, simple/compound module types) defined in it.

From the imported files, only the declaration information is used, but *no C++ code is generated*. The consequence is that one has to compile and link each network description, not only the top-level ones.

The user can specify the name of the files with or without the .ned extension. One can also include a path in the filenames, or better, use the NEDC compiler's `-I <path>` command-line option to name the directories where the imported files reside.

4.3 Channel definitions

A channel definition specifies a connection type of given characteristics. The channel name can be used later in the NED description to create connections with these parameters.

Example:

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

Any of the delay, error and datarate parameters are optional and they can appear in any order. The values are NED expressions. This means that they can be constants (integer or real), random values from various distributions, etc.

4.4 Simple module definitions

Simple modules are the basic building blocks for other (compound) modules. A simple module is defined by declaring its parameters and gates.

Example:

```
simple SomeNameForModule
    parameters:
        //...
    gates:
        //...
endsimple
```

4.4.1 Simple module parameters

Parameters are variables that belong to a module. Simple module parameters can be queried and used by simple module algorithms. For example, a parameter called `num_of_messages` can be used by a module called `MsgSource` to determine how many messages it has to generate.

Parameters are declared by listing their names in the parameters: section of a module description. The parameter type can optionally be specified as `numeric`, `numeric const` (or simply const), `bool`, `string`, or `anytype`.

Example:

```
simple MsgSource
    parameters:
        interarrival_time,
        num_of_messages : const,
        address : string;
    gates: //...
endsimple
```

If the parameter type is omitted, numeric is assumed. Practically, this means that you only need to explicitly specify the type for string, bool or char-valued parameters.

Note that the actual parameter values are given later, when the module is used as a building block of a compound module type or as a system module.

When the user writes the word const before the parameter, it is converted to constant; that is, the parameter's value is replaced by its evaluation. This can be important when the original value was a random number or an expression. One is advised to write out the const keyword for each parameter that should be constant.

Beware when using const and by-reference parameter passing (ref modifier, see later) at the same time. Converting the parameter to constant can affect other modules and cause errors that are difficult to discover.

4.4.2 Simple module gates

Gates are the connection points of modules. The starting and ending points of the connections between modules are gates. OMNeT++ supports simplex (one-directional) connections, so there are two kinds of gates: input and output. Messages are sent through output gates and received through input gates.

Gates are identified with their names. Gate vectors are supported: a gate vector contains a number of single gates.

Gates are declared by listing their names in the gates: section of a module description. An empty bracket pair [] denotes a gate vector. Elements of the vector are numbered starting with zero.

Examples:

```
simple DataLink
    parameters: //..
    gates:
        in:  from_port, from_higher_layer;
        out: to_port, to_higher_layer;
endsimple

simple RoutingModule
    parameters: //...
    gates:
        in:  output[];
```

```
        out: input[];
endsimple
```

The sizes of gate vectors are given later, when the module is used as a building block of a compound module type. Thus, every instance of the module can have gate vectors of different sizes.

4.5 Compound module definitions

Compound modules are modules that are composed of one or more submodules. Compound modules, like a simple modules, can have parameters and gates, so a compound module definition looks similar to a simple module definition, except that it also has sections to specify the submodules and connections within the module.

Submodules can either be simple or compound modules, they are equivalent.

Example:

```
module SomeNameForCompoundModule
    parameters:
        //...
    gates:
        //...
    submodules:
        //...
    connections:
        //...
endmodule
```

Any of the above sections (parameters, gates, submodules, connections) is optional.

4.5.1 Compound module parameters

Parameters are declared in the same way as with simple modules. Please refer to Section 4.4.1, ”Simple module parameters”.

Example:

```
module Router
    parameters:
        rte_processing_delay, rte_buffersize,
        num_of_ports : const;
    gates: //...
    submodules: //...
    connections://...
endmodule
```

Compound module parameters can be used in two ways:

- used in expressions for submodule parameter values
- used in defining the internal topology of the network

For example, a parameter called `num_of_ports` can be used to construct a router module with the number of ports as a parameter.

4.5.2 Compound module gates

Gates have the same role and are declared in the same way as with simple modules. Please refer to Section 4.4.2, ”Simple module gates”.

Example:

```
module Router
    parameters: //...
    gates:
        in: input_port[];
        out: output_port[];
    submodules: //...
    connections: //...
endmodule
```

4.5.3 Submodules

Submodules are defined in the submodules: section of a module description. For each submodule, there are sections to define the actual values to be passed to its parameters and the sizes of its gate vectors.

Example:

```
module NameForCompoundModule
    parameters: //...
    gates: //...
    submodules:
        SubModuleName: TypeOfSubModule
            parameters:
                //...
            gatesizes:
                //...
        SecondSubModuleName: TypeOfSecondSubModule
            //...
    connections: //...
endmodule
```

In a submodule definition, one has to supply the name of a previously defined module as the type and a module name. The description of the module type can occur in the same NED file or an imported NED file.

Module vector as submodule

It is possible to create an array of submodules (a module vector). This is done with an expression between brackets right behind the module type name. The expression can refer to module parameters. A zero value as module count is also allowed.

Example:

```
module BigCompound
    parameters:
        num_of_submods: const;
    submodules:
        Submod1: Node[3]
            //...
        Submod2: Node[num_of_submods]
```

```
            //...
        Submod3: Node[(num_of_submods+1)/2]
            //...
endmodule
```

Module type as parameter

Instead of supplying a concrete module type, one can leave it as a parameter. At the same time, to let the NED compiler know what parameters and gates that module has, the user has to supply the name of an existing module type. This is done with the like phrase.

Example:

```
module CompoundModule
    parameters:
        node_type : string;
    gates: //...
    submodules:
        theNode: node_type like GeneralNode
           parameters:
                buffer = 10;
    connections: //...
endmodule
```

The above example means that the type of the submodule theNode is not known in advance; it will be taken from the node_type parameter of CompoundModule which must be a string (for example, ”SwitchingNode”). The module type called GeneralNode must have appeared earlier in the NED files; its declaration will be used to check whether theNode’s parameters and gates exist and are used correctly. The node_type parameter will probably be given an input value somewhere higher in the module hierarchy so that the actual module type can be specified in the ini file or entered interactively.

The GeneralNode module type does not need to be implemented in C++, because no instance of it is created; it is merely used to check the correctness of the NED file.

On the other hand, the actual module type that will be substituted (i.e. SwitchingNode in our case) does not need to be declared in the NED files.

The like phrase enables the user to create *families* of modules that serve similar purposes and implement the same interface (they have the same gates and parameters) and to use them interchangeably in NED files. This scheme directly parallels with the concept of *polymorphism* used in object-oriented programming.

Submodule parameters

Right after the declaration, the values for the parameters of the declared submodules can be specified.

Example:

```
module ManyParameters
    parameters:
        par1, par2, switch;
    submodules:
        Submod1: Node
            parameters:
                p1 = 10,
                p2 = par1+par2,
                p3 = switch==0 ? par1 : par2;
        //...
endmodule
```

Expressions are mostly C-style, and they can contain parameters of the compound module being defined. A separate section is dedicated to expressions. Here, only the modes of parameter passing are discussed.

The default parameter passing method is by value. However, the user can write ref or ancestor before the parameter name. Writing ref means that the parameter is not passed by value, but by reference. This means that instead of the value of the parameter the address of the parameter is passed.

Writing ancestor before the parameter name means that the parameter will be searched upwards, among the parameters of all future enclosing modules of the current module. This reference cannot be resolved or checked by the NEDC compiler; it can only be done at runtime, when the whole network has been built up. The parameter which is found first is used; if no such parameter can be found in any of the enclosing modules, the system will give an error during runtime.

The ancestor and ref modifiers are independent, they can be used together.

For example:

```
simple sub_sub
    parameters:
        s_s_par1, s_s_par2;
endmodule sub_sub


module sub
    parameters:
        s_par;
    submodules:
        child: sub_sub
            parameters:
                s_s_par1 = ref s_par,
                s_s_par2 = ref ancestor m_par2;
endmodule sub


module mod
    parameters:
        m_par1, m_par2;
    submodules:
        child: sub
            parameters:
                s_par = m_par1;
endmodule mod
```

Again, note that the network description compiler can check for the existence of ordinary parameters but not for ancestor parameters (it cannot predict in what modules the current module will be embedded in an actual network description). Parameters taken by reference can be used as a second means of module communication, because during simulation execution, if a module changes the value of a parameter taken be reference, the changed value propagates to other modules. ref parameters can also be used to implement shared memory (see in Chapter 5).

Submodule gate sizes

The sizes of gate vectors are defined with the gatesizes: keyword. Gate vector sizes can be given as constants, parameters or expressions.

An example:

```
simple SimpleType
    gates:
```

```
        in: inputs[]; out: outputs[];
endsimple
module SomeCompound    parameters:
        num: const;
    submodules:
        Submod1: SimpleType
            gatesizes:
                inputs[10], outputs[num];
        //...
endmodule
```

Conditional parameter and gatesize sections

Multiple parameters: and gatesizes: sections can exist in a submodule definition and each of them can be tagged with conditions.

For example:

```
module Serial:
    parameters: count: const;
    submodules:
        node : Node [count]
            parameters:
                position = "middle";
            parameters if index==0:
                position = "beginning";
            parameters if index==count-1:
                position = "end";
            gatesizes:
                in[2], out[2];
            gatesizes if index==0 || index==count-1:
                in[1], in[1];
    connections:
        //...
endmodule
```

If the conditions are not disjunct and a parameter value or a gate size is defined twice, the last definition will take effect, overwriting the former ones. Thus, values intended as defaults should appear in the first sections.

4.5.4 Connections

In a compound module definition, the gates of the compound module and its immediate submodules are connected. In other words, the NED language does not support connections that would cross ”the walls” of a compound module without using gates of that module. Only point-to-point connections are supported.

In summary:

1. The gate of a submodule or enclosing module gate can be connected to another submodule or enclosing module gate
2. Gate direction must be observed (e.g. you cannot connect two submodule output gates)

Connections are specified in the Connections: section of a compound module definition. It lists the connections, separated by semicolons.

Example:

```
module SomeCompound:
    parameters: //...
    gates: //...
    submodules: //...
    connections:
        node1.output --> node2.input;
        node1.input <-- node2.output;
        //...
endmodule
```

Each connection can be:

- direct (that is, no delay, bit error rate or data rate), can use a named channel, or a channel given with delay, error and data rate values;
- single or multiple (loop) connection;
- conditional or non-conditional.

These connection types are described in the following sections.

Single connections and channels

The source gate can be an output gate of a submodule or an input gate of the compound module, and the destination gate can be an input gate of a submodule or an output gate of the compound module.

If the user does not specify a channel, the connection will have no propagation delay, no transmission delay and no bit errors:

```
Sender.outgate --> Receiver.ingate;
```

The arrow can point either left-to-right or right-to-left.

The user can also specify a channel by its name:

```
Sender.outgate --> Dialup14400 --> Receiver.ingate;
```

In this case, the NED sources must contain the definition of the channel.

One can also specify the channel parameters directly:

```
Sender.outgate --> error 1e-5 delay 0.001 --> Receiver.ingate;
```

Either of the parameters can be omitted and they can be in any order.

Loop connections

If submodule or gate vectors are used, it is possible to create more than one connection with one statement. This is termed a *multiple* or *loop connection*.

A multiple connection is created with the for statement:

```
for i=0..4 do
    Sender.outgate[i] --> Receiver[i].ingate
endfor
```

The result of the above loop connection can be illustrated as depicted in Fig. 4.1.

One can place several connections in the body of the for statement, separated by semicolons.

More than one indices can be specified in a for statement, with their own lower and upper bounds. This will be interpreted as nested for statements, the leftmost index being the outermost and the rightmost index being the innermost loop.

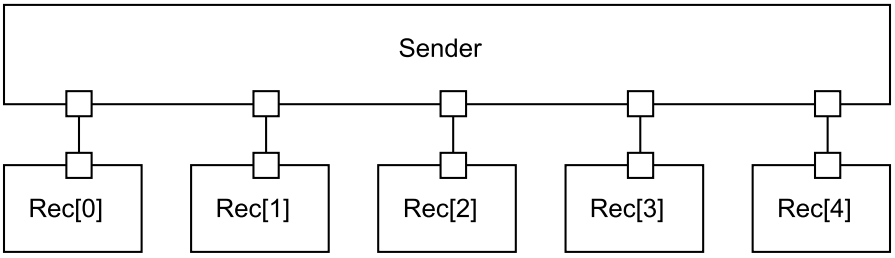

Figure 4.1: Loop connection

```
for i=0..4, j=0..4 do
    //...
endfor
```

One can also use an index in the lower and upper bound expressions of the subsequent indices:

```
for i=0..3, j=i+1..4 do
    //...
endfor
```

In the above example, the following *(i,j)* pairs will be used for the connections inside the for statement:

(0,1) (0,2) (0,3) (0,4) (1,2) (1,3) (1,4) (2,3) (2,4) (3,4)

A gate cannot be used in more than one connection and one connection cannot be made more than once. Consider the following bogus statement:

```
for i = 0..2, j = 0..2 do
    module1.out [i] --> module2.in [i];
endfor
```

It will cause a runtime error: each connection is made twice, as the index variable *j* is not used in the connection. In general, every connection inside a loop should use all the index variables at both sides of the connection.

Conditional connections

Connections can be conditional. This is a conditional connection:

```
for i=0..n do
    Sender.outgate[i] --> Receiver[i].ingate if i%2==0;
endfor
```

This way we connected every second gate.

The nocheck modifier

Conditional connections are especially useful with random numbers when they can create random connections. Here, a problem can be that by default, the simulation program checks if all gates are connected. You can turn off this check by using the nocheck modifier .

This example generates a random subgraph of a full graph:

```
module Stochastic:
    parameters: //..
```

```
    gates: //..
    submodules: //..
    connections nocheck:
        for i=0..9 do
            Sender.outgate[i] --> Receiver[i].ingate
                                  if uniform(0,1)<0.3;
        endfor
endmodule
```

When using nocheck, it is the simple modules' responsibility not to send messages on gates that are not connected.

4.6 Parameterized compound modules

With the help of conditional parameter and gatesize blocks and conditional connections, one can create complex topologies.

4.6.1 Examples

Example 1: Router

The following example contains a router module with the number of ports taken as parameter. The compound module is built using three module types: Application, RoutingModule, DataLink. We assume that their definition is in a separate NED file which we will import.

```
import "modules";
module Router:
    parameters:
        rte_processing_delay, rte_buffersize,
        num_of_ports: const;
    gates:
        in: input_ports[];
        out: output_ports[];
    submodules:
        local_user: Application;
        routing: RoutingModule
            parameters:
                processing_delay = rte_processing_delay,
                buffersize = rte_buffersize;
            gatesizes:
                input[num_of_ports+1],
                output[num_of_ports+1];
        port_if: DataLink[num_of_ports]
            parameters:
                retry_count = 5,
                window_size = 2;
    connections:
        for i=0..num_of_ports-1 do
            routing.output[i] --> port_if[i].from_higher_layer;
            routing.input[i] <-- port_if[i].to_higher_layer;
            port_if[i].to_port --> output_ports[i];
            port_if[i].from_port <-- input_ports[i];
```

```
        endfor;
        routing.output[num_of_ports] --> local_user.input;
        routing.input[num_of_ports] <-- local_user.output;
endmodule
```

Example 2: Chain

For example, one can create a chain of modules like this:

```
module Serial:
    parameters: count: const;
    submodules:
        node : Node [count]
            gatesizes:
                in[2], out[2];
            gatesizes if index==0 || index==count-1:
                in[1], out[1];
    connections:
        for i = 0..count-2 do
            node[i].out[i!=0 ? 1 : 0] --> node[i+1].in[0];
            node[i].in[i!=0 ? 1 : 0] <-- node[i+1].out[0];
        endfor
endmodule
```

Example 3: Binary Tree

Building a binary tree is a good example of using conditional connections:

```
simple BinaryTreeNode:
    gates:
        in: from_up, from_downleft, from_downright;
        out: upward, downleft, downright;
endsimple

module BinaryTree:
    parameters: height: const;
    submodules: node: BinaryTreeNode [ 2^height-1 ];
        //....
    connections:
        for i = 0..2^height-2, j = 0..2^height-2 do
            node[i].upward --> node[j].from_downleft if leftchild(i,j);
            node[i].from_up <-- node[j].downleft if leftchild(i,j);
            node[i].upward --> node[j].from_downright if rightchild(i,j);
            node[i].from_up <-- node[j].downright if rightchild(i,j);
        endfor
        //....
endmodule
```

The dotted lines should be replaced by modules that close the tree at its root and the lower edge. The $leftchild(i,j)$ and $rightchild(i,j)$ functions are:

$$leftchild(i,j) = \begin{cases} 1 & : \quad i = 2j+1 \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$rightchild(i,j) = \begin{cases} 1 & : \quad i = 2j+2 \\ 0 & : \quad \text{otherwise} \end{cases}$$

These formulas can be directly substituted in the NED description, or alternatively, written in C and linked into the simulation executable.

Example 4: Random graph

Conditional connections can also be used to generate random topologies. The following code generates a random subgraph of a full graph:

```
module RandomGraph:
    parameters:
        count: const,
        connectedness; // 0.0<x<1.0
    submodules:
        node: Node [count]
            gatesizes: in[count], out[count];
    connections nocheck:
        for i=0..count-1, j=0..count-1 do
            node[i].out[j] --> node[j].in[i]
                if i!=j and uniform(0,1)<connectedness;
        endfor
endmodule
```

Note that not each gate of the modules will be connected. By default, an unconnected gate produces a run-time error message when the simulation is started, but this error message is turned off here with the nocheck modifier. Consequently, it is the simple modules' responsibility not to send on a gate which is not leading anywhere.

4.6.2 Using const with parameterized topologies

Since parameter values can be used in defining the internal topology of the module, the const modifier has a significant role. Consider the following example:

```
simple Sender
    parameters:
        num_of_outgates;
    gates:
        out: outgate[num_of_outgates];
endsimple Sender

simple Receiver
    gates:
        in: ingate;
endsimple Receiver

module Network;
    parameters:
        num_of_mods: const;
    submodules:
        sender: Sender
            parameters:
                num_of_outgates = num_of_mods;
        receiver: Receiver [num_of_mods]
    connections:
        for i=1..num_of_mods do
```

```
            sender.outgate[i] --> receiver[i].ingate
        endfor;
endmodule

network net: Network
    parameters:
        num_of_mods = normal (5,2);
endnetwork
```

If parameter num_of_mods wasn't const, the following would happen:
normal(5,2) would be substituted for the num_of_mods. There are three places where an evaluation of num_of_mods (that is, normal (5,2)) is done (they are typed in italics in the example). It is likely that these evaluations would not result in the same value, and consequently, the gate vector sizes would not match each other and the end value of the for statement. Thus, the loop connection would not be created properly.

Using const for the parameter num_of_mods prevents this from happening: an evaluation of normal(5,2) is substituted for num_of_mods and an equal number of gates are created.

4.6.3 Design patterns for compound modules

Several approaches can be used when you want to create complex topologies which have a regular structure; three of them are described below.

'Subgraph of a Full Graph'

This pattern takes a subset of the connections of a full graph. A condition is used to "carve out" the necessary interconnection from the full graph:

```
for i=0..N-1, j=0..N-1 do
    node[i].out[...] --> node[j].in[...] if condition(i,j);
endfor
```

The RandomGraph compound module (presented earlier) is an example of this pattern, but the pattern can generate any graph where an appropriate *condition(i,j)* can be formulated. For example, when generating a tree structure, the condition would return whether node *j* is a child of node *i* or vica versa.

Though this pattern is very general, its usage can be prohibitive if the *N* number of nodes is high and the graph is sparse (it has much fewer connections that N^2). The following two patterns do not suffer from this drawback.

'Connections of Each Node'

The pattern loops through all nodes and creates the necessary connections for each one. It can be generalized like this:

```
for i=0..Nnodes, j=0..Nconns(i)-1 do
    node[i].out[j] --> node[rightNodeIndex(i,j)].in[j];
endfor
```

The Hypercube compound module (to be presented later) is a clear example of this approach. BinaryTree can also be regarded as an example of this pattern where the inner j loop is unrolled.

The applicability of this pattern depends on how easily the *rightNodeIndex(i,j)* function can be formulated.

'Enumerate All Connections'

A third pattern is to list all connections within a loop:

```
for i=0..Nconnections-1 do
    node[leftNodeIndex(i)].out[...] --> node[rightNodeIndex(i)].in[...];
endfor
```

The pattern can be used if *leftNodeIndex(i)* and *rightNodeIndex(i)* mapping functions can be sufficiently formulated.

The Serial module is an example of this approach where the mapping functions are extremely simple: *leftNodeIndex(i)=i* and *rightNodeIndex(i)=i+1*. The pattern can also be used to create a random subset of a full graph with a fixed number of connections.

In the case of irregular structures where none of the above patterns can be employed, the user can resort to specifying constant submodule/gate vector sizes and explicitly listing all connections, like he/she would do it in most existing simulators.

4.6.4 Topology templates

Overview

Topology templates are nothing more than compound modules where one or more submodule types are left as parameters (using the **like** phrase of the NED language). You can write such modules which implement mesh, hypercube, butterfly, perfect shuffle or other topologies, and you can use them wherever needed in you simulations. With topology templates, you can reuse *interconnection structure*.

An example: hypercube

The concept is demonstrated on a network with hypercube interconnection. When building an N-dimension hypercube, we can exploit the fact that each node is connected to N others which differ from it only in one bit of the binary representations of the node indices (see Fig. 4.2).

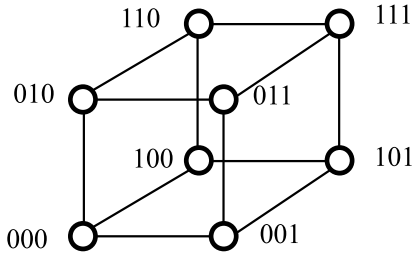

Figure 4.2: Hypercube topology

The hypercube topology template is the following (it can be placed into a separate file, e.g hypercube.ned):

```
simple Node
    gates: out: out[]; in: in[];
endsimple

module Hypercube
    parameters:
        dim, nodetype;
    submodules:
        node: nodetype[2^dim] like Node
    gatesizes:
        out[dim], in[dim];
    connections:
        for i=0..2^dim-1, j=0..dim-1 do
```

```
            node[i].out[j] --> node[i # 2^j].in[j]; // # is bitwise XOR
        endfor
endmodule
```

When you create an actual hypercube, you substitute the name of an existing module type (e.g. Hypercube_PE) for the nodetype parameter. The module type implements the algorithm the user wants to simulate and it must have the same gates that the Node type has. The topology template code can be used through importing the file:

```
import "hypercube.ned"

simple Hypercube_PE
    gates: out: out[]; in: in[];
endsimple

network hypercube: Hypercube
    parameters:
        dim = 4,
        nodetype = "Hypercube_PE";
endnetwork
```

If you put the nodetype parameter to the ini file, you can use the same simulation model to test e.g. several routing algorithms in a hypercube, each algorithm implemented with a different simple module type – you just have to supply different values to nodetype, such as ”WormholeRoutingNode”, ”DeflectionRoutingNode”, etc.

4.7 Network definition

A *network definition* (or *system module definition*) specifies the system module. In its syntax, it is very similar to a submodule declaration. The system definition starts with keyword network and ends with endnetwork.

An example:

```
network modelledNetwork: SomeModule
    parameters:
        par1=10,
        par2=normal(100,20);
endnetwork
```

Here, SomeModule is the name of a compound or a simple module type.

There can be several system definitions in a network description, each one defines a different network. The simulation program built with such a network description is able to run any of them; the desired one can be specified in the config file (see later).

4.8 Support for parallel execution

OMNeT++ simulations can be executed in parallel. This means that different parts of the model execute on different hosts or processors. (We’ll use the term ”host” or ”machine” in this sense.) The unit of granularity is the simple module: one simple module always executes on a single processor.

Parallel execution is also supported by NED: the language provides an elegant way of specifying execution hosts for different modules. We’ll discuss this feature in the following sections.

4.8.1 Extensions to the compound module and system definitions

To support the segmentation of the model for execution of different modules, the compound module definition was extended with the machines: and the on: keywords.

Example:

```
module SomeNameForCompoundModule
    machines: host1, host2, host3, host4;
    parameters: //...
    gates: //...
    submodules:
        submodule1 : submodtype1
            on: host1;
        submodule2 : submodtype2
            on: host2, host3;
        submodule3 : submodtype1
            on: host4;
    connections: //...
endmodule
```

The machines: section lists formal host names which are used in the on: lists of the submodules.

In the example, the second submodule is itself a compound module that can be further subdivided to run on two separate hosts, so its definition must have a machines: section with two parameters. You do not have to propagate host names down to simple module level: you can stop at a compound module which executes on a single host. In other words, a compound module with no machines: section is equivalent to one with one machine parameter.

Of course, you can give the same value to several machine parameters, as to submodule1's in the following example. In this case, the whole compound module will be placed on a single host, as if it never had machine parameters at all.

```
module AnotherCompoundModule
    machines: host1, host2;
    parameters: //...
    gates: //...
    submodules:
        submodule1 : submodtype1
            on: host1, host1, host1;
        //...
    connections: //...
endmodule
```

Host names propagate up to network definition level. Extension to the network definition:

```
network distVector: DistVector
    on: machine1, machine2, machine3;
endnetwork
```

The on: parameters of the network definition can be actual host names, or alternatively, they can be symbolic names that are mapped to actual host names in the config file.

4.8.2 Conditional 'on' sections

Similarly to the parameters: and gatesizes: section, multiple on: sections can exist for the submodules if they are tagged with if phrases.

This makes it possible to control the module distribution with parameters. You can even put different parts of a module vector on different machines using the index operator (see later in the section describing expressions).

Example:

```
module DistVector:
    machines: host1, host2, host3;
    submodules:
        node : Node [count]
            on if index<count*.33: host1;
            on if index>=count*.33 && index<count*.66: host2;
            on if index>=count*.66: host3;
endmodule
network distvector: DistVector
    on machine1, machine2, machine3;
endnetwork
```

4.9 Expressions

In the NED language there are a number of places where expressions are expected.

When such an expression is encountered by the NEDC compiler, it is compiled and it will be evaluated run-time.

Expressions have a C-style syntax. They are built with the usual math operators; they can use parameters taken by value or by reference; call C functions; contain random and input values etc.

4.9.1 Using parameters in expressions (ref and ancestor)

Expressions can use the parameters of the compound module being built. A parameter can be taken by value or by reference. The default is by value; to select by-reference passing for a parameter, you have to use the ref modifier. Parameters passed by reference can be used by a module to propagate values (status info etc.) to other modules.

The ancestor modifier allows one to access parameters from higher in the module hierarchy.

```
module Compound
    parameter: nnn;
    submodules:
        proc: Processor
            parameters:
                par1 = ref nnn / 2,
                par2 = 10 * ancestor par_somewhere_up;
endmodule
```

4.9.2 Operators

The following operators can be used in expressions, in order of precedence:

Operator	Meaning
-, !, ∼	unary minus, negation, bitwise complement
^	power of
∗, /, %	multiply, divide, modulus
+, -	add, subtract
<<, >>	bitwise shifting
&, |, #	bitwise and, or, xor *(^ is reserved for power)*
==	equal
!=	not equal
>, >=	greater, greater or equal
<, <=	less, less or equal
&&, ||, ##	logical operators and, or, xor
?:	the C/C++ "inline if"

4.9.3 The sizeof() and index operators

A useful operator is sizeof(), which gives the size of a vector gate. The index operator gives the index of the current submodule in its module vector.

An example for both:

```
module Compound
    gates: in: fromgens[];
    submodules:
        proc: Processor[ sizeof(fromgens) ];
            parameters: address = 10*(1+index);
    connections:
        for i = 0.. sizeof(fromgens)-1 do
            in[i] --> proc[i].input;
        endfor
endmodule
```

Here, we create as many processors as there are input gates for this compound module in the network. The address parameters of the processors are 10, 20, 30 etc.

4.9.4 Time constants

Anywhere you would put numeric constants (integer or real) to mean time in seconds, you can also specify the time in units like milliseconds, minutes or hours:

```
    ...
    parameters:
        propagation_delay = 560ms, // 0.560s
        connection_timeout = 6m 30s 500ms, // 390.5s
        lunchtime = 0.5h; // 30 min
```

The following units can be used:

ns	nanoseconds	$*10^{-9}$
us	microseconds	$*10^{-6}$
ms	milliseconds	$*10^{-3}$
s	seconds	$*1$
m	minutes	$*60$
h	hours	$*3600$
d	days	$*60*3600$

4.9.5 Random values

OMNeT++ has the following predefined distributions:

- uniform, uniform integer
- exponential
- normal, truncated normal

Each distribution has one or more parameters.

Examples:

```
uniform(0,1)       // uniform in [0,1)
intuniform(-2,2)   // uniform int, limits included: -2,-1,0,1,or 2
exponential(5)     // exponential with mean=5 (thus parameter=0.2)
normal(100,5)      // mean 100, variance 5
truncnormal(5,3)   // normal distr, truncated to nonnegative values
```

The functions all use the random number generator 0. By using the genk_-prefixed versions of the above functions, you can specify which generator should be used. The index of the generator comes as the first argument.

Example:

```
genk_normal(2,100,5) // as normal(100,5), using generator 2
```

The above distributions are implemented with C functions (see later in the Functions section). This also means that you can easily add further ones by writing their code in C++ and using the Register_Function macro. Your distributions will be treated in the same way as the built-in ones.

4.9.6 Input value

The syntax is:

```
input( 10, "Number of processors:" )
```

Or you can omit the prompt text:

```
input( 10ms )
```

Value for input parameters can be given in the config file. If they are not there, the user will be offered a prompt to enter the value.

4.9.7 Functions

In NED expressions, you can use mathematical functions:

- many of the C language's `<math.h>` library functions: `exp()`, `log()`, `sin()`, `cos()`, `floor()`, `ceil()`, `etc.`
- functions that generate random variables: `uniform`, `exponential`, `normal` and others were already discussed.
- user defined functions that can implement new functions or yield random variables of distributions that are originally not built in.

To use user-defined functions, one has to code the function in C++. The C++ function must take 0, 1, 2, or 3 arguments of type double and return a double. The function must be registered in one of the C++ files with the `Define_Function()` macro.

An example function (the following code must appear in one of the C++ sources):

```
#include <omnetpp.h>

double average(double a, double b)
{
  return (a+b)/2;
}

Define_Function(average, 2);
```

The number 2 means that the average() function has 2 arguments. After this, the average() function can be used in NED files:

```
module Compound
    parameter: a,b;
    submodules:
        proc: Processor
            parameters: av = average(a,b);
endmodule
```

An important application of this concept is to extend OMNeT++ with new distributions.

4.9.8 Display strings

Display strings specify the arrangement and appearance of modules in graphical user interfaces (currently only Tkenv): they control how the objects (compound modules, their submodules and connections) are displayed. Display strings occur in NED description's display: phrases.

The display string format is a semicolon-separated list of tags. Each tag consists of a key (usually one letter), an equal sign and a comma-separated list of parameters, like:

```
"p=100,100;b=60,10,rect;o=blue,black,2"
```

Parameters may be omitted also at the end and also inside the parameter list, like:

```
"p=100,100;b=,,rect;o=blue,black"
```

Module/submodule parameters can be included with the `$name` notation:

```
"p=$xpos,$ypos;b=rect,60,10;o=$fillcolor,black,2"
```

Objects that may have display strings are:

- compound modules (as the enclosing module in the drawing),
- submodules
- connections

Tags used in submodule display strings:

Tag	Meaning
p=*xpos*,*ypos*	Place submodule at (*xpos*,*ypos*) pixel position, with the origin being the top-left corner of the enclosing module. Defaults: an appropriate automatic layout is where submodules do not overlap. If applied to a submodule vector, *ring* or *row* layout is selected automatically.
p=*xpos*,*ypos*,**row**,*deltax*	Used for module vectors. Arranges submodules in a row starting at (*xpos*,*ypos*), keeping *deltax* distances. Defaults: *deltax* is chosen so that submodules do not overlap. **row** may be abbreviated as **r**.
p=*xpos*,*ypos*,**column**,*deltay*	Used for module vectors. Arranges submodules in a column starting at (*xpos*,*ypos*), keeping *deltay* distances. Defaults: *deltay* is chosen so that submodules do not overlap. **column** may be abbreviated as **col** or **c**.
p=*xpos*,*ypos*,**matrix**, *itemsperrow*,*deltax*,*deltay*	Used for module vectors. Arranges submodules in a matrix starting at (*xpos*,*ypos*), at most *itemsperrow* submodules in a row, keeping *deltax* and *deltay* distances. Defaults: *itemsperrow*=5, *deltax*,*deltay* are chosen so that submodules do not overlap. **matrix** may be abbreviated as **m**.
p=*xpos*,*ypos*,**ring**,*width*,*height*	Used for module vectors. Arranges submodules in an ellipse, with the top-left corner of its bounding boxes at (*xpos*,*ypos*), with the *width* and *height*. Defaults: *width*=40, *height*=24 **ring** may be abbreviated as **ri**.
p=*xpos*,*ypos*,**exact**,*deltax*,*deltay*	Used for module vectors. Each submodule is placed at *(xpos+deltax, ypos+deltay)*. This is useful if *deltax* and *deltay* are parameters (e.g.:*"p=100,100,exact,$x,$y"*) which take different values for each module in the vector. Defaults: *none* **exact** may be abbreviated as **e** or **x**.
b=*width*,*height*,**rect**	Rectangle with the given *height* and *width*. Defaults: *width*=40, *height*=24
b=*width*,*height*,**oval**	Ellipse with the given *height* and *width*. Defaults: *width*=40, *height*=24

o=*fillcolor*,*outlinecolor*,*borderwidth*	Specifies options for the rectangle or oval. Any valid Tk color specification is accepted: English color names or *#rgb*, *#rrggbb* format (where *r*,*g*,*b* are hex digits). Defaults: *fillcolor*=#8080ff (a lightblue), *outlinecolor*=black, *borderwidth*=2
i=*iconname*	Use the named icon. No default. If no icon name is present, *box* is used.

Examples:

```
"p=100,60;i=workstation"
"p=100,60;b=30,30,rect;o=4"
```

Tags used in enclosing module display strings:

Tag	Meaning
p=*xpos*,*ypos*	Place enclosing module at (*xpos*,*ypos*) pixel position, with (0,0) being the top-left corner of the window.
b=*width*,*height*,**rect**	Display enclosing module as a rectangle with the given *height* and *width*. Defaults: *width*, *height* are chosen automatically
b=*width*,*height*,**oval**	Display enclosing module as an ellipse with the given *height* and *width*. Defaults: *width*, *height* are chosen automatically
o=*fillcolor*,*outlinecolor*,*borderwidth*	Specifies options for the rectangle or oval. Any valid Tk color specification is accepted: English color names or *#rgb*, *#rrggbb* format (where *r*,*g*,*b* are hex digits). Defaults: *fillcolor*=#8080ff (a lightblue), *outlinecolor*=black, *borderwidth*=2

Tags used in connection display strings:

Tag	Meaning
m=auto **m=north** **m=west** **m=east** **m=south**	Drawing mode. Specifies the exact placement of the connection arrow. The arguments can be abbreviated as a,e,w,n,s.
m=manual,*srcpx*,*srcpy*, *destpx*,*destpy*	The manual mode takes four parameters that explicitly specify anchoring of the ends of the arrow: *srcpx*, *srcpy*, *destpx*, *destpy*. Each value is a percentage of the width/height of the source/destination module's enclosing rectangle, with the upper-left corner being the origin. Thus, `m=m,50,50,50,50` would connect the centers of the two module rectangles.
o=*color*,*width*	Specifies the appearance of the arrow. Any valid Tk color specification is accepted: English color names or #rgb, #rrggbb specification (where r,g,b are hex digits). Defaults: *color*=black, *width*=2

Examples:

```
"m=a;o=blue,3"
```

4.10 GNED – Graphical NED Editor

The GNED editor allows you to design compound modules graphically. GNED works with NED files – it doesn't use any nasty internal file format. You can load any of your existing NED files, edit the compound modules in it graphically and then save the file back. The rest of the stuff in the NED file (simple modules, channels, networks etc.) will survive the operation. GNED puts all graphics-related data into display strings.

GNED works by parsing your NED file into an internal data structure, and regenerating the NED text when you save the file. One consequence of this is that indentation will be "canonized" – hopefully you consider this fact as a plus and not as a minus. Comments in the original NED are preserved – the parser associates them with the NED elements they belong to, so comments won't be messed up even if you edit the graphical representation to death by removing/adding submodules, gates, parameters, connections, etc.

GNED is now a fully two-way visual tool. While editing the graphics, you can always switch to NED source view, edit in there and switch back to graphics. Your changes in the NED source will be immediately backparsed to graphics; in fact, the graphics will be totally reconstructed from the NED source and the display strings in it.

GNED is still under development. There are some missing functions and bugs, but overall it should be fairly reliable. See the TODO file in the GNED source directory for problems and missing features.

Comment parsing:

It is useful to know how exactly GNED identifies the comments in the NED file. The following (maybe a bit long) NED code should explain it:

```
// ----------------------------------------------------------------
// File: sample.ned
//
// This is a file comment. File comments reach from the top of
// the file till the last blank line above the first code line.
// ----------------------------------------------------------------
//

// The file comment can also contain blank lines, so this is
// still part of the above file comment.
//
// Module1 --
//
// This is a banner comment for the Module1 declaration below.
// Banner comments can be multi-line, but they are not supposed
// to contain blank lines. (Otherwise the lines above the blank
// one will be taken as part of a file comment or trailing comment.)
//
module Module1
    submodules: // and this is right-comment
// This is another banner comment, for the submodule
        submod1: Module;
            display: ''p=120,108;b=96,72,rect'';
            connections:
                out --> submod1.in; // Right-comments can also be
```

```
                                    // multi-line.
endmodule

// Finally, this is a trailing comment, belonging to the above
// module. It may contain blank lines. Trailing comments are
// mostly used to put separator lines into the file, like this:
// ---------------------------------------------------------------
// Module2 --
//
// an empty module
//
module Module2
endmodule
```

Key/mouse bindings:

In graphics view, there are two editing modes: draw and select/mode. The mouse bindings are the following:

Mouse	Effect
In *draw* mode:	
Drag out a rectangle in empty area:	create new submodule
Drag from one submodule to another:	create new connection
Click in empty area:	switch to select/move mode
In *select/move* mode:	
Click submodule/connection:	select it
Ctrl-click submodule/conn.:	add to selection
Click in empty area:	clear selection
Drag a selected object:	move selected objects
Drag submodule or connection:	move it
Drag either end of connection:	move that end
Drag corner of (sub)module:	resize module
Drag starting in empty area:	select enclosed submodules/connections
Del key	delete selected objects
Both editing modes:	
Right-click on module/submodule/connection:	popup menu
Double-click on submodule:	go into submodule
Click name label	edit name
Drag&drop module type from the tree view to the canvas	create a submodule of that type

Chapter 5

Simple Modules

The activities of simple modules are implemented by the user. The algorithms are programmed in C++, using the OMNeT++ class library. The following sections contain a short introduction to discrete event simulation in general, how its concepts are implemented in OMNeT++, and gives an overview and practical advice on how to design and code simple modules.

5.1 Simulation concepts

This section contains a very brief introduction into how Discrete Event Simulation (DES) works, in order to introduce terms we'll use when explaining OMNeT++ concepts and implementation. If you're familiar with DES, you can skip this section.

5.1.1 Discrete Event Simulation

A *Discrete Event System* is a system where state changes (events) happen at discrete points of time, and events take zero time to happen. It is assumed that nothing (i.e. nothing interesting) happens between two consecutive events, that is, no state change takes place in the system between the events (in contrast to *continuous* systems where state changes are continuous). Those systems that can be viewed as Discrete Event Systems can be modeled using Discrete Event Simulation. (Continuous systems are modelled using differential equations and suchlike.)

For example, computer networks are usually viewed as discrete event systems. Some of the events are:

- start of a packet transmission
- end of a packet transmission
- expiry of a retransmission timeout

This implies that between two events such as "start of a packet transmission" and "end of a packet transmission", nothing interesting happens. That is, the packet's state remains "being transmitted". Note that the definition of events and states always depends on the intent and purposes of the person doing the modeling. If we were interested in the transmission of individual bits, we would have included something like "start of bit transmission" and "end of bit transmission" among our events.

The time when events occur is often called *event timestamp*; with OMNeT++ we'll say *arrival time* (because in the class library, the word "timestamp" is reserved for a user-settable attribute in the event class). Time within the model is often called *simulation time*, *model time* or *virtual time* as opposed to real time or CPU time or which refers to how long the simulation program has been running or how much CPU time it has consumed.

5.1.2 The event loop

Discrete event simulations maintain a set of future events, in a data structure often called FES (Future Event Set). Such simulators usually work according to the following pseudocode:

```
initialize -- this includes building the model and
              inserting initial events to FES

while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    process event
    (processing may insert new events in  FES or delete existing ones)
}
finish simulation (write statistical results, etc.)
```

The first, initialization step usually builds the data structures representing the simulation model, calls any user-defined initialization code, and inserts initial events into the FES to ensure that the simulation can start. Initialization strategy can be quite different from one simulator to another.

The subsequent loop consumes events from the FES and processes them. Events are processed in strict timestamp order in order to maintain causality, that is, to ensure that no event may have an effect on earlier events.

Processing an event involves calls to user-supplied code. For example, using the computer network simulation example, processing a ”timeout expired” event may consist of re-sending a copy of the network packet, updating the retry count, scheduling another ”timeout” event, and so on. The user code may also remove events from the FES, for example when cancelling timeouts.

Simulation stops when there are no more events left (this happens rarely in practice), or when it isn't necessary for the simulation to run further because the model time or the CPU time has reached a given limit, or because the statistics have reached the desired accuracy. At this time, before the program exits, the simulation programmer will typically want to record statistics into output files.

5.1.3 Simple modules in OMNeT++

The user creates simple module types are by subclassing the cSimpleModule class, which is part of the OMNeT++ class library. cSimpleModule, just as cCompoundModule, is derived from a common base class, cModule.

cSimpleModule, although stuffed with simulation-related functionality, doesn't do anything useful by itself. The simulation programmer has to redefine some virtual member functions to make it do useful work.

These member functions are the following:

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

In the initialization step, OMNeT++ builds the network: it creates the necessary simple and compound modules and connects them according to the NED definitions. OMNeT++ also calls the initialize() functions of all modules.

The activity() and handleMessage() functions are called during event processing. This means that the user will implement the model's behavior in these functions. Activity() and handleMessage() implement different event processing strategies: for each simple module, the user has to redefine exactly one of these functions. activity() is a coroutine-based solution which implements the process interaction approach (coroutines are non-preemptive [cooperative] threads), and handleMessage() is a function called for each event. Modules written with these functions can be freely mixed within a simulation model, so you can choose per-module basis.

The finish() functions are called when the simulation terminates successfully. It is the place of writing statistics.

All these functions will be discussed later in detail.

5.1.4 Events in OMNeT++

OMNeT++ uses messages to represent events. Each event is represented by an instance of the cMessage class or one its subclasses; there is no separate event class. Messages are sent from one module to another – this means that the place where the "event will occur" is the *message's destination module*, and the model time when the event occurs is the *arrival time* of the message. Events like "timeout expired" are implemented with the module sending a message to itself.

Simulation time in OMNeT++ is stored in the C++ type simtime_t, which is a typedef for double.
Events are consumed from the FES in arrival time order, to maintain causality. More precisely, given two messages, the following rules apply:

1. the message with **earlier arrival time** is executed first. If arrival times are equal,
2. the one with **smaller priority value** is executed first. If priorities are the same,
3. the one **scheduled or sent earlier** is executed first.

Priority is a user-assigned integer attribute of messages.

Storing simulation time in doubles may sometimes cause inconveniences. Due to finite machine precision, two doubles calculated in two different ways do not always compare equal even if they theoretically should be. This means that if you want to explicitly rely on the arrival times of two events being the same, you should take care that simulation times which should be equal are calculated in exactly the same way. Another possible approach is to avoid equal arrival times, for example by adding/subtracting small values to schedule times to ensure specific execution order (*inorder_epsilon*).

We also thought about some *simtime_precision* parameter in the simulation kernel that would force t_1 and t_2 to be regarded equal if they are "very close" (if they differ less than *simtime_precision*). However, it is not at all clear how small *simtime_precision* should be; the mechanism incurs some run-time overhead; and all in all I'm not sure the whole thing would be of more benefit than trouble.

5.1.5 FES implementation

The implementation of the FES is a crucial factor in the performance of a discrete event simulator. In OMNeT++, the FES is implemented with *binary heap*, the most widely used data structure for this purpose. Heap is also the best algorithm we know, although exotic data structures like *skiplist* may perform better than heap in some cases. In case you're interested, the FES implementation is in the cMessageHeap class, but as a simulation programmer you won't ever need to care about it.

5.2 Defining simple module types

5.2.1 Overview

The C++ implementation of a simple module consists of:

- declaration of the module class: your class subclassed from cSimpleModule (either directly or indirectly)
- a module type registration (Define_Module or Define_Module_Like macro)
- implementation of the module class

For example, the C++ source for a Sliding Window Protocol implementation might look like this:

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

// module class declaration:
class SlidingWindow : public cSimpleModule
{
    Module_Class_Members(SlidingWindow,cSimpleModule,8192)
    virtual void activity();
};

// module type registration:
Define_Module( SlidingWindow );

// implementation of the module class:
void SlidingWindow::activity()
{
  int window_size = par("window_size");
...
}
```

In order to be able to refer to this simple module type in NED files, we should have an associated NED declaration which might look like this:

```
// file: swp.ned
simple SlidingWindow
    parameters:
        window_size: numeric const;
    gates:
        in: from_net, from_user;
        out: to_net, to_user;
endsimple
```

5.2.2 The module declaration

The module declaration

- announces that you're going to use the class as a simple module type
- associates the module class with an interface declared in NED

Forms of module declaration

Module declarations can take two forms:

```
Define_Module(classname);
Define_Module_Like(classname, neddeclname);
```

The first form associates the class (subclassed from cSimpleModule) with the NED simple module declaration of the same name. For example, the

```
Declare_Module(SlidingWindow);
```

line would ensure that when you create an instance of SlidingWindow in your NED files, the module has the parameters and gates given in the simple SlidingWindow NED declaration, and the implementation will be an instance of the SlidingWindow C++ class.

The second form associates the class with a NED simple module declaration of a different name. You can use this form when you have several modules which share the same interface. This feature will be discussed in detail in the next section.

Header files

Module declarations should not be put into header files, because they are macros expanding to lines for which the compiler generates code.

Compound modules

All module types (including compound modules) need to have module declarations. For all compound modules, the NEDC compiler generates the Define_Module(..) lines automatically. However, it is your responsibility to put Define_Module(..) lines into one of the C++ sources for all your simple module types.

Implementation

Unless you are dying to learn about the dirty internals, you may just as well skip this section. But if you're interested, here it is: Define_Module (and also Define_Module_Like) is a macro which expands to a function definition plus the definition of a global object, something like this ugly code (luckily, you won't ever need to be interested in it):

```
static cModule *MyClass__create(const char *name, cModule *parentmod)
    return (cModule *) new MyClass(name, parentmod);

cModuleType MyClass__type("MyClass","MyClass",
 (ModuleCreateFunc)MyClass__create);
```

The cModuleType object can act as a factory: it is able to create an instance of the given module type. This, together with the fact that all cModuleType objects are available in a single linked list, allows OMNeT++ to instantiate module types given only their class names as strings, without having to include the class declaration into any other C++ source.

The global object also stores the name of the NED interface associated with the module class. The interface description object (another object, generated by nedc) is looked up automatically at network construction time. Whenever a module of the given type is created, it will automatically have the parameters and gates specified in the associated interface description.

5.2.3 Several modules, single NED interface

Suppose you have three different C++ module classes (TokenRing_MAC, Ethernet_MAC, FDDI_MAC) which have identical gates and parameters. Then you can create a single NED declaration, General_MAC for them and write the following module declarations in the C++ code:

```
Define_Module_Like(TokenRing_MAC, General_MAC);
Define_Module_Like(Ethernet_MAC, General_MAC);
Define_Module_Like(FDDI_MAC, General_MAC);
```

In this case, you won't be able to directly refer to the TokenRing_MAC, Ethernet_MAC, FDDI_MAC module types in your NED files. For example, you cannot write

```
module PC
    submodules:
        mac: Ethernet_MAC; // error: Ethernet_MAC not defined
...
endmodule
```

However, you can pass the module type in a string-valued parameter to the compound module:

```
module PC
    parameters:
        mac_type: string;
    submodules:
        mac: mac_type like General_MAC; // OK!
...\\
endmodule
```

The mac_type parameter should take the value "TokenRing_MAC", "Ethernet_MAC" or "FDDI_MAC", and a submodule of the appropriate type will be created. The value for the parameter can even be given in the ini file. This gives you a powerful tool to customize simulation models (see also *Topology templates*, Section 4.6.4).

5.2.4 The class declaration

As mentioned before, simple module classes have to be derived from cSimpleModule (either directly or indirectly). In addition to overwriting some of the previously mentioned four member functions (initialize(),activity(),handleMessage(),finish()), you have to write a constructor and some more functions. Some of this task can be automated, so when writing the C++ class declaration, you have two choices:

1. either use a macro which expands to the "stock" version of the functions
2. or write them yourself.

Using macro to declare the constructor

If you choose the first solution, you use the Module_Class_Members() macro:

```
Module_Class_Members( classname, baseclass, stacksize);
```

The first two arguments are obvious (*baseclass* is usually cSimpleModule), but *stacksize* needs some explanation. If you use activity(), the module code runs as a coroutine, so it will need a separate stack. (This will be discussed in detail later.)

As an example, the class declaration

```
class SlidingWindow : public cSimpleModule
{
    Module_Class_Members( SlidingWindow,cSimpleModule,8192)
    ...
};
```

expands to something like this:

```
class SlidingWindow : public cSimpleModule
{
  public:
    SlidingWindow(const char *name, cModule *parentmodule,
        unsigned stacksize = 8192) :
        cSimpleModule(name, parentmodule, stacksize) {}
    virtual const char *className() const {return "SlidingWindow";}
...
};
```

Expanded form of the constructor

You will implement:

- a constructor with the argument list: (const char *name, cModule *parentmodule, unsigned stacksize = *stacksize*)
- a className() function which returns the name of the class as char*

The advantage is that you get full control over the constructor, so you can initialize data members of the class (if you have any). You should not change the number or types of the arguments taken by the constructor, because it is called by OMNeT++-generated code. Also, remember to overwrite the className() function.

An example:

```
class TokenRing_MAC : public cSimpleModule
{
  public:
    cQueue queue; // a data member
    TokenRing_MAC(const char *name, cModule *parentmodule, unsigned stacksize = 8192);
    virtual const char *className() const {return "TokenRing_MAC";}
...
};

TokenRing_MAC(const char *name, cModule *parentmodule, unsigned stacksize) :
    cSimpleModule(name, parentmodule, stacksize), queue(''queue'') // initialize data me
{
}
```

Stack size decides between activity() and handleMessage()

- if the specified stack size is zero, handleMessage() will be used;
- if it is greater than zero, activity() will be used.

If you make mistake (e.g. you forget to set zero stack size for a handleMessage() simple module): the default versions of the functions issue error messages telling you what is the problem.

5.2.5 Decomposing activity()/handleMessage() and inheritance

It is usually a good idea to decompose a activity() or handleMessage() function when it grows too large. ”Too large” is a matter of taste of course, but you should definitely consider splitting up the function if it is more that a few screens (say 50-100 lines) long. This will have a couple of advantages:

- will help future readers of the code understand your program;
- will help *you* understand what it is you're really programming and bring some structure into it;
- will enable you to customize the class by inheriting from it and overwriting member functions

If you have variables which you want to access from all member functions (typically state variables are like that), you'll need to add those variables to the class as data members.

Let's see an example:

```
class TransportProtocol : public cSimpleModule
{
  public:
    Module_Class_Members(TransportProtocol, cSimpleModule, 8192)
    int window_size;
    int n_s; // N(s)
    int n_r; // N(r)
    cOutVector eedVector;
    cStdDev eedStats;
    //...

    virtual void activity();
    virtual void recalculateTimeout();
    virtual void insertPacketIntoBuffer(cMessage *packet);
    virtual void resendPacket(cMessage *packet);
    //...
};

Define_Module( TransportProtocol );

void TransportProtocol::activity()
{
    window_size = par("window_size");
    n_s = n_r = 0;
    eedVector.setName(''End-to-End Delay'');
    eedStats.setName(''eedStats'');
    //...
}

//...
```

Note that you may have to use the expanded form of the constructor (instead of Module_Class_Members()) to pass arguments to the constructors of member objects like eedVector and eedStats. But most often you don't need to go as far as that; for example, you can set parameters later from activity(), as shown in the example above.

To implement another variant of the Transport Protocol which uses a different timeout scheme, you could simply subclass TransportProtocol:

```
class AdvancedTransportProtocol : public TransportProtocol
{
  public:
    Module_Class_Members(AdvancedTransportProtocol, TransportProtocol, 8192)
    virtual void recalculateTimeout();
```

```
};

Define_Module( AdvancedTransportProtocol );

void AdvancedTransportProtocol::recalculateTimeout()
{
    //...
}
```

5.3 Adding functionality to cSimpleModule

This section discusses cSimpleModule's four previously mentioned member functions, intended to be redefined by the user: initialize(), activity(), handleMessage() and finish().

5.3.1 activity()

Process-style description

With activity(), you can code the simple module much like you would code an operating system process or a thread. You can wait for an incoming message (event) at any point of the code, you can suspend the execution for some time (model time!), etc. When the activity() function exits, the module is terminated. (The simulation can continue if there are other modules which can run.)

The most important functions you can use in activity() are (they will be discussed in detail later):

- receive..() family of functions – to receive messages (events)
- wait() – to suspend execution for some time (model time)
- 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()
- end() – to finish execution of this module (same as exiting the activity() function)

The activity() function normally contains an infinite loop, with at least a wait() or receive() call in its body.

Examples:

TBD

Application area

One area where the process-style description is especially convenient is when the process has many states but transitions are very limited, ie. from any state the process can only go to one or two other states. For example, this is the case when programming a network application which uses a single network connection. The pseudocode of the application which talks to a transport layer protocol might look like this:

```
activity()
{
    while(true)
    {
        open connection by sending OPEN command to transport layer
        receive reply from transport layer
        if (open not successful)
```

```
        {
            wait(some time)
            continue // loop back to while()
        }

        while(there's more to do)
        {
            send data on network connection
            if (connection broken)
            {
                continue outer loop // loop back to outer while()
            }
            wait(some time)
            receive data on network connection
            if (connection broken)
            {
                continue outer loop // loop back to outer while()
            }
            wait(some time)
        }
        close connection by sending CLOSE command to transport layer
        if (close not successful)
        {
            // handle error
        }
        wait(some time)
    }
}
```

If you want to handle several connections simultaneously, you may dynamically create as instances of the simple module above as needed. Dynamic module creation will be discussed later.

Activity() is run as a coroutine

Activity() is run in a coroutine. Coroutines are a sort of threads which are scheduled non-preemptively (this is also called cooperative multitasking). From one coroutine you can switch to another coroutine by a *transferTo(otherCoroutine)* call. Then this coroutine is suspended and *otherCoroutine* will run. Later, when *otherCoroutine* does a *transferTo(firstCoroutine)* call, execution of the first coroutine will resume from the point of the *transferTo(otherCoroutine)* call. The full state of the coroutine, including local variables are preserved while the thread of execution is in another coroutines. This implies that each coroutine must have an own processor stack, and *transferTo()* involves a switch from one processor stack to another.

Coroutines are at the heart of OMNeT++, and the simulation programmer doesn't ever need to call *transferTo()* or other functions in the coroutine library, nor does he need to care about the coroutine library implementation. But it is important to understand how the event loop found in discrete event simulators works with coroutines.

When using coroutines, the event loop looks like this (simplified):

```
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    transferTo(module containing the event)
}
```

That is, when the module has an event, the simulation kernel transfers the control to the module's coroutine.

It is expected that when the module ”decides it has finished the processing of the event”, it will transfer the control back to the simulation kernel by a *transferTo(main)* call. Initially, simple modules using activity() are ”booted” by events (*”starter messages”*) inserted into the FES by the simulation kernel before the start of the simulation.

How does the coroutine know it has ”finished processing the event”? The answer: *when it requests another event*. The functions which request events from the simulation kernel are the receive..() family and wait(), so their implementations contain a *transferTo(main)* call somewhere.

Their pseudocode, as implemented in OMNeT++:

```
receiveNew() // other receive...() variations are similar
{
    transferTo(main)
    retrieve current event
    return the event // remember: events = messages
}

wait()
{
    create an event e and schedule it at (current sim. time + wait interval)
    while(true) {
        transferTo(main)
        retrieve current event
        if (current event is e)
            break from loop
        else
            store current event for later use (in the ''put-aside queue'')
    }
    delete event e
    return
}
```

Thus, the receive...() and wait() calls are special points in the activity() function, because that’s where:

- simulation time elapses in the module, and
- other modules get a chance to execute.

Starter messages

Modules written with activity() need starter messages to ”boot”. These starter messages are inserted into the FES automatically by OMNeT++ at the beginning of the simulation, even before the initialize() functions are called.

Coroutine stack size

All the simulation programmer needs to care about coroutines is to choose the processor stack size for them. This cannot be automated (Eerrr... at least not without hardware support, some trick with virtual memory handling).

8 or 16 kbytes is usually a good choice, but you may need more if the module uses recursive functions or has local variables which occupy a lot of stack space. OMNeT++ has a built-mechanism that will usually detect if the module stack is too small and overflows. OMNeT++ can also tell you how much stack space a module actually uses, so you can find it out if you overestimated the stack needs.

initialize() and finish() with activity()

Because local variables of activity() are preserved across events, you can store everything (state information, packet buffers, etc.) in them. Local variables can be initialized at the top of the activity() function, so there isn't much need to use initialize().

However, you need finish() if you want to write statistics at the end of the simulation. And because finish() cannot access the local variables of activity(), you have to put the variables and objects that contain the statistics into the module class. You still don't need initialize() because class members can also be initialized at the top of activity().

Thus, a typical setup looks like this pseudocode:

```
class MySimpleModule...
{
    ...
    variables for statistics collection
    activity();
    finish();
};

MySimpleModule::activity()
{
    declare local vars and initialize them
    initialize statistics collection variables

    while(true)
    {
        ...
    }
}

MySimpleModule::finish()
{
    record statistics into file
}
```

Advantages and drawbacks

Advantages:

- initialize() not needed, state can be stored in local variables of activity()
- process-style description is a natural programming model in many cases

Drawbacks:

- memory overhead: stack allocation may unacceptably increase the memory requirements of the simulation program if you have several thousands or ten thousands of simple modules;
- run-time overhead: switching between coroutines is somewhat slower than a simple function call

Other simulators

Coroutines are used by a number of other simulation packages:

- All simulation software which inherit from SIMULA (e.g. C++SIM) are based on coroutines, although all in all the programming model is quite different.

- The simulation/parallel programming language Maisie and its successor PARSEC (from UCLA) also use coroutines (although implemented on with ”normal” preemptive threads). The philosophy is quite similar to OMNeT++. PARSEC, being ”just” a programming language, has a more elegant syntax but much less features than OMNeT++.
- Many Java-based simulation libraries are based on Java threads.

5.3.2 handleMessage()

Function called for each event

The idea is that at each event we simply call a user-defined function instead of switching to a coroutine that has activity() running in it. The ”user-defined function” is the handleMessage(cMessage *msg) virtual member function of cSimpleModule; the user has to redefine the function to make it do useful work. Calls to handleMessage() occur in the main stack of the program – no coroutine stack is needed and no context switch is done.

The handleMessage() function will be called for every message that arrives at the module. The function should process the message and return immediately after that. The simulation time is potentially different in each call. No simulation time elapses within a call to handleMessage().

The pseudocode of the event loop which is able to handle both activity() and handleMessage() simple modules:

```
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    m:= module containing this event
    if (m works with handleMessage())
        m->handleMessage( event )
    else // m works with activity()
    transferTo( m )
}
```

Modules with handleMessage() are NOT started automatically: the simulation kernel creates starter messages only for modules with activity(). This means that you have to schedule self-messages from the initialize() function if you want the handleMessage() simple module to start working ”by itself”, without first receiving a message from other modules.

Programming with handleMessage()

To use the handleMessage() mechanism in a simple module, you must specify *zero stack size* for the module. This is important, because this tells OMNeT++ that you want to use handleMessage() and not activity().

Message/event related functions you can use in handleMessage():

- 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()

You cannot use the receive...() family and wait() functions in handleMessage(), because they are coroutine-based by nature, as explained in the section about activity(). You also cannot use end() because its job is to terminate the coroutine.

You have to add data members to the module class for every piece of information you want to preserve. This information cannot be stored in local variables of handleMessage() because they are destroyed when the function returns. Also, they cannot be stored in static variables in the function (or the class), because they would be shared between all instances of the class.

Data members to be added to the module class will typically include things like:

- state (e.g. IDLE/BUSY, CONN_DOWN/CONN_ALIVE/...)
- other variables which belong to the state of the module: retry counts, packet queues, etc.
- values retrieved/computed once and then stored: values of module parameters, gate indices, routing information, etc.
- pointers of message objects created once and then reused for timers, timeouts, etc.
- variables/objects for statistics collection

You can initialize these variables from the initialize() function. The constructor is not a very good place for this purpose because it is called in the network setup phase when the model is still under construction, so a lot of information you may want to use is not yet available then.

Another task you have to do in initialize() is to schedule initial event(s) which trigger the first call(s) to handleMessage(). After the first call, handleMessage() must take care to schedule further events for itself so that the ”chain” is not broken. Scheduling events is not necessary if your module only has to react to messages coming from other modules.

finish() is used in the normal way: to record statistics information accumulated in data members of the class at the end of the simulation.

Application area

There are two areas where handleMessage() is definitely a better choice than activity():

1. For modules which have to maintain little or no state information, such as packet sinks.
2. Other good candidates are modules with a large state space and many arbitrary state transition possibilities (i.e. where there are many possible subsequent states for any state). Such algorithms are difficult to program with activity(), or the result is code which is better suited for handleMessage() (see rule of thumb below). Most communication protocols are like this.

There’s also a good rule of thumb. If your module, programmed with activity(), looks like this:

```
activity()
{
    initialization code
    while(true)
    {
        msg = receive();
        // arbitrary code which doesn’t contain any receive() or wait() calls
    }
}
```

Then it can be trivially converted to handleMessage():

```
initialize()
{
    initialization code
```

```
}

handleMessage( msg )
{
    // arbitrary code which doesn't contain any receive() or wait() calls
}
```

Example 1: Simple traffic generators and sinks

The code for simple packet generators and sinks programmed with handleMessage() might be as simple as this:

```
PacketGenerator::handleMessage(m)
{
    create and send out packet
    schedule m again to trigger next call to handleMessage // (self-message)
}
PacketSink::handleMessage(m)
{
    delete m
}
```

Note that *PacketGenerator* will need to redefine initialize() to create *m* and schedule the first event.

The following simple module generates packets with exponential inter-arrival time. (Some details in the source haven't been discussed yet, but the code is probably understandable nevertheless.)

```
class Generator : public cSimpleModule
{
    Module_Class_Members(Generator,cSimpleModule,0)
    // note zero stack size!
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module( Generator );

void Generator::initialize()
{
    // schedule first sending
    scheduleAt(simTime(), new cMessage);
}

void Generator::handleMessage(cMessage *msg)
{
    // generate & send packet
    cMessage *pkt = new cMessage;
    send(pkt, ''out'');
    // schedule next call
    scheduleAt(simTime()+exponential(1.0), msg);
}
```

Example 2: Bursty traffic generator

A bit more realistic example is to rewrite our Generator to create packet bursts, each consisting of burst_length packets.

We add some data members to the class:

- burst_length will store the parameter that specifies how many packets a burst must contain,
- burst_ctr will count in how many packets are left to be sent in the current burst.

The code:

```
class BurstyGenerator : public cSimpleModule
{
    Module_Class_Members(Generator,cSimpleModule,0)
    // note the zero stack size!
    int burst_length;
    int burst_ctr;
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module( BurstyGenerator );
void BurstyGenerator::initialize()
{
    // init parameters and state variables
    burst_length = par("burst_length");
    burst_ctr = burst_length;
    // schedule first packet of first burst
    scheduleAt(simTime(), new cMessage);
}

void BurstyGenerator::handleMessage(cMessage *msg)
{
    // generate & send packet
    cMessage *pkt = new cMessage;
    send(pkt, ''out'');
    // if this was the last packet of the burst
    if (--burst_ctr == 0)
    {
        // schedule next burst
        burst_ctr = burst_length;
        scheduleAt(simTime()+exponential(5.0), msg);
    }
    else
    {
        // schedule next sending within burst
        scheduleAt(simTime()+exponential(1.0), msg);
    }
}
```

Advantages and drawbacks

Advantages:

- consumes less memory: no separate stack needed for simple modules
- fast: function call is faster than switching between coroutines

Drawbacks:

- local variables cannot be used to store state information
- need to redefine initialize()
- programming model is inconvenient in some cases

Other simulators

Many simulation packages use a similar approach, often topped with something like a state machine (FSM) which hides the underlying function calls. Such systems are:

- OPNET$^{(TM)}$ (MIL3, Inc.) which uses FSM's designed using a graphical editor;
- NetSim++ clones OPNET's approach;
- SMURPH (University of Alberta) defines a (somewhat eclectic) language to describe FSMs, and uses a precompiler to turn it into C++ code;
- Ptolemy (UC Berkeley) uses a similar method.

OMNeT++'s FSM support is described in the next section.

5.3.3 initialize() and finish()

Purpose

initialize() – to provide place for any user setup code

finish() – to provide place where the user can record statistics after the simulation has completed

When and how they are called

The initialize() functions of the modules are invoked *before* the first event is processed, but *after* the initial events (starter messages) have been placed into the FES by the simulation kernel.

Both simple and compound modules have initialize() functions. A compound module has its initialize() function called *before* all its submodules have.

The finish() functions are called when the event loop has terminated, and only if it terminated normally (i.e. not with a runtime error). The calling order is the reverse as with initialize(): first submodules, then the containing compound module. (The bottom line is that in the moment there's no "official" possibility to redefine initialize() and finish() for compound modules; the unofficial way is to write into the nedc-generated C++ code. Future versions of OMNeT++ will support adding these functions to compound modules.)

This is summarized in the following pseudocode (although you won't find this code "as is" in the simulation kernel sources):

```
perform simulation run:
    build network (i.e. the system module and its submodules recursively)
    insert starter messages for all submodules using activity()
    do callInitialize() on system module
        enter event loop // (described earlier)
    if (event loop terminated normally) // i.e. not with a runtime
error
        do callFinish() on system module
    clean up
callInitialize()
{
    call to user-defined initialize() function
```

```
    if (module is compound)
        for (each submodule)
            do callInitialize() on submodule
}
callFinish()
{
    if (module is compound)
        for (each submodule)
            do callFinish() on submodule
    call to user-defined finish() function
}
```

initialize() vs. constructor

Usually you should not put simulation-related code into the simple module constructor. For example, modules often need to investigate their surroundings (maybe the whole network) at the beginning of the simulation and save the collected info into internal tables. Code like that cannot be placed into the constructor since the network is still being set up when the constructor is called.

finish() vs. destructor

Keep in mind that finish() is not always called, so it isn't a good place for cleanup code which should run every time the module is deleted. finish() is only a good place for writing statistics, result post-processing and other stuff which are to run only on successful completion.

Cleanup code should go into the destructor. But in fact, you almost never need to write a destructor because OMNeT++ keeps track of objects you create and disposes of them automatically (sort of automatic garbage collection). However it cannot track objects not derived from cObject (see later), so they may need to be deleted manually from the destructor.

Multi-stage initialization

In simulation models, when one-stage initialization provided by initialize() is not sufficient, one can use multi-stage initialization. Modules have two functions which can be redefined by the user:

```
void initialize(int stage);

int numInitStages();
```

At the beginning of the simulation, initialize(0) is called for all modules, then initialize(1), initialize(2), etc. For each module, numInitStages() must be redefined to return the number of init stages required, e.g. for a two-stage init, numInitStages() should return 2, and initialize(int stage) must be implemented to handle the stage=0 and stage=1 cases.

The callInitialize() function performs the full multi-stage initialization for that module and all its submodules.

If you do not redefine the multi-stage initialization functions, the default behavior is single-stage initialization: the default numInitStages() returns 1, and the default initialize(int stage) simply calls initialize().

"end-of-simulation event"

The task of finish() is solved in many simulators (e.g. OPNET) by introducing a special *end-of-simulation* event. This is not a very good practice because the simulation programmer has to code the algorithms (often FSMs) so that they can *always* properly respond to end-of-simulation events, in whichever state they are. This often makes program code unnecessarily complicated.

This fact is also evidenced in the design of the PARSEC simulation language (UCLA). Its predecessor Maisie used end-of-simulation events, but – as documented in the PARSEC manual – this is led to awkward programming in many cases, so for PARSEC, end-of-simulation events were dropped in favour of finish() (called *finalize()* in PARSEC).

5.4 Finite State Machines in OMNeT++

Overview

Finite State Machines (FSMs) can make life with handleMessage() easier. OMNeT++ provides a class and a set of macros to build FSMs. OMNeT++'s FSMs work very much like OPNET's or SDL's.

The key points are:

- There are two kinds of states: *transient* and *steady*. At each event (that is, at each call to handleMessage()), the FSM transitions out of the current (*steady*) state, undergoes a series of state changes (runs through a number of *transient* states), and finally arrives at another *steady* state. Thus between two events, the system is always in one of the steady states. Transient states are therefore not really a must – they exist only to group actions to be taken during a transition in a convenient way.
- You can assign program code to entering and leaving a state (known as entry/exit code). Staying in the same state is handled as leaving and re-entering the state.
- Entry code should not modify the state (this is verified by OMNeT++). State changes (transitions) must be put into the exit code.

OMNeT++'s FSMs *can* be nested. This means that any state (or rather, its entry or exit code) may contain a further full-fledged FSM_Switch (see below). This allows you to introduce sub-states and thereby bring some structure into the state space if it would become too large.

The FSM API

FSM state is stored in an object of type cFSM. The possible states are defined by an enum; the enum is also a place to tell which state is transient and which is steady. In the following example, SLEEP and ACTIVE are steady states and SEND is transient (the numbers in parens must be unique within the state type and they are used for constructing the numeric IDs for the states):

```
enum {
  INIT = 0,
  SLEEP = FSM_Steady(1),
  ACTIVE = FSM_Steady(2),
  SEND = FSM_Transient(1),
  };
```

The actual FSM is embedded in a switch-like statement, FSM_Switch(), where you have cases for entering and leaving each state:

```
FSM_Switch(fsm)
{
  case FSM_Exit(state1):
    //...
  break;
  case FSM_Enter(state1):
    //...
  break;
  case FSM_Exit(state2):
    //...
  break;
  case FSM_Enter(state2):
    //...
  break;
    //...
};
```

State transitions are done via calls to FSM_Goto(), which simply stores the new state in the cFSM object:

```
FSM_Goto(fsm,\textit{newState});
```

The FSM starts from the state with the numeric code 0; this state is conventionally named INIT.

Debugging FSMs

If you #define FSM_DEBUG before including omnetpp.h, each state transition will be logged to ev:

```
#define FSM_DEBUG
#include <omnetpp.h>
```

The actual printing is done through the FSM_Print() macro. You might redefine it if you don't like what it currently does:

```
#define FSM_Print(fsm,exiting)    (ev << "FSM " << (fsm).name()        << ((exiting) ? "
```

Implementation

The FSM_Switch() is a macro. It expands to a switch() statement embedded in a for() loop which repeats until the FSM reaches a steady state. (The actual code is rather ugly, but if you're dying to see it, it's in cfsm.h.)

Infinite loops are avoided by counting state transitions: if an FSM goes through 64 transitions without reaching a steady state, the simulation will terminate with an error message.

An example

Let us write another flavour of a bursty generator. It has two states, SLEEP and ACTIVE. In the SLEEP state, the module does nothing. In the ACTIVE state, it sends messages with a given inter-arrival time. The code was taken from the fifo2 sample simulation.

```
#define FSM_DEBUG
#include <omnetpp.h>

class BurstyGenerator : public cSimpleModule
{
  public:
  Module_Class_Members(BurstyGenerator,cSimpleModule,0)
  // parameters
  double sleepTimeMean;
  double burstTimeMean;
  double sendIATime;
  cPar *msgLength;
  // FSM and its states
  cFSM fsm;

  enum {
    INIT = 0,
    SLEEP = FSM_Steady(1),
    ACTIVE = FSM_Steady(2),
    SEND = FSM_Transient(1),
    };
  // variables used
  int i;
  cMessage *startStopBurst;
```

```
  cMessage *sendMessage;
  // the virtual functions
  virtual void initialize();
  virtual void handleMessage(cMessage *msg);
};

Define_Module( BurstyGenerator );

void BurstyGenerator::initialize()
{
  fsm.setName("fsm");
  sleepTimeMean = par("sleep_time_mean");
  burstTimeMean = par("burst_time_mean");
  sendIATime = par("send_ia_time");
  msgLength = &par("msg_length");
  i = 0;
  WATCH(i); // always put watches in initialize()
  startStopBurst = new cMessage("startStopBurst");
  sendMessage = new cMessage("sendMessage");
  scheduleAt(0.0,startStopBurst);
}

void BurstyGenerator::handleMessage(cMessage *msg)
{
  FSM_Switch(fsm)
 {
    case FSM_Exit(INIT):
    // transition to SLEEP state
      FSM_Goto(fsm,SLEEP);
    break;
    case FSM_Enter(SLEEP):
      // schedule end of sleep period (start of next burst)
      scheduleAt(simTime()+exponential(sleepTimeMean),
      startStopBurst);
    break;
    case FSM_Exit(SLEEP):
      // schedule end of this burst
      scheduleAt(simTime()+exponential(burstTimeMean),
      startStopBurst);
      // transition to ACTIVE state:
      if (msg!=startStopBurst)
        error("invalid event in state ACTIVE");
      FSM_Goto(fsm,ACTIVE);
      break;
    case FSM_Enter(ACTIVE):
    // schedule next sending
      scheduleAt(simTime()+exponential(sendIATime), sendMessage);
    break;
    case FSM_Exit(ACTIVE):
      // transition to either SEND or SLEEP
      if (msg==sendMessage) {
      FSM_Goto(fsm,SEND);
      } else if (msg==startStopBurst) {
      cancelEvent(sendMessage);
```

```
      FSM_Goto(fsm,SLEEP);
      } else
      error("invalid event in state ACTIVE");
    break;
    case FSM_Exit(SEND):
    {
      // generate and send out job
      char msgname[32];
      sprintf( msgname, "job-%d", ++i);
      ev << "Generating " << msgname << endl;
      cMessage *job = new cMessage(msgname);
      job->setLength( (long) *msgLength );
      job->setTimestamp();
      send( job, "out" );
      // return to ACTIVE
      FSM_Goto(fsm,ACTIVE);
      break;
    }
  }
}
```

5.5 Message transmission modeling

Data rate modeling

If data rate is specified for a connection, a message will have a certain nonzero transmission time, depending on its length. This means that when a message is sent out through an output gate, the message ”reserves” the gate for a given period (”it is being transmitted”).

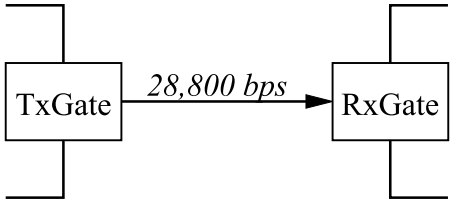

Figure 5.1: Connection with a data rate

While a message is under transmission, other messages have to wait until the transmission finishes. You can still use send() while the gate is busy, but the message’s arrival will be delayed; just like the gate had an internal queue for the messages waiting to be transmitted.

The OMNeT++ class library provides you with functions to check whether a certain output gate is transmitting or to learn when it finishes transmission.

If the connection with a data rate is not the immediate one connected to the simple module’s output gate but the second one in the route, you have to check the second gate’s busy condition.

Implementation of message sending

Message sending is implemented in the following way: the arrival time and the bit error flag of a message are calculated at once, when the send() (or similar) function is invoked. That is, if the message travels through several links until it reaches its destination, it is *not* scheduled individually for each link, but rather, every calculation is done once, within the send() call. This implementation was chosen because of its run-time efficiency.

In the actual implementation of queuing the messages at busy gates and modeling the transmission delay, messages do not actually queue up in gates; gates do not have internal queues. Instead, as the time when each gate will finish transmission is known at the time of sending the message, the arrival time of the message can be calculated in advance. Then the message will be stored in the event queue (FES) until the simulation time advances to its arrival time and it is retrieved by its destination module.

TBD add pseudocode

Consequence

The implementation has the following consequence. If you change the delay (or the bit error rate, or the data rate) of a link during simulation, the modeling of messages sent ”just before” the parameter change will not be accurate. Namely, if link parameters change while a message is ”under way” in the model, that message will not be affected by the parameter change, although it should. However, all subsequent messages will be modelled correctly. Similar for data rate: if a data rate changes during the simulation, the change will affect only the messages that are *sent* after the change.

If it is important to model gates and channels with changing properties, you can go two ways:

- write sender module such that they schedule events for when the gate finishes its current transmission and send then;
- alternatively, you can implement channels with simple modules (“active channels”).

The approach of some other simulators

Note that some simulators (e.g. OPNET) assign *packet queues* to input gates (ports), and messages send are buffered at the destination module (or the remote end of the link) until received by the destination module. With that approach, events and messages are separate entities, that is, a *send* operation includes placing the message in the packet queue *and* scheduling an event which will signal the arrival of the packet. In some implementations, also output gates have packet queues where packets wait until the channel becomes free (available for transmission).

OMNeT++ gates don’t have associated queues. The place where the sent but not yet received messages are buffered is the *FES*. OMNeT++’s approach is potentially faster than the above mentioned solution because it doesn’t have the enqueue/dequeue overhead and also spares an event creation. The drawback is, as mentioned above, that changes to channel parameters do not take effect immediately.

5.6 Coding conventions

Here’s a bunch of advice on how to write OMNeT++ models. Some of them are “rules of thumb”, saying if you program like that, you’re likely to have less trouble; other conventions are aimed at making the models produced by the OMNeT++ community more consistent.

Conventions for writing simple modules:

1. Put the NED description, the C++ class declaration and the implementation into three separate files. Do not put two or more modules’ code into the same file unless they are build upon one another - don’t be afraid of small files! Thus, for a simple module called Foobar, you should have Foobar.ned, Foobar.h and Foobar.cc. This reduces coupling of module sources and makes your code more reusable.

2. Adopt a good coding style. Some hints: Choose your favourite indentation style and keep to that consistently. I recommend four-space indents and the brace placement style in which the OMNeT++ sources are written. Write only one statement per line. Avoid putting comments at the end of the line - place them *above* the code on a separate line instead! Use blank lines to break the code into not-very-long logical blocks, and put a few-word comment above each block what that block does.

Leave at least two blank lines between two (member) functions. The purpose of all that is that the structure of your code be obvious at the first glance!

3. Identifiers: Begin module type names with a capital letter, and capitalize the beginning of each word, like in TokenRingMAC. Do not use underscore '_' in module names. Use the C++-style naming on member functions: beginning of each word is capitalized (except for the first one) and no underscores: sendUnnumberedFrame(). On parameter names, you may use C-style (window_size) or C++-style (windowSize) naming, whichever you prefer.

4. Make the functions virtual. Maybe someone who reuses your code will need a different behavior than what you thought of.

5. Use inheritance if you're writing a very complex simple module: create a basic simple module class and build upon it deriving new module classes. This will make your code more readable and easier to manage/reuse. Unfortunately, inheritance is not supported in NED so you actually have to make distinct NED descriptions for each simple module class. Even if you have an abstract classes, prepare a NED description for it: it is useful as a reference to others who might derive a different simple module class from your abstract class. Inheritance in NED is planned in later releases of OMNeT++.

6. Avoid global variables (and what's the same, static class members). They are not reset to their initial value (zero) when you run the simulation, stop it and rebuild the network. This can cause several problems when you use Cmdenv to execute several runs one after another, or in Tkenv when you rebuild the network from the menu.

7. Query the values of parameters into state variables (–>class members) of the *same* name at the top of the activity() function. If you know the value of a parameter is a random value (like uniform 0..10) or it can change during simulation, then to avoid having to look it up by name each time (like d=par("delay")) you may query its pointer into a cPar* state variable with the same name prepended with 'p' (like pDelay=&par("delay")).

8. Use ev.printf() and ev<<... (see later) to print out information on what the module is doing. Doing so will pay out several times when it comes to debugging. Use a parameter and a state variable called debug. Surround your debugging output (ev<<... and ev.printf() calls) with if(debug). You may introduce more specific debug switches (like debug_queuing etc.)

5.7 Component libraries

Because of the structure of the simulation system, one can create libraries of reusable elements in several ways. The three basic types are:

- simple module libraries
- NED source libraries
- precompiled compound module libraries

The elegant thing is that the user of the library does not need to know which kind of library he/she is using; the three types of libraries are equivalent in terms of usage.

5.7.1 Simple module libraries

Simple modules that can be used in more than simulations can form an object library. Good candidates for module libraries are simple modules that implement:

- Physical/Data-link protocols: Ethernet, Token Ring, FDDI, LAPB etc.
- Higher layer protocols: IP, TCP, X.25 L2/L3, etc.
- Network application types: E-mail, NFS, X, audio etc.
- Basic elements: message generator, sink, concentrator/simple hub, queue etc.
- Modules that implement routing algorithms in a multiprocessor or network
- ...

To create a library, you compile the simple module C++ sources and collect the object files in one directory. You'll also need to the provide the NED descriptions:

```
library/generator.ned
generator.o
sink.ned
sink.o
ethernet.ned
ethernet.o
```

The NED files contain the interfaces of the simple modules. For example:

```
// generator.ned
simple Generator
    parameters:
        interarrival time, message_length, message_kind;
    gates:
        out: output;
endsimple
```

The user of the library would include generator.ned in his/her NED files, and link the executable with generator.o. This is more or less the same concept as conventional C/C++ header files and libraries. The basic advantage is the same as with C/C++: you save compilation time and hide concrete implementation. The latter also means that you can give the module library to others without having to share the C++ source.

It could also be meaningful to provide the C++ header files with the module class declarations. This would enable the user to directly call the member functions of the module object from the simulation program and derive new module classes by redefining the virtual functions.

You do not need to have a separate NED file for each module: you could merge all of them into a single library.ned that contains the NED declarations of all modules without all side effects. However, it is not recommended to put all object files into one library (.a or .lib), because then every simple module would be present in simulation programs linked with the library, regardless whether the simulation uses them or not.

5.7.2 Compound module NED source libraries

The NED sources of reusable compound modules can also be placed in a library. Candidates are:

- network nodes such as hubs, bridges, routers
- different workstation/computer types: file server, X terminal etc.
- node of a massively parallel multiprocessor (used in testing different topologies)

- topology templates: parameterized ring, mesh, hypercube, torus etc. topologies, with the sizes (shapes etc) and the actual node types to be left as parameters
- ...

The NED sources are used through the import mechanism; the corresponding simple module object files still to be linked in the executable.

The user does not necessarily notice that he/she is using a compound module library and not a simple module library. In NED files, the user imports and uses the compound module sources in exactly the same way as he/she used the simple module interface declarations. Linking also goes in the same way; if the simple modules objects necessary for a certain compound module are aggregated into a library (.a or .lib), the user does not even notice the difference from the number of files he/she has to link in.

5.7.3 Precompiled compound module libraries

If you share a compound module with others, you do not necessarily have to share the NED source and reveal the internals of the compound module. You can turn the compound module into something that very much looks like a simple module.

Suppose you have the following compound module:

```
// router-compound.ned
module Router:
    parameters:
        processing_delay, buffersize;
    gates:
        in: input_ports[];
        out: output_ports[];
    submodules:
        routing: RoutingModule
            parameters:
                //...
            gatesizes:
                //...
        datalink: DataLink[num_of_ports]
            parameters:
                retry_count = 5,
                window_size = 2;
                //...
    connections:
    //...
endmodule
```

First, you would compile this NED file with the NEDC compiler and the resulting C++ code with the C++ compiler. Then you would aggregate this object file with the simple module object files into a single library (.a or .lib). Also, you would write a separate NED file that declares the interface of the new ”simple” module:

```
// router-simple.ned
simple Router:
    parameters:
        processing_delay, buffersize;
    gates:
```

```
        in: input_ports[];
        out: output_ports[];
endsimple
```

The method produced a *precompiled compound module*. The resulting two files can be placed into a simple module library and can be used identically to ordinary simple modules.

Using precompiled compound modules you can hide the internal complexity of your model from direct inspection. However, nothing can prevent a user from building a simulation executable with it and exploring the structure of your compound module using OMNeT++ simulation kernel functions. Consequently, using precompiled compound modules is more useful as a structuring tool.

5.8 Some simulation techniques

5.8.1 Modeling computer networks

The hierarchical module structure of OMNeT++ allows you to organize the model around different levels:

Physical topology:

1. Top-level network
2. Subnetwork (site)
3. LAN
4. node

Within a node:

1. OSI layers. The Data-Link, Network, Transport, Application layers are of greater importance.
2. Applications/protocols within a layer.

The advantage of OMNeT++ over many existing simulators is that the depth of the module nesting is not limited, and, what is in connection with the previous one, that a simple module can be transformed into a compound module by splitting the code into several simple modules *without affecting existing users* of the module and vica versa. The latter means that the programmer of the model is not under pressure from possibly incorrect early design decisions about what to implement with a single module and what with a compound module.

5.8.2 Modeling multiprocessor systems

One can make use of flexible model topologies. It is straightforward to create ring, mesh, butterfly, torus, hypercube, tree, fat tree and other topologies with conditional loop connections.

Furthermore, general *topology templates* (e.g. mesh or hypercube) can be created, where the types of the actual nodes are left as parameters. The actual node types are substituted as parameter values for each concrete simulation. Topology templates could be placed in a library and imported from there if needed.

5.8.3 Parameter tuning

Tuning means finding the parameter values which produce optimal operation of the system. In OMNeT++, you can tune the model during runtime. The code that monitors performance and changes parameter values can be placed:

- inside the model. In this case, the code does not necessarily form separate module(s); you can add the extra code to any already existing module.
- outside the model of the actual system. If you choose this method, you will create new modules that monitor and control the model.

OMNeT++ supports the model tuning concept by providing reference parameters. Parameters that influence the model performance and need to be tuned will be declared at the highest layer and taken by both the model and the monitor part.

An example of model tuning is how one can determine the critical throughput of a communication network by changing the offered load according to performance measures of the network (queuing times etc.)

5.8.4 Multiple experiments within one simulation run

One might need to perform a large number of simulation runs where the model parameters are not known in advance. This can be the case when one wants to optimize a system and parameter tuning cannot be used because

1. for each experiment, he wants to start the model from a well-defined initial state, or
2. he wants to change the model topology from one simulation run to the other

In this case, the following solution be followed. The network would consist of only one simple module that would organize the simulation runs by creating, running and destroying the actual models with each experiment. The simple module's code would look like this:

```
SimulationManager::activity()
{
    determine parameters for the first run
    while(true)
    {
        create the model (a compound module) with the current run parameters
        schedule
        wait( some time) // while the model runs
        delete future events that belong to the model
        get statistics out of the model
        destroy the model
        if (simulation is done)
            break
        calculate parameters for the next run
    }
    write out results
}
```

The solutions built into OMNeT++ (flexible module topologies, dynamic creation of compound modules etc.) strongly support this concept.

5.8.5 Dynamic topology optimization

Dynamic topology optimization is the generalization of the ”parameter tuning” and ”multiple experiments within one simulation run” concepts. If one wants to simulate large systems, it is possible that one part of the model needs its topology to be optimized (optimal number of servers, optimal interconnection etc.) while other parts of the model have reached their steady state and should not be bothered.

This can be achieved by modifying the previous scheme. Parts of the model that do not need topology optimization can be created once and left running for the whole duration of the simulation; other parts are examined and their structure is modified from time to time.

Chapter 6

The Simulation Library

OMNeT++ has a rich C++ class library which you can use when implementing simple modules. A quick overview of the areas supported by the simulation library:

- sending and receiving messages, scheduling and cancelling events, terminating the module or the simulation: member functions of cSimpleModule
- events, messages, network packets: the cMessage and cPacket classes
- random number generation: normal(), exponential(), etc.
- access to module gates and parameters: member functions of cModule (base class for cSimpleModule); cPar and cGate classes
- accessing other modules in the network: member functions of cModule and cGate
- storing data in containers: cArray, cQueue, cBag and cLinkedList classes
- discovering network topology and support routing: cTopology class
- recording statistics into file: cOutVector class
- collecting simple statistics: cStdDev and cWeightedStddev classes
- distribution estimation: cLongHistogram, cDoubleHistogram, cVarHistogram, cPSquare, cKSplit classes
- making variables inspectable in the graphical user interface (Tkenv): the WATCH macro (cWatch class)
- sending debug output to and prompting for user input in the graphical user interface (Tkenv): the ev object (cEnvir class)

6.1 Class library conventions

Base class

Classes in the OMNeT++ simulation library are derived from cObject. Functionality and conventions that come from cObject:

- name attribute
- className() member and other member functions giving textual information about the object

- conventions for assignment, copying, duplicating the object
- ownership control for containers derived from cObject
- support for traversing the object tree
- support for inspecting the object in graphical user interfaces (Tkenv)
- support for automatic cleanup (garbage collection) at the end of the simulation

Classes inherit and redefine several cObject member functions; in the following we'll discuss some of the practically important ones.

Setting and getting attributes

Member functions that set and query object attributes follow consistent naming. The setter member function has the form setSomething(...) and its getter counterpart is named something(), i.e. the "get" verb found in Java and some other libraries is omitted for brevity. For example, the *length* attribute of the cMessage class can be set and read like this:

```
msg->setLength( 1024 );
length = msg->length();
```

className()

For each class, the className() member function returns the class name as a string:

```
const char *classname = msg->className(); // returns "cMessage"
```

Name attribute

An object can be assigned a name (a character string). The name string is the first argument to the constructor of every class, and it defaults to NULL (no name string). If you supply a name string, the object will make its own copy (strdup). As an example, you can create a message object like this:

```
cMessage *mymsg = new cMessage("mymsg");
```

You can also set the name after the object has been created:

```
mymsg->setName("mymsg");
```

You can get a pointer to the internally stored copy of the name string like this:

```
const char *name = mymsg->name(); // --> returns ptr to internal copy
                                  // of "mymsg"
```

For convenience and efficiency reasons, the empty string "" and NULL are treated as equivalent by library objects: "" is stored as NULL (so that it does not consume heap), but it is returned as "" (so that it is easier to print out etc). Thus, if you create a message object with either NULL or "" as name, it will be stored as NULL and name() will return a pointer to "", a static string:

```
cMessage *msg = new cMessage(NULL, <additional args>);
const char *str = msg->name(); // --> returns ptr to  ""
```

fullName() and fullPath()

Objects have two more member functions which return other sort of names based on the name attribute: fullName() and fullPath().

Suppose we have a module in the network university_lan, compound module fddi_ring, simple module station[10]. If you call the functions on the simple module object (cSimpleModule inherits from cObject, too), the functions will return these values:

```
ev << module->name(); // --> "station"
ev << module->fullName(); // --> "station[10]"
ev << module->fullPath(); // --> ''university_lan.fddi_ring.station[10]''
```

These functions work for any object. For example, a local object inside the module would produce results like this:

void FDDIStation::activity()

```
{
    cQueue buffer("buffer");
    ev << buffer->fullPath(); // --> "university_lan.fddi_ring.
                              // station[10].buffer"
}
```

fullName() and fullPath(), together with className() can be used for example to generate informative error messages.

Be aware that fullName() and fullPath() return pointers to static buffers. Each call will overwrite the previous content of the buffer, so for example you shouldn't put two calls in a single printf() statement:

```
ev.printf("object1 is '%s', object2 is '%s'\n",
          object1->fullPath(),
          object2->fullPath()
         ); // WRONG! Same string is printed twice!!!
```

Copying and duplicating objects

The dup() member function creates an exact copy of the object, duplicating contained objects also if necessary. This is especially useful in the case of message objects. dup() returns a pointer of type cObject *, so it needs to be cast to the proper type:

```
cMessage *copyMsg = (cMessage *) msg->dup();
```

dup() works through calling the copy constructor, which in turn relies on the assignment operator between objects. operator=() can be used to copy contents of an object into another object of the same type. The copying is done properly; object contained in the object will also be duplicated if necessary. For various reasons, operator=() does not copy the name string; the copy constructor does it.

Iterators

There are several container classes in the library (cQueue, cArray etc.) For many of them, there is a corresponding iterator class that you can use to loop through the objects stored in the container.

For example:

```
cQueue queue;

//..
```

```
for (cQueueIterator queueIter(queue); !queueIter.end(); queueIter++)
{
    cObject *containedObject = queueIter();
}
```

Ownership control

By default, if a container object is destroyed, it destroys the contained objects too. If you call dup(), the contained objects are duplicated too for the new container. This is done so because contained objects are owned by the container; ownership is defined as the right/duty of deallocation. However, there is a fine-grain ownership control mechanism built in which allows you to specify on per-object basis whether you want objects to be owned by the container or not; by calling the takeOwnership() member function with false, you tell the container that you don't want it to become the owner of objects that will be inserted in the future. You can explicitly select an owner for any object by calling its setOwner() member function.

6.2 Utilities

Tracing

The tracing feature will be used extensively in the code examples, so it is shortly introduced here. It will be covered in detail in a later section.

The ev object represents the user interface of the simulation. You can send debugging output to ev with the C++-style output operators:

```
ev << "packet received, sequence number is "
   << seq_num << endl;
```

An alternative solution is ev.printf():

```
ev.printf("packet received, sequence number is %d\n", seq_num);
```

Simulation time conversion

There are utility functions which convert simulation time (simtime_t) to a printable string (like "3s 130ms 230us") and vica versa.

The simtimeToStr() function converts a simtime_t (passed in the first arg) to textual form. The result is placed into the buffer pointed to by the second arg. If the second arg is omitted or it is NULL, simtimeToStr() will place the result into a static buffer which is overwritten with each call:

```
char buf[32];
ev.printf("t1=%s, t2=%s\n", simtimeToStr(t1), simTimeToStr(t2,buf));
```

The strToSimtime() function parses a time specification passed in a string, and returns a simtime_t. If the string cannot be entirely interpreted, -1 is returned:

```
simtime_t t = strToSimtime("30s 152ms");
```

Another variant, strToSimtime0()can be used if the time string is a substring in a larger string. Instead of taking a char*, it takes a reference to char* (char*&) as the first argument. The function sets the pointer to the first character that could not be interpreted as part of the time string, and returns the value. It never returns -1; if nothing at the beginning of the string looked like simulation time, it returns 0.

```
const char *s = ''30s 152ms and some rubbish'';

simtime_t t = strToSimtime0(s); // now s points to "and some rubbish"
```

Utility <string.h> functions

The opp_strdup(), opp_strcpy(), opp_strcmp() functions are the same as their <string.h> equivalents, except that they treat NULL and the empty string ("") as identical, and opp_strdup() uses operator new instead of malloc().

The opp_concat() function might also be useful, for example in constructing object names. It takes up to four const char * pointers, concatenates them in a static buffer and returns pointer to the result. The result's length shouldn't exceed 255 characters.

6.3 Messages and packets

6.3.1 The cMessage class

In OMNeT++, cMessage is a central class. Objects of cMessage and subclasses may model a number of things: events; messages; packets, frames, cells, bits or signals travelling in a network; entities travelling in a system and so on.

Attributes

A cMessage object has number of attributes. Some are used by the simulation kernel, others are provided just for the convenience of the simulation programmer. A more-or-less complete list:

- The *name* attribute is inherited from cObject.
- The *message kind* attribute is supposed to carry some message type information. Zero and positive values can be freely used for any purpose. Negative values are reserved for use by the OMNeT++ simulation library; especially, MK_PACKET (-1) and MK_INFO (-2) are used to denote that the message is a network packet (see the cPacket class later).
- The *length* attribute (understood in bits) is used to compute transmission delay when the message travels through a connection that has an assigned data rate.
- The *bit error flag* attribute is set to true by the simulation kernel with a probability of $1-(1-ber)^{length}$ when the message is sent through a connection that has an assigned bit error rate (*ber*).
- The *priority* attribute is used by the simulation kernel to order messages in the message queue (FES) that have the same arrival time values.
- The *time stamp* attribute is not used by the simulation kernel; you can use it for purposes like remembering the time when the message was enqueued or re-sent.
- Other attributes and data members make simulation programming easier, they will be discussed later: *parameter list*, *encapsulated message*, *context pointer*.
- A number of read-only attributes store information about the message's (last) sending/scheduling: *source/destination module and gate*, *sending (scheduling) and arrival time*. They are mostly used by the simulation kernel while the message is in the FES, but the information is still in the message object when a module receives the message.

Basic usage

A cMessage object can be created in the following way:

```
cMessage *msg = new cMessage("msg-name", kind, length,
                             priority, errorflag);
```

The kind, length, and priority are integers, and errorflag is boolean. All arguments have default values, so the following initializations are also valid:

```
cMessage *msg1 = new cMessage;
cMessage *msg2 = new cMessage("data-packet", DATAPACKET_KIND, 8*1500 );
```

Once a message has been created, its data members can be changed by the following functions:

```
msg->setKind( kind );
msg->setLength( length );
msg->setPriority( priority );
msg->setBitError( err );
msg->setTimestamp();
msg->setTimestamp( simtime );
```

With these functions the user can set the message kind, the message length, the priority, the error flag and the time stamp. The setTimeStamp() function without any argument sets the time stamp to the current simulation time.

The values can be obtained by the following functions:

```
int k       = msg->kind();
int p       = msg->priority();
int l       = msg->length();
bool b      = msg->hasBitError();
simtime_t t = msg->timestamp();
```

Duplicating messages

It is often needed to duplicate a message (for example, send one and keep a copy). This can be done in the standard ways as for any other OMNeT++ object:

```
cMessage *copy1 = (cMessage *) msg->dup();
cMessage *copy2 = new cMessage( *msg );
```

The two are equivalent. The resulting message is an exact copy of the original, including message parameters (cPar or other object types) and encapsulated messages.

6.3.2 Attaching parameters and objects to a message

Adding, setting and reading parameters

You can add any number of parameters to a cMessage object. Parameters are objects of cPar type. You add a new parameter to the message with the addPar() member function:

```
msg->addPar("dest_addr");
```

You can get back the reference to the parameter object with the par() member function, and because cPar supports typecasting and assignment, it is easy to read and set the value of a parameter:

```
long dest_addr = msg->par("dest_addr");
msg->par("dest_addr") = 168;
```

The addPar() function also returns a reference to the added cPar object, so you can set the value of the new parameter at the same place:

```
msg->addPar("dest_addr") = 168;
```

You can use the hasPar() function to see if the message has a given parameter or not:

```
if (!msg->hasPar("dest_addr"))
  msg->addPar("dest_addr");
```

Numeric indices

Message parameters can be accessed also by index in the parameter array. The findPar() function returns the index of a parameter or -1 if the parameter cannot be found. The parameter can then be accessed using an overloaded par() function. Access by index is more efficient than access by name (although access by name might become faster in the future by using hashtables):

```
long dest_addr = 0;
int index = msg->findPar("dest_addr");

if (index>=0)
  dest_addr = msg->par(index);
```

Adding arbitrary data by accessing the internal array

Message parameters are stored in an object of type cArray which can store any object type not only cPars. The parList() member function lets you directly access the internal cArray, so by calling cArray's member functions you can attach any object to the message. An example:

```
cLongHistogram *pklen_distr = new cLongHistogram("pklen_distr");
msg->parList().add( pklen_distr );

...

cLongHistogram *pklen_distr =
    (cLongHistogram *) msg->parList().get("pklen_distr");
```

You should take care that names of the attached objects do not clash with parameter names.

If you do not add parameters to the message and do not call the parList() function, the internal cArray object will not be created. This saves you both storage and execution time.

You can attach non-object types (or non-cObject objects) to the message by using cPar's void* pointer 'P') type (see later in the description of cPar). An example:

```
struct conn_t *conn = new conn_t; // conn_t is a C struct
msg->addPar("conn") = (void *) conn;
msg->par("conn").configPointer(NULL,NULL,sizeof(struct conn_t));
```

Runtime overhead

It has been reported that using cPar message parameters might account for quite a large part of execution time (sometimes as much as 80%!). If your simulation is going to be very CPU-intensive, you're probably

better off subclassing either cMessage or rather cPacket, and adding the required parameters as ints, longs, bools, etc. to the new message class.

Some time in the future OMNeT++ will directly support message subclassing, and it will make the new parameters inspectable in the graphical user interface (Tkenv). This is a feature much demanded by users.

However, if you don't expect your simulations execute for hours, then cPar parameters are the most convenient way to go.

6.3.3 Message encapsulation

It is often necessary to encapsulate a message into another when you're modeling layered protocols of computer networks. Although you can encapsulate messages by adding them to the parameter list, there's a better way.

The encapsulate() function encapsulates a message into another one. The length of the message will grow by the length of the encapsulated message. An exception: when the encapsulating (outer) message has zero length, OMNeT++ assumes it is not a real packet but some out-of-band signal, so its length is left at zero.

```
cMessage *userdata = new cMessage("userdata");

userdata->setLength(8*2000);
cMessage *tcpseg = new cMessage("tcp");
tcpseg->setLength(8*24);
tcpseg->encapsulate(userdata);
ev << tcpseg->length() << endl; // --> 8*2024 = 16192
```

A message can only hold one encapsulated message at a time. The second encapsulate() call will result in an error. It is also an error if the message to be encapsulated isn't owned by the module.

You can get back the encapsulated message by decapsulate():

```
cMessage *userdata = tcpseg->decapsulate();
```

decapsulate() will decrease the length of the message accordingly, except if it was zero. If the length would become negative, an error occurs.

The encapsulatedMsg() function returns a pointer to the encapsulated message, or NULL if no message was encapsulated.

6.3.4 Information about the last sending

Readonly attributes

The following functions exist in cMessage:

```
bool isSelfMessage()
cGate *senderGate();           // return NULL if scheduled
cGate *arrivalGate();          // or unsent message

int senderModuleId();
int senderGateId();
int arrivalModuleId();
int arrivalGateId();
```

```
simtime_t creationTime();
simtime_t sendingTime();
simtime_t arrivalTime();
bool arrivedOn(int g);
bool arrivedOn(const char *s, int g=0);
```

TBD comments

Context pointer

cMessage contains a void* pointer which is set/returned by the setContextPointer() and contextPointer() functions:

```
void *context =...;
msg->setContextPointer( context );
void *context2 = msg->contextPointer();
```

It can be used for any purpose by the simulation programmer. It is not used by the simulation kernel, and it is treated as a mere pointer (no memory management is done on it).

Intended purpose: a module which schedules several self-messages (timers) will need to identify a self-message when it arrives back to the module, ie. the module will have to determine which timer went off and what to do then. The context pointer can be made to point at a data structure kept by the module which can carry enough ”context” information about the event.

6.3.5 The cPacket class

The cPacket class is derived from cMessage. It is intended as a base for all messages that model packets or frames in a telecommunications network.

cPacket adds two new data members to cMessage: *protocol* and *PDU* type (packet/frame/event type). Both are short integers, and are handled by the following member functions:

```
short protocol();
short pdu();
setProtocol(short p);
setPdu(short p);
```

Acceptable message kind values are:

- MK_PACKET
- MK_INFO

The cPacket constructor sets the message kind to MK_PACKET. Both MK_PACKET and MK_INFO are defined as negative integers. (Remember, negative message kind values are reserved for the simulation library.)

The protocol and PDU fields would ideally take a value from the protocol.h header in the simulation library. The contents of protocol.h is currently experimental; comments and contributions are welcome.

TDB examples for protocol and pdu values.

6.3.6 Subclassing cMessage and cPacket

TBD include an example

6.4 Sending and receiving messages

6.4.1 Sending messages

Once the message has been created, it can be sent through an output gate using one of these functions:

```
send(cMessage *msg, const char *gate_name, int index);
send(cMessage *msg, int gate);
```

For the first function, the argument gate_name is the name of the gate the message has to be sent through. If this gate is a vector gate, index determines though which particular output gate this has to be done; otherwise, the index argument is not needed.

The second function uses the gate number and because it does not have to search through the gate array, it is faster than the first one.

Examples:

```
send( new cMessage("token"), "out-gate");
send( new cMessage("token"), "vectorgate", i);

int out_gate_id = findGate("out-gate");
for (i=0; i<n; i++)
{
  send( new cMessage("packet"), out_gate_id);
  wait(in_time);
}
```

All message sending functions check that you actually own the message you are about to send. If the message is with another module, currently scheduled or in a queue etc., you'll get a runtime error. (The feature does not increase runtime overhead significantly, because it uses the object ownership management; it merely checks that the owner of the message is the module which wants to send it.)

6.4.2 Delayed sending

It is often needed to model a delay (processing time etc) immediately followed by message sending. In OMNeT++, it is possible to implement it like this:

```
wait( some_delay );
send( msg, "outgate" );
```

If the module needs to react to messages that arrive during the delay, wait() cannot be used and the timer mechanism described under *Self-messages* would need to be employed.

However, there is a more straightforward method than the above two, and this is delayed sending. Delayed sending can be done with one of these functions:

```
sendDelayed(cMessage *msg, double delay, const char *gate_name,
int index);
sendDelayed(cMessage *msg, double delay, int gate_id);
```

The arguments are the same as for send(), except for the extra *delay* parameter. The effect of the function is the same as if the module had kept the message for the delay interval and sent it afterwards. That is, the

sending time of the message will be the current simulation time (time at the sendDelayed() call) plus the delay. The delay value must be nonnegative.

Example:

```
sendDelayed( new cMessage("token"), 0.005, "out-gate");
```

6.4.3 Direct message sending

Sometimes it is necessary or convenient to ignore gates/connections and send a message directly to a remote destination module. The sendDirect() function does that, and it takes the pointer of the remote module (cModule *). You can also specify a delay and an input gate of the destination module.

```
cModule *destinationmodule =...;
double delay = truncnormal(0.005, 0.0001);
sendDirect( new cMessage, delay, destinationmodule, "in" );
```

The destination module receives the message as if it was sent ”normally”.

6.4.4 Receiving messages

With activity() only! The message receiving functions can only be used in the activity() function, handleMessage() gets the messages in its argument list.

A message can be received by a number of functions, the most general one is the receive() function:

```
cMessage *msg = receive();
```

Simple module objects contain a built-in queue object called putAsideQueue. The put-aside queue is used by some of the message-receiving functions.

There are two groups of functions that receive messages:

- `receive()`, `receiveOn()`
- `receiveNew()`, `receiveNewOn()`

The functions receive()/receiveOn()check the put-aside queue first and try to return a message from it. Only if they do not find an appropriate message in the put-aside queue, will wait for new messages.

The functions receiveNew()/receiveNewOn() wait for new messages, ignoring the put-aside queue.

Furthermore, the ...On() functions expect messages to arrive on a specific gate. Messages that arrive on another gate are inserted the put-aside queue. The On-less versions accept any message.

Since the receive() and receiveOn() return messages also from the put-aside queue, the arrival times of messages they return can be less than the current simulation time. A naive (and also incorrect) approach to check whether a message is a new one or it has been retrieved from the putaside-queue could be the following:

```
cMessage *msg = receive();

if (msg->arrivalTime()<simTime()) // not correct! several events may
                                  // occur at the same simulation time
{
  // handle msg as an old message
}
```

The correct way to do this is to check the putaside-queue:

```
bool queue_was_empty = putAsideQueue.empty();
cMessage *msg = receive();

if (!queue_was_empty)
{
  // handle msg as an old message
}
```

To discard the contents of the put-aside queue, one could use the following code:

```
while (!putAsideQueue.empty())
  delete receive();
```

To demonstrate receiveOn(), the following code fragment waits for a message on one specific input gate and discards all messages that arrived on other gates in the meanwhile:

```
cMessage *msg = receiveNewOn("important_input_gate");
while (!putAsideQueue.empty())
  delete receive();
```

The above code is almost equivalent to the following, except that it preserves the previous contents of the put-aside queue:

```
cMessage *msg;
for(;;)
{
  msg = receiveNew();
  if (msg->arrivedOn("important_input_gate"))
    break;
  delete msg;
}
```

All message receiving functions can be given a timeout value. (This is a *delta*, not an absolute simulation time.) If an appropriate message doesn't arrive within the timeout period, the function returns a NULL pointer. An example:

```
simtime_t timeout = 3.0;
cMessage *msg = receive( timeout );

if (msg=NULL)
  // timeout expired without any messages\\
else
  // process message
```

6.4.5 The wait() function

With activity() only! The wait() function's implementation contains a receive() call which cannot be used in handleMessage().

The wait() function suspends the execution of the module for a given amount of simulation time (a *delta*), regardless whether messages arrive at the module in the meanwhile or not:

```
wait( delay_interval );
```

In other simulation software, wait() is often called hold. Internally, the wait() function is implemented by a scheduleAt() followed by a receive(). The wait() function is very convenient in modules that do not need to be prepared for arriving messages, for example message generators. An example:

```
for(;;)
{
  wait( par("interarrival-time") );
  // generate and send message
}
```

The messages that arrived during the wait() call will accumulate in the putaside-queue. The putaside-queue can be examined directly (an example was shown in the previous section), and its contents is also retrieved by the receive() or receiveOn() functions.

6.4.6 Self-messages

The module can send a message to itself using the scheduleAt() function:

```
scheduleAt( time, msg );
```

scheduleAt() accepts an *absolute* simulation time (usually simTime()+*something*). Messages sent via scheduleAt() are called *self-messages*, and in OMNeT++ they are used to model events which occur within the module. Self-messages are delivered to the module in the same way as other messages (via the usual receive calls or handleMessage()); the module may call the isSelfMessage() member of any received message to determine if it is a self-message.

Before self-messages are delivered, they can be cancelled (removed from the FES). This is particularly useful because self-messages are often used to model timers.

```
cancelEvent( msg );
```

The cancelEvent() function takes a pointer to the message to be cancelled, and also returns the same pointer. After having it cancelled, you may delete the message or reuse it in the next scheduleAt() calls. cancelEvent()gives an error if the message is not in the FES.

The following example shows how to implement timers:

```
cMessage *timeout_msg = new cMessage;

scheduleAt( simTime()+10.0, timeout_msg );
//...

cMessage *msg = receive();
if (msg == timeout_msg)
{
  // timeout expired
}
else
{
  // other message has arrived, timer can be cancelled now:
  delete cancelEvent( timeout_msg );
}
```

You can determine if a message is currently in the FES by calling its isScheduled() member:

```
if (msg->isScheduled())
  delete cancelEvent(msg);
else
  ...
```

An advanced version of the above code which also checks the put-aside queue:

```
if (msg->isScheduled())
  delete cancelEvent(msg);
else if (putAsideQueue.contains(msg))
  delete putAsideQueue.remove(msg);
else
  ...
```

6.4.7 Querying the state of an output gate

You may have reasons to check whether a certain output gate is transmitting or to learn when it will finish transmission. This is done with gate object's isBusy() and transmissionFinishes() member functions. The latter function, transmissionFinishes() returns the time when the gate will finish its current transmission or (if it is currently free) when it finished its last transmission.

An example:

```
cMessage *packet = new cMessage("DATA");
packet->setLength( 1000 );

if (gate("TxGate")->isBusy()) // if gate is busy, wait until it
{                             // becomes free
  wait( gate("TxGate")->transmissionFinishes() - simTime());
}
send( packet, ''TxGate'');
```

If the connection with a data rate is not immediately the one connected to the simple module's output gate but the second one in the route, you have to check the second gate's busy condition. You would use the following code:

```
if (gate("mygate")->toGate()->isBusy())
  //...
```

Note that if data rates change during the simulation, the changes will affect only the messages that are *sent* after the change.

6.4.8 Stopping the simulation

Normal termination

You can finish the simulation with the endSimulation() function:

endSimulation();

However, typically you don't need endSimulation() because you can specify simulation time and CPU time limits in the ini file (see later).

Stopping on errors

If your simulation detects an error condition and wants to stop the simulation, you can do it with the error() member function of cModule. It is used like printf():

```
if (windowSize<1)
  error("Invalid window size %d; must be >=1", windowSize);
```

Do not include a newline (”\n”) or punctuation (period or exclamation mark) in the printed-out text, it will be added by OMNeT++.

6.5 Accessing module parameters and gates

6.5.1 Module parameters

Module parameters can be accessed with the par() member function of cModule:

```
cPar& delay_par = par("delay");
```

The cPar class is a general value-storing object. It supports type casts to numeric types, so parameter values can be read like this:

```
int num_tasks = par("num_tasks");
double proc_delay = par("proc_delay");
```

If the parameter is a random variable or its value can change during execution, it is best to store a reference to it and re-read the value each time it is needed:

```
cPar& wait_time = par("wait_time");
for(;;)
{
  //...
  wait( (simtime_t)wait_time );
}
```

If the wait_time parameter was given a random value (e.g. exponential(1.0)) in the NED source or the ini file, the above code results in a different delay each time.

Parameter values can also be changed from the program, during execution. If the parameter was taken by reference (with a ref modifier in the NED file), other modules will also see the change. Thus, parameters taken by reference can be used as a means of module communication.

An example:

```
par("wait_time") = 0.12;
```

Or:

```
cPar& wait_time = par("wait_time");
wait_time = 0.12;
```

See cPar explanation later in this manual for further information on how to change a cPar’s value.

6.5.2 Gates and links

Gate objects

Module gates are cGate objects. Gate objects know whether and to which gate they are connected, and they can be asked about the parameters of the link (delay, data rate, etc.)

The gate()member function of cModule returns a pointer to a cGate object, and an overloaded form of the function lets you to access elements of a vector gate:

```
cGate *outgate = gate("out");
cGate *outvec5gate = gate("outvec",5);
```

For gate vectors, the first form returns the first gate in the vector (at index 0).

The isVector() member function can be used to determine if a gate belongs to a gate vector or not. But this is almost insignificant, because non-vector gates are treated as vectors with size 1.

Given a gate pointer, you can use the size() and index() member functions of cGate to determine the size of the gate vector and the index of the gate within the vector:

```
int size2 = outvec5gate->size(); // --> size of outvec[]
int index = outvec5gate->index(); // --> 5 (it is gate 5 in the vector)
```

For non-vector gates, size() returns 1 and index() returns 0.

The type() member function returns a character, 'I' for input gates and 'O' for output gates:

```
char type = outgate->type() // --> 'O'
```

Gate IDs

Module gates (input and output, single and vector) are stored in an array within their modules. The gate's position in the array is called the *gate ID*. The gate ID is returned by the id() member function:

```
int id = outgate->id();
```

For a module with input gates from_app and in[3] and output gates of to_app and status, the array may look like this:

ID	dir	name[index]
0	*input*	`from_app`
1	*output*	`to_app`
2	*empty*	
3	*input*	`in[0]`
4	*input*	`in[1]`
5	*input*	`in[2]`
6	*output*	`status`

The array may have empty slots. Gate vectors are guaranteed to occupy contiguous IDs, that is, it is legal to calculate the ID of *gate[k]* as `gate("gate",0).id()+k`.

Message sending and receiving functions accept both gate names and gate IDs; the functions using gates IDs are a bit faster. Gate IDs do not change during execution, so it is often worth retrieving them in advance and using them instead of gate names.

Gate IDs can also be determined with the findGate() member of cModule:

```
int id1 = findGate("out");
int id2 = findGate("outvect",5);
```

Link parameters

The following member functions return the link attributes:

```
cLinkType *link = outgate->link();
cPar *d = outgate->delay();
cPar *e = outgate->error();
cPar *r = outgate->datarate();
```

Transmission state

The isBusy() member function returns whether the gate is currently transmitting, and if so, the transmissionFinishes() member function returns the simulation time when the gate is going to finish transmitting.

Connectivity

TBD figure

The isConnected() member function returns whether the gate is connected. If the gate is an output gate, the gate to which it is connected is obtained by the toGate() member function. For input gates, the function is fromGate().

```
cGate *gate = gate("somegate");
if (gate->isConnected())
{
  cGate *othergate = (gate->type()=='O') ?
                     gate->toGate() : gate->fromGate();

  ev << "gate is connected to: " << othergate->fullPath() << endl;
}
else
{
  ev << "gate not connected" << endl;
}
```

An alternative to isConnected() is to check the return value of toGate() or fromGate(). The following code is fully equivalent to the one above:

```
cGate *gate = gate("somegate");
cGate *othergate = (gate->type()=='O') ?
                   gate->toGate() : gate->fromGate();
if (othergate)
  ev << "gate is connected to: " << othergate->fullPath() << endl;
else
  ev << "gate not connected" << endl;
```

To find out to which simple module a given output gate leads finally, you would have to walk along the path like this (the ownerModule() member function returns the module to which the gate belongs):

```
cGate *gate = gate("out");
while (gate->toGate()!=NULL)
{
  gate = gate->toGate();
```

```
}

cModule *destmod = gate->ownerModule();
```

but luckily, there are two convenience functions which do that: sourceGate() and destinationGate().

6.6 Walking the module hierarchy

Module vectors

If a module is part of a module vector, the index() and size() member functions can be used to query its index and the vector size:

```
ev << "This is module [" << module->index() <<
      "] in a vector of size [" << module->size() << "].\n";
\end{Verbatom}


\textbf{Module IDs}


Each module in the network has a unique ID that is returned by
the id() member function. The module ID is used internally by
the simulation kernel to identify modules.

\begin{Verbatim}
int myModuleId = id();
```

If you know the module ID, you can ask the simulation object (a global variable) to get back the module pointer:

```
int id = 100;
cModule *mod = simulation.module( id );
```

Module IDs are guaranteed to be unique, even when modules are created and destroyed dynamically. That is, an ID which once belonged to a module which was deleted is never issued to another module later.

Walking up and down the module hierarchy

The surrounding compound module can be accessed by the parentModule() member function:

```
cModule *parent = parentModule();
```

For example, the parameters of the parent module are accessed like this:

```
double timeout = parentModule()->par( "timeout" );
```

cModule's findSubmodule() and submodule() member functions make it possible to look up the module's submodules by name (or name+index if the submodule is in a module vector). The first one returns the numeric module ID of the submodule, and the latter returns the module pointer. If the submodule is not found, they return -1 or NULL, respectively.

```
int submodID = compoundmod->findSubmodule("child",5);
cModule *submod = compoundmod->submodule("child",5);
```

The moduleByRelativePath() member function can be used to find a submodule nested deeper than one level below. For example,

```
compoundmod->moduleByRelativePath("child[5].grandchild");
```

would give the same results as

```
compoundmod->submodule("child",5)->submodule("grandchild");
```

(Provided that child[5] does exist, because otherwise the second version will crash with an access violation because of the NULL pointer.)

The cSimulation::moduleByPath() function is similar to cModule's moduleByRelativePath() function, and it starts the search at the top-level module.

Iterating over submodules

To access all modules within a compound module, use cSubModIterator. For example:

```
for (cSubModIterator submod(*parentModule()); !submod.end(); submod++)
{
  ev << submod()->fullName();
}
```

(submod() is pointer to the current module the iterator is at.)

The above method can also be used to iterate along a module vector, since the name() function returns the same for all modules:

```
for (cSubModIterator submod(*parentModule()); !submod.end(); submod++)
{
  if (submod()->isName( name() )) // if submod() is in the same
                                  // vector as this module
  {
    int its_index = submod()->index();
    // do something to it
  }
}
```

Walking along links

To determine the module at the other end of a connection, use cGate's fromGate(), toGate() and ownerModule() functions. For example:

```
cModule *neighbour = gate( "outputgate" )->toGate()->ownerModule();
```

For input gates, you would use fromGate() instead of toGate().

6.7 Dynamic module creation

Why

If you do not know how many modules you'll need, you can create modules dynamically and dispose of them when they are no longer needed. Both simple and compound modules can be created this way. If you create a compound module dynamically, all its submodules will be recursively built.

Let's suppose you are implementing a transport protocol for a computer network model. It is convenient to have a separate module to handle each connection. However, there's no way to know how many connections there'll be simultaneously. The solution is to create a manager module which receives connection requests and creates a module for each connection. The Dyna example simulation does something like this.

It is often convenient to use direct message sending with dynamically created modules.

Overview

To understand how dynamic module creation works, you have to know a bit about how normally OMNeT++ instantiates modules. Each module type (class) has a corresponding description object of the class cModuleType. This object is created under the hood by the Define_Module() macro, and it has a factory function which can instantiate the module class (this function basically only consists of a return new *module-class*(...) statement).

The cModuleType object can be looked up by its name string (which is the same as the module class name). Once you have its pointer, it's possible to call its factory method and create an instance of the corresponding module class – without having to include the module's class declaration into your C++ file.

The cModuleType object also knows what gates and parameters the given module type has to have. (This info comes from compiled NED code.)

Simple modules can be created in one step. For a compound module, the situation is more complicated, because its internal structure (submodules, connections) may depend on parameter values and gate vector sizes. Thus, for compound modules it is generally required to set parameter values and gate vector sizes after creation of the module itself, but before creating its submodules and internal connections.

As you know already, simple modules with activity() need a starter message. For statically created modules, this message is created automatically by OMNeT++, but for dynamically created modules, you have to do this explicitly by calling the appropriate functions.

Calling initialize() has to take place after insertion of the stater messages, because the initializing code may insert new messages into the FES, and these messages should be processed *after* the starter message.

TBD

```
cModuleType *moduleType = findModuleType("TCPConnectionHandler");
```

Simple form

Mainly for creating simple modules.

TBD

cModuleType has createScheduleInit(const char *name, cModule *parentmod) convenience function to get a module up and running in one step.

```
mod = modtype->createScheduleInit("name",this);
```

Does `create()+buildInside()+callInitialize()+scheduleStart(now)`.

Should work for both simple and compound modules.

Not applicable if the module:

- has parameters to be set
- has gate vector sizes to be set
- has gates to be connected before initialize()

Example:

TBD

Expanded form

If the previous simple form cannot be used. There are 5 steps:

1. find descriptor object
2. create module
3. set up parameters and gate sizes (if needed)
4. call function that builds out submodules and finalizes the module
5. call function that creates activation message(s) for the new simple module(s)

Each step (except for Step 3.) can be done with one line of code.

See the following example where Step 3. is omitted:

```
// find descriptor object
cModuleType *moduleType = findModuleType("TCPConnectionHandler");
// create (possibly compound) module and build its submodules (if any)
cModule *module = moduleType->create( "TCPconn", this );
moduleType->buildInside( module );
// create activation message
module->scheduleStart( simTime() );
```

If you want to set up parameter values or gate vector sizes (Step 3.), the code goes between the create() and buildInside() calls:

```
cModuleType *moduleType = findModuleType("TCP-conn-handler");
cModule *module = moduleType->create( "TCPconn", this );
// set up parameters and gate sizes before we set up its submodules
module->par("window-size") = 4096;
module->setGateSize("to-apps", 3);
moduleType->buildInside( module );
module->scheduleStart( simTime() );
```

Deleting

To delete a module dynamically:

```
module->deleteModule();
```

If the module was a compound module, this involves recursively destroying all its submodules. A simple module can also delete itself; in this case, if the module was implemented using activity(), the deleteModule() call does not return to the caller (the reason is that deleting the module also deletes the CPU stack of the coroutine).

Currently, you cannot safely delete a compound module from a simple module in it; you must delegate the job to a module outside the compound module.

Creating connections

There are two functions that you can use to connect gates. For a normal user, they are useful for creating connections to dynamically created modules.

```
connect( cModule *src_module, int src_gatenumber,
         cLinkType *channeltype,
```

```
        cModule *dest_module, int dest_gatenumber );

connect( cModule *src_module, int src_gatenumber,
        cPar *delay, cPar *error, cPar *datarate,
        cModule *dest_module, int dest_gatenumber );
```

Any of the channeltype, delay, error and datarate pointers can be NULL.

An example:

```
connect( this, findGate("out"),
        (cLinkType *)NULL,
        module, module->findGate("in",0) );
```

6.8 Routing support: cTopology

6.8.1 Overview

The cTopology class was designed primarily to support routing in telecommunication or multiprocessor networks.
A cTopology object stores an abstract representation of the network in graph form:

- each cTopology node corresponds to a *module* (simple or compound), and
- each cTopology edge corresponds to a *link* or *series of connecting links*.

You can specify which modules (either simple or compound) you want to include in the graph. The graph will include all connections among the selected modules. In the graph, all nodes are at the same level, there's no submodule nesting. Connections which span across compound module boundaries are also represented as one graph edge. Graph edges are directed, just as module gates are.

If you're writing a router or switch model, the cTopology graph can help you determine what nodes are available through which gate and also to find optimal routes. The cTopology object can calculate shortest paths between nodes for you.

The mapping between the graph (nodes, edges) and network model (modules, gates, connections) is preserved: you can easily find the corresponding module for a cTopology node and vica versa.

6.8.2 Basic usage

You can extract the network topology into a cTopology object by a single function call. You have several ways to select which modules you want to include in the topology:

- by module type
- by a parameter's presence and its value
- with a user-supplied boolean function

First, you can specify which node types you want to include. The following code extracts all modules of type Router or User. (Router and User can be both simple and compound module types.)

```
cTopology topo;
topo.extractByModuleType( "Router", "User", NULL );
```

Any number of module types (up to 32) can be supplied; the list must be terminated by NULL.

Second, you can extract all modules which have a certain parameter:

```
topo.extractByParameter( "ip_address" );
```

You can also specify that the parameter must have a certain value for the module to be included in the graph:

```
cPar yes = "yes";
topo.extractByParameter( "include_in_topo", &yes );
```

The third form allows you to pass a function which can determine for each module whether it should or should not be included. You can have cTopology pass supplemental data to the function through a void* pointer. An example which selects all top-level modules (and does not use the void* pointer):

```
int select_function(cModule *mod, void *)
{
  return mod->parentModule() == simulation.systemModule();
}

topo.extractFromNetwork( select_function, NULL );
```

TBD one more example which *does use* the void* ptr.

A cTopology object uses two types: sTopoNode for nodes and sTopoLink for edges. (sTopoLinkIn and sTopoLinkOut are 'aliases' for sTopoLink; we'll speak about them later.)

Once you have the topology extracted, you can start exploring it. Consider the following code (we'll explain it shortly):

```
for (int i=0; i<topo.nodes(); i++)
{
  sTopoNode *node = topo.node(i);
  ev << "Node i=" << i << " is " << node->module()->fullPath() << endl;
  ev << " It has " << node->outLinks() << " conns to other nodes\n";
  ev << " and " << node->inLinks() << " conns from other nodes\n";

  ev << " Connections to other modules are:\n";
  for (int j=0; j<node->outLinks(); j++)
  {
    sTopoNode *neighbour = node->out(j)->remoteNode();
    cGate *gate = node->out(j)->localGate();
    ev << " " << neighbour->module()->fullPath()
       << " through gate " << gate->fullName() << endl;
  }
}
```

The nodes() member function (1st line) returns the number of nodes in the graph, and node(i) returns a pointer to the *i*th node, an sTopoNode structure.

The correspondence between a graph node and a module can be obtained by:

```
sTopoNode *node = topo.nodeFor( module );
cModule *module = node->module();
```

The nodeFor() member function returns a pointer to the graph node for a given module. (If the module is not in the graph, it returns NULL). nodeFor() uses binary search within the cTopology object so it is fast enough.

sTopoNode's other member functions let you determine the connections of this node: inLinks(), outLinks() return the number of connections, in(i) and out(i) return pointers to graph edge objects.

By calling member functions of the graph edge object, you can determine the modules and gates involved. The remoteNode() function returns the other end of the connection, and localGate(), remoteGate(), localGateId() and remoteGateId() return the gate pointers and ids of the gates involved. (Actually, the implementation is a bit tricky here: the same graph edge object sTopoLink is returned either as sTopoLinkIn or as sTopoLinkOut so that "remote" and "local" can be correctly interpreted for edges of both directions.)

6.8.3 Shortest paths

The real power of cTopology is in finding shortest paths in the network to support optimal routing. cTopology finds shortest paths from *all* nodes *to* a target node. The algorithm is computationally inexpensive. In the simplest case, all edges are assumed to have the same weight.

A real-life example when we have the target module pointer, finding the shortest path looks like this:

```
cModule *targetmodulep =...;
sTopoNode *targetnode = topo.nodeFor( targetmodulep );
topo.unweightedSingleShortestPathsTo( targetnode );
```

This performs the Dijkstra algorithm and stores the result in the cTopology object. The result can then be extracted using cTopology and sTopoNode methods. Naturally, each call to unweightedSingleShortestPathsTo() overwrites the results of the previous call.

Walking along the path from our module to the target node:

```
sTopoNode *node = topo.nodeFor( this );

if (node == NULL)
{
  ev < "We (" << fullPath() << ") are not included in the topology.\n";
}
else if (node->paths()==0)
{
  ev << "No path to destination.\n";
}
else
{
  while (node != topo.targetNode())
  {
    ev << "We are in " << node->module()->fullPath() << endl;
    ev << node->distanceToTarget() << " hops to go\n";
    ev << "There are " << node->paths()
       << " equally good directions, taking the first one\n";
    sTopoLinkOut *path = node-\texttt{>}path(0);
    ev << "Taking gate " << path->localGate()->fullName()
       << " we arrive in " << path->remoteNode()->module()->fullPath()
       << " on its gate " << path->remoteGate()->fullName() << endl;
    node = path->remoteNode();
  }
}
```

The purpose of the distanceToTarget() member function of a node is self-explanatory. In the unweighted case, it returns the number of hops. The paths() member function returns the number of edges which are part of a shortest path, and path(i) returns the *i*th edge of them as sTopoLinkOut. If the shortest paths were created by the ...SingleShortestPaths() function, paths() will always return 1 (or 0 if the target is not reachable), that is, only one of the several possible shortest paths are found. The ...MultiShortestPathsTo() functions find all paths, at increased run-time cost. The cTopology's targetNode() function returns the target node of the last shortest path search.

You can enable/disable nodes or edges in the graph. This is done by calling their enable() or disable() member functions. Disabled nodes or edges are ignored by the shortest paths calculation algorithm. The enabled() member function returns the state of a node or edge in the topology graph.

One usage of disable() is when you want to determine in how many hops the target node can be reached from our node *through a particular output gate*. To calculate this, you calculate the shortest paths to the target *from the neighbor node*, but you must disable the current node to prevent the shortest paths from going through it:

```
sTopoNode *thisnode = topo.nodeFor( this );
thisnode->disable();
topo.unweightedSingleShortestPathsTo( targetnode );
thisnode->enable();

for (int j=0; j<thisnode->outLinks(); j++)
{
  sTopoLinkOut *link = thisnode->out(i);
  ev << "Through gate " << link->localGate()->fullName() << " : "
     << 1 + link->remoteNode()->distanceToTarget() << " hops" << endl;
}
```

In the future, other shortest path algorithms will also be implemented:

```
unweightedMultiShortestPathsTo(sTopoNode *target);
weightedSingleShortestPathsTo(sTopoNode *target);
weightedMultiShortestPathsTo(sTopoNode *target);\\
```

6.9 Generating random numbers

Having high quality random numbers is usually very important in simulation programs. The random number generator used in OMNeT++ is a linear congruential generator (LCG) with a cycle length of 2^{31}-2. The startup code of OMNeT++ contains code that checks if the random number generator works OK, so you do not have to worry about this if you port the simulator to a new architecture or use a different compiler.

If a simulation program uses random numbers for more than one purpose, the numbers should come from different random number generators. OMNeT++ provides several independent random number generators (by default 32; this number is #defined as NUM_RANDOM_GENERATORS in utils.h).

To avoid unwanted correlation, it is also important that different simulation runs and different random number sources within one simulation run use non-overlapping series of random numbers, so the generators should be started with seeds well apart. For selecting good seeds, the seedtool program can be used (it is documented later).

The random number generator was taken from [JAIN91, pp. 441-444,455]. It has the following properties:

- Range: $1...2^{31} - 2$
- Period length: $2^{31} - 2$

- Method: $x := (x * 7^5) mod (2^{31} - 1)$
- To check: if $x[0] = 1$ then $x[10000] = 1,043,618,065$
- Required hardware: exactly 32-bit integer arithmetics

The concrete implementation:

```
long intrand()
{
  const long int a=16807, q=127773, r=2836;
  seed=a*(seed%q) - r*(seed/q);
  if (seed<=0) seed+=INTRAND_MAX;
  return seed;
}
```

6.9.1 Using random number generators directly

The generator is directly accessible through the intrand() function:

```
long rnd = intrand();   // in the range 1..INTRAND\_MAX-1
```

The random number seed can be specified in the ini file (random-seed=) or set directly from within simple modules with the randseed() function:

```
randseed( 10 );         // set seed to 10
long seed = randseed(); // current seed value
```

Zero is not allowed as a seed.

The intrand() and randseed() functions use generator 0. They have another variant which uses a specified generator:

```
long rnd = genk_intrand(6); // like intrand(), using generator 6
genk_randseed( k, 167 );    // set seed of generator k to 167
```

The intrand(n) and dblrand() functions are based on intrand():

```
int dice = 1 + intrand(6); // result of intrand(6) is in the range 0..5
                           // (it is calculated as intrand()%6)

double prob = dblrand();   // dblrand() produces numbers in [0,1)
                           // calculated as intrand()/(double)INTRAND_MAX
```

They also have their counterparts that use generator *k*:

```
int dice = 1 + genk_intrand(k,6); // uses generator k
double prob = genk_dblrand(k);    // ""
```

6.9.2 Random numbers from distributions

The following functions are based on dblrand() and return random variables of different distributions:

```
double uniform(double lower_limit, double upper_limit);
double intuniform(double lower_limit, double upper_limit);
double exponential(double mean);
double normal(double mean, double deviation);
double truncnormal(double mean, double deviation);
```

They are the same functions that can be used in NED files. intuniform() generates integers including both the lower and upper limit, so for example the outcome of tossing a coin could be written as intuniform(1,2). truncnormal() is the normal distribution truncated to nonnegative values; its implementation generates a number with normal distribution and if the result is negative, it keeps generating other numbers until the outcome is nonnegative.

The counterparts of the above functions using generator *k*:

```
double genk_uniform(double k, double lower_limit, double upper_limit);
double genk_intuniform(double k, double lower_limit, double upper_limit);
double genk_exponential(double k, double mean);
double genk_normal(double k, double mean, double deviation);
double genk_truncnormal(double k, double mean, double deviation);
```

Note that they take the number of the generator as a double; it is so because these functions are designed so that they can be used with the cPar class and in NED files. You will find more information about this in the section describing cPar.

If the above distributions do not suffice, you can write your own functions. If you register your functions with the Register_Function() macro, you can use them in NED files and ini files too. You can find the implementation of many distributions in the class library of GNU C++.

6.9.3 Random numbers from histograms

You can also specify your distribution as a histogram. The `cLongHistogram`, `cDoubleHistogram`, `cVarHistogram`, `cKSplit` or `cPSquare` classes are there to generate random numbers from equidistant-cell or equiprobable-cell histograms. This feature is documented later, with the statistical classes.

6.10 Container classes

6.10.1 Queue class: cQueue

Basic usage

cQueue is a container class that acts as a queue. cQueue can hold objects of type derived from cObject (almost all classes from the OMNeT++ library), such as cMessage, cPar, etc. Internally, cQueue uses a double-linked list to store the elements.

As an example of use, the simple modules' put-aside queues (putAsideQueue member) are cQueues which store cMessage objects. (However, the Future Event Set [FES] is not a cQueue; it is implemented with heap [class cMessageHeap] because it is a lot more efficient.)

A queue object has a head and a tail. Normally, new elements are inserted at its head and elements are removed at its tail.

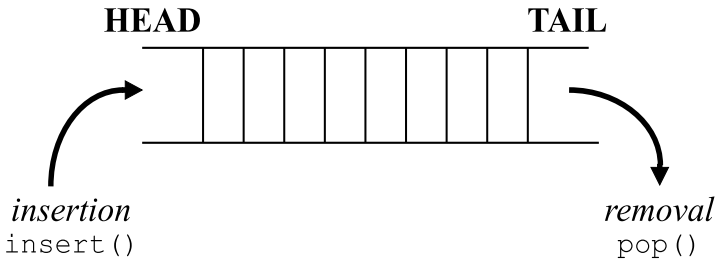

Figure 6.1: What is what with cQueue

The basic cQueue member functions dealing with insertion and removal are insert() and pop(). They are used like this:

```
cQueue queue("my-queue");
cMessage *msg;

// insert messages
for (int i=0; i<10; i++)
{
  msg = new cMessage;
  queue.insert( msg );
}

// remove messages
while( ! queue.empty() )
{
  msg = (cMessage *)queue.pop();
  delete msg;
}
```

The length() member function returns the number of items in the queue, and empty() tells whether there's anything in the queue.

There are other functions dealing with insertion and removal. The insertBefore() and insertAfter() functions insert a new item exactly before and after a specified one, regardless of the ordering function.

The tail() and head() functions return pointers to the objects at the tail and head of the queue, without affecting queue contents.

The pop()function can be used to remove items from the tail of the queue, and the remove() function can be used to remove any item known by its pointer from the queue:

queue.remove(msg);

Priority queue

By default, cQueue implements a FIFO, but it can also act as a priority queue, that is, it can keep the inserted objects ordered. If you want to use this feature, you have to provide a function that takes two cObject pointers, compares the two objects and returns -1, 0 or 1 as the result (see the reference for details). An example of setting up an ordered cQueue:

```
cQueue sortedqueue("sortedqueue", cObject::cmpbyname, true );
                       // sorted by object name, ascending
```

If the queue object is set up as an ordered queue, the insert() function uses the ordering function: it searches

the queue contents from the head until it reaches the position where the new item needs to be inserted, and inserts it there.

Iterators

Normally, you can only access the objects at the head or tail of the queue. However, if you use an iterator class, cQueueIterator, you can examine each object in the queue.

The cQueueIterator constructor takes two arguments, the first is the queue object and the second one specifies the initial position of the iterator: 0=tail, 1=head. Otherwise it acts as any other OMNeT++ iterator class: you can use the ++ and – operators to advance it, the () operator to get a pointer to the current item, and the end() member function to examine if you're at the end (or the beginning) of the queue.

An example:

```
for( cQueueIterator iter(queue,1); !iter.end(), iter++)
{
  cMessage *msg = (cMessage *) iter();
  //...
}
```

6.10.2 Expandable array: cArray

Basic usage

cArray is a container class that holds objects derived from cObject. cArray stores the pointers of the objects inserted instead of making copies. cArray works as an array, but if it gets full, it grows automatically. Internally, cArray is implemented with an array of pointers; if the array gets full, it is reallocated.

cArray objects are used in OMNeT++ to store parameters attached to messages, and internally, for storing module parameters and gates.

Creating an array:

```
cArray array("array");
```

Adding an object at the first free index:

```
cPar *p = new cPar("par");
int index = array.add( p );
```

Adding an object at a given index (if the index is occupied, you'll get an error message):

```
cPar *p = new cPar("par");
int index = array.addAt(5,p);
```

Finding an object in the array:

```
int index = array.find(p);
```

Getting a pointer to an object at a given index:

```
cPar *p = (cPar *) array[index];
```

You can also search the array or get a pointer to an object by the object's name:

```
int index = array.find("par");
Par *p = (cPar *) array["par"];
```

You can remove an object from the array by calling remove() with the object name, the index position or the object pointer:

```
array.remove("par");
array.remove(index);
array.remove( p );
```

The remove() function doesn't deallocate the object, but it returns the object pointer. If you also want to deallocate it, you can write:

```
delete array.remove( index );
```

Iteration

cArray has no iterator, but it's easy to loop through all the indices with an integer variable. The items() member function returns the largest index plus one.

```
for (int i=0; i<array.items(); i++)
{
  if (array[i]) // is this position used?
  {
    cObject *obj = array[i];
    ev << obj->name() << endl;
  }
}
```

6.11 Non-object container classes

There are two container classes to store non-object items: cLinkedList and cBag. The first one parallels with cQueue, the second one with cArray. They can be useful if you have to deal with C structs or objects that are not derived from cObject.

See the class library reference for more info about them.

6.12 The parameter class: cPar

6.12.1 Basic usage

cPar is a class that was designed to hold a value. The value is numeric (long or double) in the first place, but string, pointer and other types are also supported.

cPar is used in OMNeT++ in the following places:

```
\item{as module parameters}
\item{as message parameters}
```

There are many ways to set a cPar's value. One is the set...Value() member functions:

```
cPar pp("pp");
pp.setDoubleValue(1.0);
```

or by using overloaded operators:

```
cPar pp("pp");
pp = 1.0;
```

For reading its value, it is best to use overloaded type cast operators:

```
double d1 = (double)pp;
// or simply:
double d2 = pp;
```

Long integers:

```
pp = 89363L; // or:
pp.setLongValue( 89363L );
```

Character string:

```
pp = "hi there"; // or:
pp.setStringValue( "hi there" );
```

The cPar object makes its own copy of the string, so the original one does not need to be preserved. Short strings (less than ~20 chars) are handled more efficiently because they are stored in the object's memory space (and are not dynamically allocated).

There are several other types cPar can store: such as boolean, void* pointer; cObject* pointer, function with constant args; they will be mentioned in the next section.

For numeric and string types, an input flag can be set. In this case, when the object's value is first used, the parameter value will be searched for in the configuration (ini) file; if it is not found there, the user will be given a chance to enter the value interactively.

Examples:

```
cPar inp("inp");
inp.setPrompt("Enter my value:");
inp.setInput( true );   // make it an input parameter
double a = (double)inp; // the user will be prompted HERE
```

6.12.2 Random number generation through cPar

Setting cPar to call a function with constant arguments can be used to make cPar return random variables of different distributions:

```
cPar rnd("rnd");
rnd.setDoubleValue(intuniform, -10.0, 10.0);// uniform distr.
rnd.setDoubleValue(normal, 100.0, 5.0); // normal distr. (mean,dev)
rnd.setDoubleValue(exponential, 10.0); // exponential distr. (mean)
```

intuniform, normal etc. are ordinary C functions taking double arguments and returning double. Each time you read the value of a cPar containing a function like above, the function will be called with the given constant arguments (e.g. normal(100.0,5.0)) and its return value used.

The above functions use number 0 from the several random number generators. To use another generator, use the genk_xxx versions of the random functions:

```
rnd.setDoubleValue(genk\_normal, 3, 100.0, 5.0); // uses generator 3
```

A cPar object can also be set to return a random variable from a distribution collected by a statistical data collection object:

```
cDoubleHistogram hist =....; // the distribution
cPar rnd2("rnd2");
rnd2.setDoubleValue(hist);
```

6.12.3 Storing object and non-object pointers in cPar

cPar can store pointers to OMNeT++ objects. You can use both assignment and the setObjectValue() member function:

```
cQueue *queue = new cQueue("queue"); // just an example
cPar par1, par2;
par1 = (cObject *) queue;
par2.setObjectValue( queue );
```

To get the store pointer back, you can use typecast or the objectValue() member function:

```
cQueue *q1 = (cQueue *)(cObject *)par1;
cQueue *q2 = (cQueue *)par2.objectValue();
```

Whether the cPar object will own the other object or not is controlled by the takeOwnership() member function, just as with container classes. This is documented in detail in the class library reference. By default, cPar will own the object.

cPar can be used to store non-object pointers (for example C structs) or non-OMNeT++ object types in the parameter object. It works very similarly to the above mechanism. An example:

```
double *mem = new double[15];
cPar par1, par2;
par1 = (void *) mem;
par2.setPointerValue( (void *)mem );
...
double *m1 = (double *)(void *)par1;
double *m2 = (double *)par2.pointerValue();
```

Memory management can be specified by cPar's configPointer() member function. It takes three arguments: a pointer to a user-supplied deallocation function, a pointer to a user-supplied duplication function and an item size. If all three are 0 (NULL), no memory management is done, that is, the pointer is treated as a mere pointer. This is the default behaviour. If you supply only the item size (and both function pointers are NULL), cPar will use the delete operator to deallocate the memory area when the cPar object is destructed, and it will use new char[size] followed by a memcpy() to duplicate the memory area whenever the cPar object is duplicated. If you need more sophisticated memory management, you can supply your own deallocation and duplication functions. All this is documented in detail in the class library reference. An example for simple memory management:

```
double *mem = new double[15];
cPar par;
par.setPointerValue((void *) mem);
par.configPointer(NULL, NULL, 15*sizeof(double));
// -> now if par goes out of scope, it will delete the 15-double array.
```

The configPointer() setting only affects what happens when the cPar is deleted, duplicated or copied, but does *not* apply to assigning new pointers. That is, if *you* assign a new void* to the cPar, you simply overwrite the pointer – the block denoted by the old pointer is *not* deleted. This fact can be used to extract some dynamically allocated block out of the cPar: carrying on the previous example, you would extract the array of 15 doubles from the cPar like this:

```
double *mem2 = (double *)par.pointerValue();
par.setValue( (void *)0 );
// -> now par has nothing to do with the double[15] array
```

However, if you assign some non-pointer value to the cPar, beware: this *will* activate the memory management for the block. If you temporarily use the same cPar object to store other than void* ('P') values, the configPointer() setting is lost.

6.12.4 Reverse Polish expressions

This feature is rarely needed by the user, it is more used internally. A cPar object can also store expressions. In this case, the expression must be given in reversed Polish form. An example:

```
sXElem *expression = new sXElem[5];
expression[0] = &par( ''count'' ); // pointer to
module parameter
expression[1] = 1;
expression[2] = '+';
expression[3] = 2;
expression[4] = '/';

cPar expr("expr");
expr.setDoubleValue(expression,5);
```

The cPar object created above contains the *(count+1)/2* expression where *count* is a module parameter. Each time the cPar is evaluated, it recalculates the expression, using the current value of count. Note the & sign in front of par("count") expression: if it was not there, the parameter would be taken by value, evaluated once and then the resulting constant would be used.

Another example is a distribution with mean and standard deviation given by module parameters:

```
sXElem *expression = new sXElem[3];
expression[0] = &par("mean");
expression[1] = &par("stddev");
expression[2] = normal; // pointer to the normal(double,double) func.

cPar expr("expr");
expr.setDoubleValue(expression,3);
```

For more information, see the reference and the code NEDC generates for parameter expressions.

6.12.5 Using redirection

A cPar object can be set to stand for a value actually stored in another cPar object. This is called *indirect* or *redirected* value. When using redirection, every operation on the value (i.e. reading or changing it) will be actually done to the other cPar object:

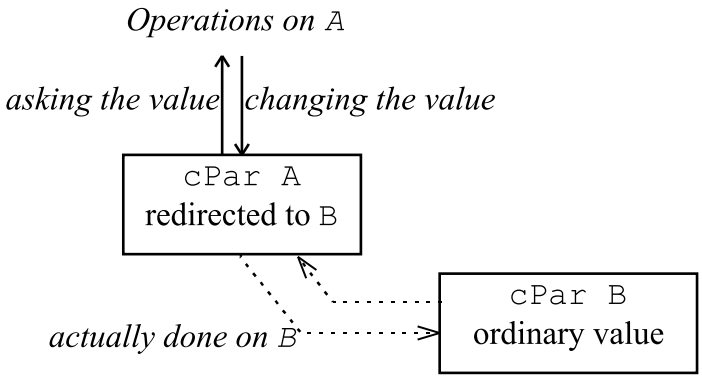

Figure 6.2: cPar redirection

Redirection is how module parameters taken by reference are implemented. The redirection does not include name strings. That is, if you say A->setName(”newname”) in the above example, A’s name will be changed as the name member is not redirected. (This is natural if you consider parameters taken by reference: a parameter should/can have different name than the value it refers to.)

You create a redirection with the setRedirection() function:

```
cPar *bb = new cPar("bb"); // background value
bb = 10L;
cPar a("a"); // we’ll redirect this object

a.setRedirection(bb); // create redirection
```

Now every operation you do on a’s value will be done to bb:

```
long x = a; // returns bb’s value, 10L
a = 5;      // bb’s value changes to 5
```

The only way to determine whether a is really holding the value or it is redirected to another cPar is to use the isRedirected() member function which returns a bool, or redirection() which returns the pointer to the background object, or NULL if there’s no redirection:

```
cPar *redir = a.redirection(); // returns bb’s pointer
if (redir != NULL)
  ev << "a is redirected to " << redir->name() << endl;
```

To break the link between the two objects, use the cancelRedirection() member function. (No other method will work, including assigning a the value of another cPar object.) The cancelRedirection() function gives the (long)0 value to the redirected object (the other will be unaffected). If you want to cancel the indirection but keep the old value, you can do something like this:

```
cPar *value = a.redirection();  // bb’s pointer
a.cancelRedirection();          // break the link; value of a is now 0\\
a = *value;                     // copy the contents of bb into a
```

6.12.6 Type characters

Internally, cPar objects identify the types of the stored values by type characters. The type character is returned by the type() member function:

```
cPar par = 10L;
char typechar = par.type(); // returns 'L'
```

The full table of type characters is presented in the *Summary* section below.

TBD isNumeric() function.

6.12.7 Summary

The various cPar types and the member functions used to manipulate them are summarized in the following table:

Type char	Type name	Member functions	Description

S	string	`setStringValue( const char *);`
		`const char * stringValue();`
		`op const char *();`
		`op=(const char *);`
			string value. Short strings (len<=27) are stored inside cPar object, without using heap allocation.

B	boolean	`setBoolValue(bool);`
		`bool boolValue();`
		`op bool();`
		`op=(bool);`
			boolean value. Can also be retrieved from the object as long (0 or 1).

L	long int	`setLongValue(long);`
		`long longValue();`
		`op long();`
		`op=(long);`
			signed long integer value. Can also be retrieved from the object as double.

D	double	`setDoubleValue(double);`
		`double doubleValue();`
		`op double();`
		`op=(double);`
			double-precision floating point value.

F	function	`setDoubleValue( MathFunc, [double], [double], [double]);`
		`double doubleValue();`
		`op double();`
			Mathematical function with constant arguments. The function is given by its pointer; it must take 0,1,2 or 3 doubles and return a double. This type is mainly used to generate random numbers: e.g. the function takes mean and standard deviation and returns random variable of a certain distribution.

X	expr.	`setDoubleValue( sXElem*,int);`
		`double doubleValue();`
		`op double();`
			Reverse Polish expression. Expression can contain constants, cPar objects, refer to other cPars (e.g. module parameters), can use many math operators (+-*/^% etc), function calls (function must take 0,1,2 or 3 doubles and return a double). The expression must be given is in an sXElem array (see later).

T	distrib.	`setDoubleValue( cStatistic*);`
		`double doubleValue();`
		`op double();`
			random variable generated from a distribution collected by a statistical data collection object (derived from cStatistic).

P	void* pointer	`setPointerValue(void*);`
		`void *pointerValue();`
		`op void *();`
		`op=(void *);`
			pointer to a non-cObject item (C struct, non-cObject object etc.) Memory management can be controlled through the config-Pointer() member function.

O	object pointer	`setObjectValue(cObject*);`
		`cObject *objectValue();`
		`op cObject *();`
		`op=(cObject *);`
			pointer to an object derived from cObject. Ownership management is done through takeOwnership().

I	indirect value	`setRedirection(cPar*);`
		`bool isRedirected();`
		`cPar *redirection();`
		`cancelRedirection();`
			value is redirected to another cPar object. All value setting and reading operates on the other cPar; even the type() function will return the type in the other cPar (so you'll never get 'I' as the type). This redirection can only be broken with the cancelRedirection() member function. Module parameters taken by REF use this mechanism.

6.13 Statistics and distribution estimation

6.13.1 cStatistic and descendants

There are several statistic and result collection classes: `cStdDev`, `cWeightedStdDev`, `LongHistogram`, `cDoubleHistogram`, `cVarHistogram`, `cPSquare` and `cKSplit`. They are all derived from the abstract base class cStatistic.

- `cStdDev` keeps number of samples, mean, standard deviation, minimum and maximum value etc.
- `cWeightedStdDev` is similar to cStdDev, but accepts weighted observations. cWeightedStdDev can be used for example to calculate time average. It is the only weighted statistics class.
- `cLongHistogram` and cDoubleHistogram are descendants of cStdDev and also keep an approximation of the distribution of the observations using equidistant (equal-sized) cell histograms.
- `cVarHistogram` implements a histogram where cells do not need to be the same size. You can manually add the cell (bin) boundaries, or alternatively, automatically have a partitioning created where each bin has the same number of observations (or as close to that as possible).
- `cPSquare` is a class that uses the P^2 algorithm described in [JCH85]. The algorithm calculates quantiles without storing the observations; one can also think of it as a histogram with equiprobable cells.
- `cKSplit` uses a novel, experimental method, based on an adaptive histogram-like algorithm. (Published papers about *k-split* can be downloaded from the OMNeT++ Web site; just go one level up in the directories: http://www.hit.bme.hu/phd/vargaa). Because k-split is not very well known, we'll devote a section to it.

Basic usage

One can insert an observation into a statistic object with the collect() function or the += operator (they are equivalent). cStdDev has the following methods for getting statistics out of the object: samples(), min(), max(), mean(), stddev(), variance(), sum(), sqrSum() with the obvious meanings. An example usage for cStdDev:

```
cStdDev stat("stat");
for (int i=0; i<10; i++)
  stat.collect( normal(0,1) );
long num_samples = stat.samples();
double smallest = stat.min(),
largest = stat.max();
double mean = stat.mean(),
standard_deviation = stat.stddev(),
variance = stat.variance();
```

6.13.2 Distribution estimation

Initialization and usage

The distribution estimation classes (the histogram classes, cPSquare and cKSplit) are derived from cDensityEstBase. Distribution estimation classes (except for cPSquare) assume that the observations are within a range. You may specify the range explicitly (based on some a-priori info about the distribution) or you may let the object collect the first few observations and determine the range from them. Methods which let you specify range settings are part of cDensityEstBase. The following member functions exist:

```
setRange(lower,upper);
setRangeAuto(num_firstvals, range_ext_factor);
setRangeAutoLower(upper, num_firstvals, range_ext_factor);
setRangeAutoUpper(lower, num, range_ext_factor);
```

setNumFirstVals(num_firstvals);

The following example creates a histogram with 20 cells and automatic range estimation:

```
cDoubleHistogram histogram("histogram", 20);
histogram.setRangeAuto(100,1.5);
```

Here, 20 is the number of cells (not including the underflow/overflow cells, see later), and 100 is the number of observations to be collected before setting up the cells. 1.5 is the range extension factor. It means that the actual range of the initial observations will be expanded 1.5 times and this expanded range will be used to lay out the cells. This method increases the chance that further observations fall in one of the cells and not outside the histogram range.

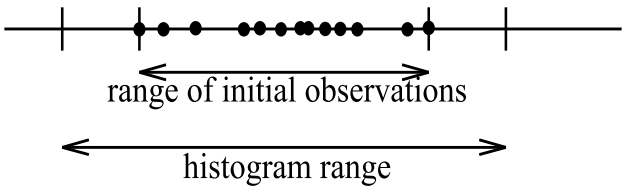

Figure 6.3: Setting up a histogram's range

After the cells have been set up, collecting can go on.

The transformed() function returns *true* when the cells have already been set up. You can force range estimation and setting up the cells by calling the transform() function.

The observations that fall outside the histogram range will be counted as underflows and overflows. The number of underflows and overflows are returned by the underflowCell() and overflowCell() member functions.

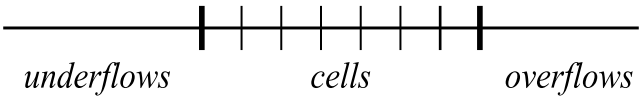

Figure 6.4: Histogram structure after setting up the cells

You create a P^2 object by specifying the number of cells:

```
cPSquare psquare("interarrival-times", 20);
```

Afterwards, a cPSquare can be used with the same member functions as a histogram.

Getting histogram data

There are three member functions to explicitly return cell boundaries and the number of observations is each cell. cells() returns the number of cells, basepoint(int k) returns the *k*th base point, cell(int k) returns the number of observations in cell *k*, and cellPDF(int k) returns the PDF value in the cell (i.e. between basepoint(k) and basepoint(k+1)). These functions work for all histogram types, plus cPSquare and cKSplit.

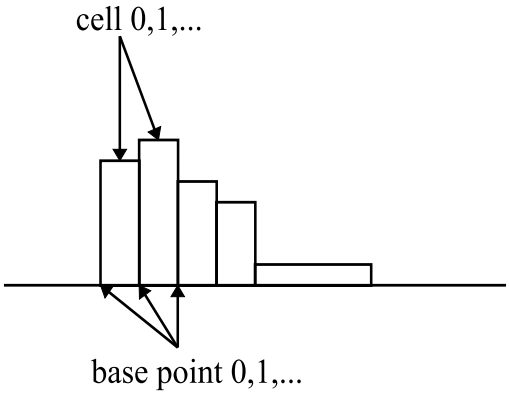

Figure 6.5: base points and cells

An example:

```
long n = histogram.samples();
for (int i=0; i<histogram.cells(); i++)
{
  double cellWidth = histogram.basepoint(i+1)-histogram.basepoint(i);
  int count = histogram.cell(i);
  double pdf = histogram.cellPDF(i);
  //...
}
```

The pdf(x) and cdf(x) member functions return the value of the probability density function and the cumulated density function at a given *x*, respectively.

Random number generation from distributions

The random() member function generates random numbers from the distribution stored by the object:

```
double rnd = histogram.random();
```

cStdDev assumes normal distribution.

You can also wrap the distribution object in a cPar:

```
cPar rnd_par("rnd_par");
rnd_par.setDoubleValue(&histogram);
```

The cPar object stores the pointer to the histogram (or P^2 object), and whenever it is asked for the value, calls the histogram object's random() function:

```
double rnd = (double)rnd_par; // random number from the cPSquare
```

Storing/loading distributions

The statistic classes have loadFromFile() member functions that read the histogram data from a text file. If you need a custom distribution that cannot be written (or it is inefficient) as a C function, you can describe it in histogram form stored in a text file, and use a histogram object with loadFromFile().

You can also use saveToFile()that writes out the distribution collected by the histogram object:

```
FILE *f = fopen("histogram.dat","w");
histogram.saveToFile( f ); // save the distribution
fclose( f );

FILE *f2 = fopen("histogram.dat","r");}
cDoubleHistogram hist2("Hist-from-file");
hist2.loadFromFile( f2 ); // load stored distribution
fclose( f2 );
```

Histogram with custom cells

`cVarHistogram` class. TBD comments.

Now we do support the following 2 uses of cVarHistogram:

- add all the boundaries (manually) before collecting samples
- collect samples and transform() makes the boundaries

Transform types for cVarHistogram:

- `HIST_TR_NO_TRANSFORM`: no transformation; uses bin boundaries previously defined by addBinBound()
- `HIST_TR_AUTO_EPC_DBL`: automatically creates equiprobable cells
- `HIST_TR_AUTO_EPC_INT`: like the above, but uses a different hack :-)

Creating an object:

```
cVarHistogram(const char *s=NULL,
              int numcells=11,
              int transformtype=HIST_TR_AUTO_EPC_DBL);
```

Manually adding a cell boundary:

```
void addBinBound(double x);
```

Rangemin and rangemax is chosen after collecting the num_firstvals initial observations. One cannot add cell boundaries when the histogram has already been transformed.

6.13.3 The k-split algorithm

Purpose

The k-split algorithm is an on-line distribution estimation method. It was designed for on-line result collection in simulation programs. The method was proposed by Varga and Fakhamzadeh in 1997. The primary advantage of k-split is that without having to store the observations, it gives a good estimate without requiring a-priori information about the distribution, including the sample size. The k-split algorithm can be extended to multi-dimensional distributions, but here we deal with the one-dimensional version only.

The algorithm

The k-split algorithm is an adaptive histogram-type estimate which maintains a good partitioning by doing cell splits. We start out with a histogram range $[x_{lo}, x_{hi})$ with k equal-sized histogram cells with observation counts $n_1, n_2, \cdots n_k$. Each collected observation increments the corresponding observation count. When an observation count n_i reaches a *split threshold*, the cell is split into k smaller, equal-sized cells with observation counts $n_{i,1}, n_{i,2}, \cdots n_{i,k}$ initialized to zero. The n_i observation count is remembered and is called the *mother observation count* to the newly created cells. Further observations may cause cells to be split further (e.g. $n_{i,1,1}, ...n_{i,1,k}$ etc.), thus creating a k-order tree of observation counts where leaves contain live counters that are actually incremented by new observations, and intermediate nodes contain mother observation counts for their children. If an observation falls outside the histogram range, the range is extended in a natural manner by inserting new level(s) at the top of the tree. The fundamental parameter to the algorithm is the split factor k. Low values of k, $k = 2$ and $k = 3$ are to be considered. In this paper we examine only the $k = 2$ case.

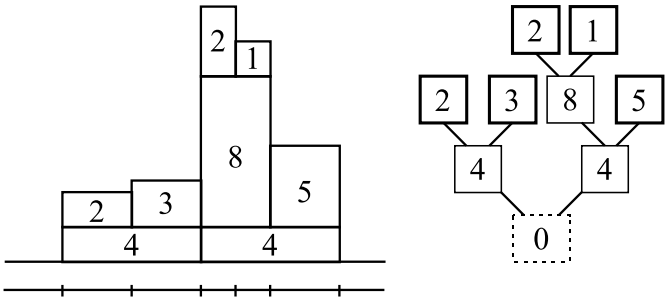

Figure 6.6: Illustration of the k-split algorithm, $k = 2$. The numbers in boxes represent the observation count values

For density estimation, the total number of observations that fell into each cell of the partition has to be determined. For this purpose, mother observations in each internal node of the tree must be distributed among its child cells and propagated up to the leaves.

Let $n_{...,i}$ be the (mother) observation count for a cell, $s_{...,i}$ be the total observation count in a cell $n_{...,i}$ plus the observation counts in all its sub-, sub-sub-, etc. cells), and $m_{...,i}$ the mother observations propagated to the cell. We are interested in the $\tilde{n}_{...,i} = n_{...,i} + m_{...,i}$ estimated amount of observations in the tree nodes, especially in the leaves. In other words, if we have $\tilde{n}_{...,i}$ estimated observation amount in a cell, how to divide it to obtain $m_{...,i,1}, m_{...,i,2} \cdots m_{...,i,k}$ that can be propagated to child cells. Naturally, $m_{...,i,1} + m_{...,i,2} + \cdots + m_{...,i,k} = \tilde{n}_{...,i}$.

Two natural distribution methods are even distribution (when $m_{...,i,1} = m_{...,i,2} = \cdots = m_{...,i,k}$) and proportional distribution (when $m_{...,i,1} : m_{...,i,2} : \cdots : m_{...,i,k} = s_{...,i,1} : s_{...,i,2} : \cdots : s_{...,i,k}$). Even distribution is optimal when the $s_{...,i,j}$ values are very small, and proportional distribution is good when the $s_{...,i,j}$ values are large compared to $m_{...,i,j}$. In practice, a linear combination of them seems appropriate, where $\lambda = 0$ means even and $\lambda = 1$ means proportional distribution:

$$m_{\ldots,i,j} = (1-\lambda)\frac{\tilde{n}_{\ldots,i}}{k} + \lambda\tilde{n}_{\ldots,i}\frac{s_{\ldots,i,j}}{s_{\ldots,i}}, \lambda \in [0,1] \tag{6.1}$$

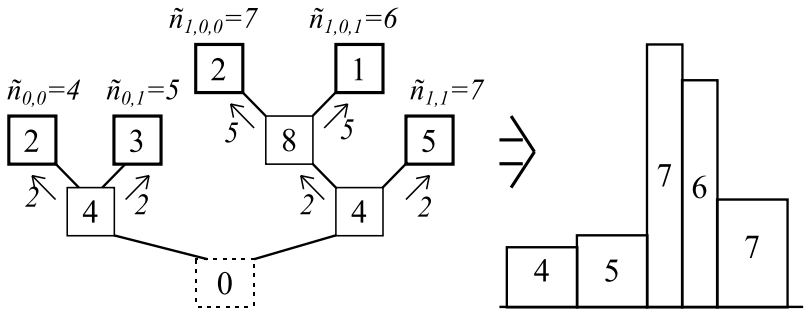

Figure 6.7: Density estimation from the k-split cell tree. We assume $\lambda = 0$, i.e. we distribute mother observations evenly.

Note that while $n_{\ldots,i}$ are integers, $m_{\ldots,i}$ and thus $\tilde{n}_{\ldots,i}$ are typically real numbers. The histogram estimate calculated from k-split is not exact, because the frequency counts calculated in the above manner contain a degree of estimation themselves. This introduces a certain *cell division error*; the λ parameter should be selected so that it minimizes that error. It has been shown that the cell division error can be reduced to a more-than-acceptable small value.

Strictly speaking, the k-split algorithm is semi-online, because its needs some observations to set up the initial histogram range. However, because of the range extension and cell split capabilities, the algorithm is not very sensitive to the choice of the initial range, so very few observations are enough for range estimation (say $N_{pre} = 10$). Thus we can regard k-split as an on-line method.

K-split can also be used in semi-online mode, when the algorithm is only used to create an optimal partition from a larger number of N_{pre} observations. When the partition has been created, the observation counts are cleared and the N_{pre} observations are fed into k-split once again. This way all mother (non-leaf) observation counts will be zero and the cell division error is eliminated. It has been shown that the partition created by k-split can be better than both the equi-distant and the equal-frequency partition.

OMNeT++ contains an experimental implementation of the k-split algorithm, the cKSplit class. Research on k-split is still under way.

The cKSplit class

TBD comments

Member functions:

```
void setCritFunc(KSplitCritFunc _critfunc, double *_critdata);
void setDivFunc(KSplitDivFunc \_divfunc, double *\_divdata);
void rangeExtension( bool enabled );

struct sGrid
{
  int parent;   // index of parent grid
  int reldepth; // depth = (reldepth - rootgrid's reldepth)
  long total;   // sum of cells & all subgrids (includes 'mother')
  int mother;   // observations 'inherited' from mother cell
  int cells[K]; // cell values
};

int treeDepth();
```

```
int treeDepth(sGrid& grid);

double realCellValue(sGrid& grid, int cell);
void printGrids();

sGrid& grid(int k);
sGrid& rootGrid();
```

6.13.4 Transient detection and result accuracy

In many simulations, only the steady state performance (i.e. the performance after the system has reached a stable state) is of interest. The initial part of the simulation is called the transient period. After the model has entered steady state, simulation must proceed until enough statistical data have been collected to compute result with the required accuracy.

Detection of the end of the transient period and a certain result accuracy is supported by OMNeT++. The user can attach transient detection and result accuracy objects to a result object (cStatistic's descendants). The transient detection and result accuracy objects will do the specific algorithms on the data fed into the result object and tell if the transient period is over or the result accuracy has been reached.

The base classes for classes implementing specific transient detection and result accuracy detection algorithms are:

- cTransientDetection: base class for transient detection
- cAccuracyDetection: base class for result accuracy detection

Basic usage

TBD comments

Attaching detection objects to a cStatistic and getting pointers to the attached objects:

```
addTransientDetection(cTransientDetection *object);
addAccuracyDetection(cAccuracyDetection *object);
cTransientDetection *transientDetectionObject();
cAccuracyDetection *accuracyDetectionObject();
```

Detecting the end of the period:

- polling the detect() function of the object
- installing a post-detect function

Transient detection

Currently one transient detection algorithm is implemented, i.e. there's one class derived from cTransientDetection. The cTDExpandingWindows class uses the sliding window approach with two windows, and checks the difference of the two averages to see if the transient period is over.

```
void setParameters(int reps=3,
                   int minw=4,
                   double wind=1.3,
                   double acc=0.3);
```

Accuracy detection

Currently one transient detection algorithm is implemented, i.e. there's one class derived from cAccuracyDetection. The algorithm implemented in the cADByStddev class is: divide the standard deviation by the square of the number of values and check if this is small enough.

```
void setParameters(double acc=0.1,
                   int reps=3);
```

6.14 Recording simulation results

6.14.1 Output vectors: cOutVector

Objects of type cOutVector are responsible for writing time series data (referred to as *output vectors*) to a file. The record() member is used to output a value (or a value pair) with a timestamp.

It can be used like this:

```
cOutVector resp_v("response time");

while (...)
{
  double response_time;
  //...
  resp_v.record( response_time );\\
  //...
}
```

All cOutVector objects write to the same, common file. The file is textual; each record() call generates a line in the file. The output file can be processed using Plove, but otherwise its simple format allows it to be easily processed with sed, awk, grep and the like, and it can be imported by spreadsheet programs. The file format is described later in this manual (in the section about simulation execution).

You can disable the output vector or specify a simulation time interval for recording either from the ini file or directly from program code:

```
cOutVector v("v");
simtime_t t =...;

v.enable();
v.disable();
v.setStartTime( t );
v.setStopTime( t+100.0 );
```

If the output vector object is disabled or the simulation time is outside the specified interval, record() doesn't write anything to the output file. However, if you have a Tkenv inspector window open for the output vector object, the values will be displayed there, regardless of the state of the output vector object.

6.14.2 Output scalars

While output vectors are to record time series data and thus they typically record a large volume of data during a simulation run, output scalars are supposed to record a single value per simulation run. You can use outputs scalars

- to record summary data at the end of the simulation run

- to do several runs with different parameter settings/random seed and determine the dependence of some measures on the parameter settings. For example, multiple runs and output scalars are the way to produce *Throughput vs. Offered Load* plots.

Output scalars are recorded with the recordScalar() member function. It is overloaded, you can use it to write doubles and strings (const char *):

```
double avg_throughput = total_bits / simTime();
recordScalar("Average throughput", avg_throughput);
```

You can record whole statistics objects by calling recordStats():

```
cStdDev *eedstats = new cStdDev;
...
recordStats("End-to-end Statistics", eedstats);
```

Calls to recordScalar() and recordStats() are usually placed in the redefined finish() member function of a simple module.

The above calls write into the (textual) output scalar file. The output scalar file is preserved across simulation runs (unlike the output vector file is, scalar files are not deleted at the beginning of each run). Data are always appended at the end of the file, and output from different simulation runs are separated by special lines.

6.15 Deriving new classes

Nearly all classes in the simulation class library are descendants of cObject. If you want to derive a new class from cObject or a cObject descendant, you must redefine some member functions so that objects of the new type can fully co-operate with other parts of the simulation system. A more-or-less complete list of these functions are presented here. Do not be embarrassed at the length of the list: most functions are not absolutely necessary to implement. For example, you do not need to redefine forEach() unless your class is a container class.

- default constructor, copy constructor. The copy constructor can simply call the assignment operator.
- `operator=()`: the assignment operator (copies object contents from another object)
- `dup()`: duplicates the object by creating an exact copy (uses copy constructor)
- `className()`: returns class name string
- `info()`: returns a one-line info about object contents
- `writeContents()`: write a more detailed report about the object into a file
- `forEach()`: iterates through all contained objects if any
- `netPack()`, `netUnpack()`: they are needed only if objects of this type will be sent over PVM/MPI from one segment to another. The netPack(),netUnpack()functions of the library classes are in the sim/pvm (sim/mpi) directory.
- `inspectorFactoryName()`: used by Tkenv to create inspector windows for objects of this type.

One should also use the Register_Class() macro to register the new class. It is used by the createOne() function and the PVM/MPI extension of OMNeT++.

Let us see a simple example. The header file:

```
// File: cmyclass.h
#include "cobject.h"

class cMyClass : public cObject
{
 public:
  int samples;

  cMyClass(const cMyClass& myclass);
  cMyClass(const char *name=NULL, int k=0);
  virtual ~cMyClass() {}
  virtual const char *className() const {return "cMyClass";}
  virtual cObject *dup() const {return new cMyClass(*this);}
  virtual void info(char *buf);
  virtual void writeContents(ostream& os);
  cMyClass& operator=(const cMyClass& myclass);
};
```

The corresponding .cc file:

```
// File: cmyclass.cc
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include "cmyclass.h"

Register_Class( cMyClass );

cMyClass::cMyClass(const cMyClass& myclass) : cObject()
{
  setName( myclass.name() );
  operator=( myclass );
}

cMyClass::cMyClass(const char *name, int k) : cObject( name )
{
  samples = k;
}

void cMyClass::info(char *buf)
{
  cObject::info( buf );
  sprintf( buf+strlen(buf), " samples=%d", samples);
}

void cMyClass::writeContents(ostream& os)
{
  os << " samples: " << samples << '\n';
}

cMyClass& cMyClass::operator=(const cMyClass& myclass)
{
  cObject::operator=(myclass);
  samples = myclass.samples;
```

```
}
```

See the virtual functions of cObject in the class library reference for more information.

6.16 Tracing and debugging aids

6.16.1 Displaying information about module activity

The global object called ev represents the user interface of the simulation program. You can send data to ev using the C++-style I/O operator (<<).

```
ev << "started\n";
ev << "about to send message #" <<  i << endl;
ev << "queue full, discarding packet\n";
```

The exact way messages are displayed to the user depends on the user interface. In the command-line user interface (Cmdenv), it is simply dumped to the standard output. (This output can also be disabled from the ini file so that it doesn't slow down simulation when it is not needed.) In windowing user interfaces (Tkenv), each simple module can have a separate text output window.

The above means that you should *not* use `printf`, `cout <<` and the like because with Tkenv, their output would appear in the xterm window behind the graphical window of the simulation application.

The user can also specify a phase string that is displayed at the top of the text output windows. The phase string can indicate what the module is currently doing.

```
setPhase("starting up");
for(;;)
{
  setPhase("idle");
  //...
  setPhase("opening connection");
  ev << "connection request from " << src << "\n";
  //..
  setPhase("connection alive");
  //..
  setPhase("closing connection");
  //...
}
```

Writing out informative messages at strategic points of the code is an effective way debugging.

6.16.2 Watches

You may want some of your int, long, double, char, etc. variables to be inspectable in Tkenv and to be output into the snapshot file. In this case, you can create `cWatch` objects for them with the `WATCH` macro:

```
int i; WATCH(i);
char c; WATCH(c);
```

Tkenv also lets you change the value of the WATCHed variables.

The `WATCH()` macro expands to a dynamically created `cWatch` object. The object remembers the address and type of your variable. The macro expands to something like:

```
new cWatch("i",i);
```

You can also make a WATCH for pointers of type `char*` or `cObject*`, but this may cause a segmentation fault if the pointer does not point to a valid location when Tkenv or `snapshot()` wants to use it.

You can also set watches for variables that are members of the module class or for structure fields:

```
WATCH( lapbconfig.timeout );
```

Placement of WATCHes

Be careful not to execute a WATCH statement more than once, as each call would create a new cWatch object! If you use activity(), the best place for WATCHes is the top of the activity() function. If you use handleMessage(), place the WATCH() statement into initialize(). WATCH() creates a dynamic cWatch object, and we do not want to create a new object each time handleMessage() is called.

6.16.3 Snapshots

The snapshot() function outputs textual information about all or selected objects of the simulation (including the objects created in module functions by the user) into the snapshot file.

```
bool snapshot(cObject *obj = &simulation, const char *label = NULL);
```

The function can be called from module functions, like this:

```
snapshot();     // dump the whole network
snapshot(this); // dump this simple module and all its objects
snapshot(&putAsideQueue);       // dump queue contents
snapshot(&simulation.msgQueue); // dump future events
```

This will append snapshot information to the end of the snapshot file. (The snapshot file name has an extension of .sna, default is omnetpp.sna. Actual file name can be set in the config file.)

The snapshot file output is detailed enough to be used for debugging the simulation: by regularly calling snapshot(), one can trace how the values of variables, objects changed over the simulation. The arguments: label is a string that will appear in the output file; obj is the object whose inside is of interest. By default, the whole simulation (all modules etc) will be written out.

If you run the simulation with Tkenv, you can also create a snapshot from the menu.

An example of a snapshot file:

```
================================================
|| SNAPSHOT ||
================================================
| Of object:    'simulation'
| Label:        'three-station token ring'
| Sim. time:    0.0576872457 ( 57ms)
| Network:      'token'
| Run no.       1
| Started at:   Mar 13, 1997, 14:23:38
| Time:         Mar 13, 1997, 14:27:10
| Elapsed:      5 sec
| Initiated by: operator
================================================
```

```
(cSimulation) 'simulation' begin
  Modules in the network:
    'token' #1 (TokenRing)
      'comp[0]' #2 (Computer)
        'mac' #3 (TokenRingMAC)
        'gen' #4 (Generator)
        'sink' #5 (Sink)
      'comp[1]' #6 (Computer)
        'mac' #7 (TokenRingMAC)
        'gen' #8 (Generator)
        'sink' #9 (Sink)
      'comp[2]' #10 (Computer)
        'mac' #11 (TokenRingMAC)
        'gen' #12 (Generator)
        'sink' #13 (Sink)
end

(cCompoundModule) 'token' begin
  #1 params     (cArray) (n=6)
  #1 gates      (cArray) (empty)
  comp[0]          (cCompoundModule,#2)
  comp[1]          (cCompoundModule,#6)
  comp[2]          (cCompoundModule,#10)
end

(cArray) 'token.parameters' begin
  num_stations (cModulePar) 3 (L)
  num_messages (cModulePar) 10000 (L)
  ia_time      (cModulePar) truncnormal(0.005,0.003) (F)
  THT          (cModulePar) 0.01 (D)
  data_rate    (cModulePar) 4000000 (L)
  cable_delay  (cModulePar) 1e-06 (D)
end

(cModulePar) 'token.num_stations' begin
  Type: L
  Value: 3
end

[...token.num_messages omitted...]

(cModulePar) 'token.ia_time' begin
  Type:  F
  Value: truncnormal(0.005,0.003)
end

[...rest of parameters & gates stuff deleted from here...]

(cCompoundModule) 'token.comp[0]' begin
  parameters    (cArray) (empty)
  gates         (cArray) (n=2)
  mac           (TokenRingMAC,#3)
```

```
  gen           (Generator,#4)
  sink          (Sink,#5)
end

(cArray) 'token.comp[0].parameters' begin
end

(cArray) 'token.comp[0].gates' begin
  in            (cGate)  <-- comp[2].out
  out           (cGate)  --> D --> comp[1].in
end

(cGate) 'token.comp[0].in' begin
  type:  input
  inside connection:  token.comp[0].mac.phy_in
  outside connection: token.comp[2].out
  delay: -
  error: -
  data rate: -
end

(cGate) 'token.comp[0].out' begin
  type: output
  inside connection: token.comp[0].mac.phy_out
  outside connection: token.comp[1].in
  delay: (cPar) 1e-06 (D)
  error: -
  data rate: -
end

(TokenRingMAC) 'token.comp[0].mac' begin
  parameters    (cArray) (n=2)
  gates         (cArray) (n=4)
  local-objects (cHead)
  class-data-members (cHead)
  putaside-queue (cQueue) (empty)
end

[...comp[0].mac parameters stuff deleted from here...]

(cArray) 'token.comp[0].mac.gates' begin
  phy_in        (cGate)  <-- <parent>.in
  from_gen      (cGate)  <-- gen.out
  phy_out       (cGate)  --> <parent>.out
  to_sink       (cGate)  --> sink.in
end

[...detailed gate list deleted from here...]

(cHead) 'token.comp[0].mac.local-objects' begin
  sendqueue-length (cOutVector) (single)
  send-queue   (cQueue) (n=11)
end
```

```
(cOutVector) 'token.comp[0].mac.local-objects.sendqueue-length' begin
end

(cQueue) 'token.comp[0].mac.local-objects.send-queue' begin
  0-->1         (cMessage) Tarr=0.0158105774 ( 15ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0163553310 ( 16ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0205628236 ( 20ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0242203591 ( 24ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0300994268 ( 30ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0364005251 ( 36ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0370745702 ( 37ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0387984129 ( 38ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0457462493 ( 45ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0487308918 ( 48ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0514466766 ( 51ms) Src=#4 Dest=#3
end

(cMessage) 'token.comp[0].mac.local-objects.send-queue.0-->1' begin
  #4 --> #3
  sent:         0.0158105774 ( 15ms)
  arrived:      0.0158105774 ( 15ms)
  length:       33536
  kind:         0
  priority:     0
  error:        FALSE
  time stamp:   0.0000000 ( 0.00s)
  parameter list:
    dest        (cPar) 1 (L)
    source      (cPar) 0 (L)
    gentime     (cPar) 0.0158106 (D)
end

(cArray) 'token.comp[0].mac.local-objects.send-queue.0-->1.par-vector' begin
  dest          (cPar) 1 (L)
  source        (cPar) 0 (L)
  gentime       (cPar) 0.0158106 (D)
end

[...message parameters and the other messages' stuff deleted...]

(cHead) 'token.comp[0].mac.class-data-members' begin
end

(cQueue) 'token.comp[0].mac.putaside-queue' begin
end

[...comp[0].gen and comp[0].sink stuff deleted from here...]
[...whole comp[1] and comp[2] stuff deleted from here...]

(cMessageHeap) 'simulation.message-queue' begin
  1-->0         (cMessage) Tarr=0.0576872457 ( 57ms) Src=#8 Dest=#7
                (cMessage) Tarr=0.0577201630 ( 57ms) Mod=#8 (selfmsg)
                (cMessage) Tarr=0.0585677054 ( 58ms) Mod=#4 (selfmsg)
                (cMessage) Tarr=0.0594939072 ( 59ms) Mod=#12 (selfmsg)
```

```
                (cMessage) Tarr=0.0601010000 ( 60ms) Mod=#7 (selfmsg)
  1-->2         (cMessage) Tarr=0.0601020000 ( 60ms) Src=#11 Dest=#13
end

[...detailed list of message queue contents deleted from here...]
```

To reduce the size of the file, you may well decide to make a snapshot only of a part of the model. This example reports only about the current simple module's put-aside queue:

```
snapshot(&putAsideQueue);
```

6.16.4 Breakpoints

With activity() only! In those user interfaces which support debugging, breakpoints stop execution and the state of the simulation can be examined.

You can set a breakpoint inserting a breakpoint() call into the source:

```
for(;;)
{
  cMessage *msg = receive();
  breakpoint("before-processing");
  breakpoint("before-send");
  send( reply_msg, "out" );
  //..
}
```

In user interfaces that do not support debugging, breakpoint() calls are simply ignored.

6.16.5 Disabling warnings

Some container classes and functions suspend the simulation and issue warning messages in potentially bogus/dangerous situations, for example when an object is not found and NULL pointer/reference is about to be returned. Very often this is useful, but sometimes it is more trouble. You can turn warnings on/off from the ini file (warnings=yes/no).

It is a good practice to leave warnings enabled, and temporarily disable warnings in places where OMNeT++ would normally issue warnings but you know the code is correct. This is done in the following way:

```
bool w = simulation.warnings();
simulation.setWarnings( false );
...
... // critical code
...
simulation.setWarnings( w );
```

6.16.6 Getting coroutine stack usage

If is important to choose the correct stack size for modules. If the stack is too large, it unnecessarily consumes memory; if it is too small, stack violation occurs.

From the Feb99 release, OMNeT++ contains a mechanism that detects stack overflows. It checks the intactness of a predefined byte pattern (0xdeadbeef) at the stack boundary, and reports "stack violation" if

it was overwritten. The mechanism usually works fine, but occasionally it can be fooled by large – and not fully used – local variables (e.g. char buffer[256]): if the byte pattern happens to fall in the middle of such a local variable, it may be preserved intact and OMNeT++ does not detect the stack violation.

To be able to make a good guess about stack size, you can use the stackUsage() call which tells you how much stack the module actually uses. It is most conveniently called from finish():

```
void FooModule::finish()
{
  ev << stackUsage() <<  "bytes of stack used\n";
}
```

The value includes the extra stack added by the user interface library (see *extraStackforEnvir* in envir/omnetapp.h), which is currently 8K for Cmdenv and at least 16K for Tkenv (the actual value is dependent on the operating system, e.g. SUN Solaris needs more space).

stackUsage()also works by checking the existence of predefined byte patterns in the stack area, so it is also subject to the above effect with local variables.

6.17 Changing the network graphics at run-time

Sometimes it is useful to change the appearance of some components in the network graphics, such as the color of the modules, color/width of connection arrows etc.

The appearance of nodes and connections is determined by the display strings. Display strings are initially taken from the NED description (stuff like: `display: "p=100,10;i=pc"`). You can change the display string of a module or connection arrow at run-time by calling setDisplayString(). The display string of a connection arrow is stored in its source gate. Display string changes will immediately take effect.

Setting the module's appearance when it is displayed as a component within a compound module$^{(new)}$:

```
setDisplayString("p=100,100;b=60,30,rect;o=red,black,3", true);
```

Setting appearance of a compound module when it's displayed as a bounding box for its submodules:

```
parentModule()->setDisplayStringAsParent(''p=100.....", true);
```

Setting appearance of a connection, via its source gate:

```
gate("out")->setDisplayString("o=yellow,3");
```

The setDisplayString() methods additionally take a bool argument called immediate. It specifies whether the display string change should take effect immediately, or only after processing the current event (the default is *immediate=true*). If several display string changes are going to be done within one event, then *immediate=false* is useful because it reduces the number of necessary redraws. *Immediate=false* also uses less stack. But its drawback is that a setDisplayString() followed by a send() would actually be displayed in reverse order (message animation first), because message animations are performed immediately (actually within the send() call).

6.18 Tips for speeding up the simulation

Here are a few tips that can help you make the simulation faster:

- Turn off the display of screen messages when you run the simulation. You can do this in the ini file. Alternatively, you can place #ifdefs around your ev<< and ev.printf() calls and turn off the define when compiling the simulation for speed.
- Store the module parameters in local variables to avoid calling cPar member functions every time.
- Use gate numbers instead of gate names.
- Try to minimize message creations and deletions. Reuse messages if possible.
- Do not give name strings to objects that are created and deleted many times (pass NULL pointer as name).
- Use numeric index to get an object from a cArray, not the object name. You can do this also with message parameters.

Two techniques are discussed here in detail:

- message subclassing, and
- using shared objects

6.18.1 Using shared objects

In a complex simulation, a lot of messages are created, sent and destroyed. Messages typically have some parameters attached to them as cPar objects and it frequently happens that a certain parameter has identical values in all messages (for example, source address in a frame is the same in all messages sent by one module). Still, separate parameter objects are created and destroyed with each message, which is very costly. One could save significant amount of CPU time and memory if a single object could serve as a parameter to all existing messages.

This can be achieved with proper ownership control. See the following example:

```
void MyComputer::activity()
{
  cPar source_addr;     // address of this node
  cPar dest_addrs[3];   // possible destinations

  source_addr.setStringValue( "DECnet000728" );
  dest_addrs[0].setStringValue( "cisco_F99030" );
  dest_addrs[1].setStringValue( "DEC___28E6AD" );
  dest_addrs[2].setStringValue( "DECnet000B04" );

  long k=0;
  for(;;)
  {
    cMessage *packet = new cMessage("DATA");

    packet->addPar( *new cPar("sequence", 'L', k++) );

    packet->parList().takeOwnership( false ); // NOTE THIS LINE!!!

    packet->addPar( source_addr );
    packet->addPar( dest_addrs[ k%3 ] );

    send(packet, "output-gate");
```

```
    wait( truncnormal(1.5, 0.5) );
  }
}
```

The above simple module code models the message generation part of a computer on a LAN. The module sends out messages (packets) to different stations in every 1.5 seconds or so. The messages have three parameters: the source address, the destination address and a sequence number. The source address is the same in each packet, and there are only three possible destination stations. The sequence number is different in each packet.

To avoid the overhead caused by having to create source and destination address objects for each message, the module creates these objects only once; they will be shared among all messages. Separate sequence number objects are created for each message.

Let us see what happens to the sequence number object when it is inserted into the message. The message object, by default, takes the ownership of the object. Ownership means the *responsibility of destruction*; that is, when the message is destroyed, the parameter object will be destroyed as well.

This is exactly what we need most of the time. But if we just added the *shared* source and destination address objects to a message, then we would have problems when the message is destroyed. Somehow it must be told to the message object to leave our shared parameters alone and not to become their owner. This is exactly what the

```
packet->parList().takeOwnership(false);
```

line does: it sets a flag that tells the message (to be more precise, to its internal parameter list object) not to take the ownership of objects that will be inserted from then on. It does not affect objects already inserted. As a result, all messages will just hold pointers to the shared cPar objects and never do any harm to them.

The above example shows that with CPU-intensive simulations, you can save a lot of computation time and memory just by using the ownership mechanism already present in OMNeT++.

6.19 Building large networks

There are situations when using NED files to describe network topology is inconvenient, for example because the topology information comes from an external source (e.g. it is exported from a network management program). In such case, you have two possibilities to avoid writing NED files by hand:

1. generating NED files from data files
2. building the network from C++ code

The two solutions have different advantages and disadvantages. The first is more useful in the model development phase, while the second one is better for writing larger scale, more productized simulation programs. In the next sections we examine both methods.

6.19.1 Generating NED files

Text processing programs like awk or perl are excellent tools to read in textual data files and generate NED files from them. Perl also has extensions to access SQL databases, so it can also be used if the network topology is stored in a database.

The advantage is that the necessary awk or perl program can be written in a releatively short time, and it is inexpensive to maintain afterwards: if the structure of the data files change, the NED-creating program

can be easily modified. The disadvantage is that the resulting NED files are often quite big and the C++ compilation of the *_n.cc files take too long.

This method is best suited in the first phase of a simulation project when the topology, the format of the data files, etc. have not yet settled down.

6.19.2 Building the network from C++ code

Another alternative is to write C++ code which becomes part of the simulation executable. The code would read the topology data from data files or a database, and build the network directly. The code which builds the network would be quite similar to the *_n.cc files output by nedc.

Since writing such code is more complex than letting perl generate NED files, this method is recommended when the simulation program has to be somewhat more productized, for example when OMNeT++ and the simulation model is embedded into a larger program, e.g. a network design tool.

Chapter 7

Building Simulation Programs

7.1 Overview

As it was already mentioned, an OMNeT++ model physically consists of the following parts:

- NED language topology description(s). These are files with the .ned suffix.
- Simple modules. These are C++ files, with .cc suffix.

Model files are usually placed in the projects/modelname subdirectory of the main OMNeT++ directory.

The NED files are compiled into C++ using the NEDC compiler which is part of OMNeT++. The NEDC compiler (source and executable) is normally located in the nedc subdirectory of the main OMNeT++ directory.

The simulation system provides the following components that will be part of the simulation executable:

- Simulation kernel with the simulation class library. This is a library file with .a or .lib extension, normally in the sim subdirectory of the main OMNeT++ directory. It comes in several versions: libsim_std.a (sim_std.lib) is the standard version and libsim_pvm.a (sim_pvm.lib) and libsim_mpi.a (sim_mpi.lib) are the ones to be used with parallel execution.
- User interfaces. These are also library files (.a or .lib file), normally in the envir directory and other directories. The common part of all user interfaces is libenvir.a (envir.lib), and the specific user interfaces are libcmdenv.a (cmdenv.lib), libtkenv.a (tkenv.lib).

Simulation programs are built from the above components. First, the NED files are compiled into C++ source code using the NEDC compiler. Then all C++ sources are compiled and linked with the simulation kernel and a user interface to form a simulation executable.

The following figure gives an overview of the process of building and running simulation programs.

This section discusses how to use the simulation system on the following platforms:

- Unix with gcc installed (and which is similar, Cygwin on Windows NT)
- MSVC 6.0 on Windows NT
- Borland C++ 5.0 on Windows NT

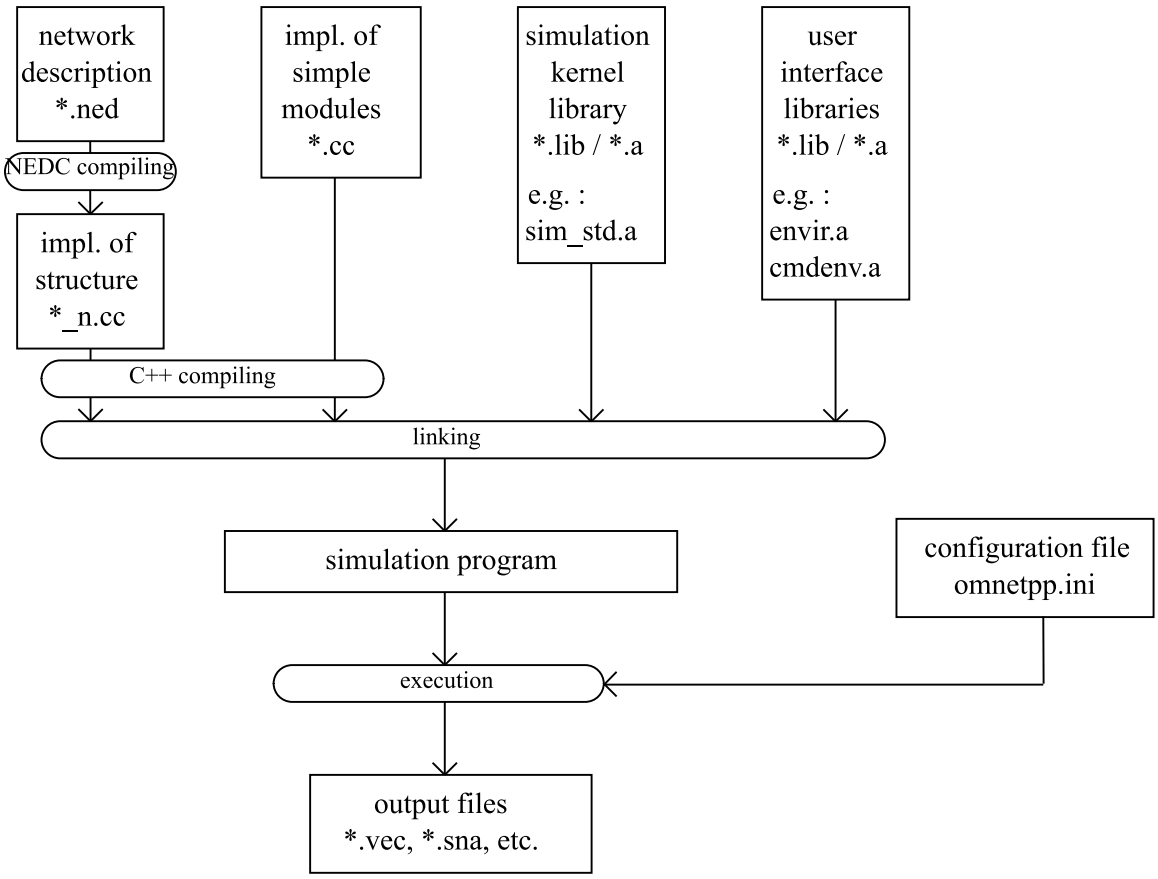

Figure 7.1: usmanFig17 about here.

7.2 Using Unix and gcc

7.2.1 Installation

The installation process depends on what distribution you take (source, precompiled RPM, etc.) and it may change from release to release. The readme files in the distribution should give you enough (and up-to-date) guidance to go through the installation.

7.2.2 Producing a makefile with the opp_makemake script

The opp_makemake script can automatically generate the makefile for your simulation program, based on the source files it finds in your directory. opp_makemake has several options, the following command will display a summary:

```
opp_makemake -h
```

To be able to use opp_makemake, you have to collect all your sources (.ned, .cc, .h files) in one directory. (Large models which spread across several directories are covered later in this section.)

Then type

```
opp_makemake
```

This will create a file named Makefile. Thus if you simply type make, your simulation program should build. The name of the executable will be the same as the name of the directory containing the files.

The freshly generated makefile doesn't contain dependencies, it is advisable to add them by typing make depend. (You'll need a program named makedepend for that, it's present on most Unix systems and in also Cygwin. The warnings during the dependency generation process can be safely ignored.)

In addition to the simulation executable, the makefile contains other targets too. As mentioned, make depend adds (or refreshes) dependencies in the makefile. make clean deletes all files that were produced by the make process. make re-makemake regenerates the makefile using opp_makemake (this is useful if e.g. after upgrading OMNeT++, if opp_makemake has changed). make re-makemake-m is similar to re-makemake, but it regenerates the Makefile.in file too (see later).

If you already had a makefile in that directory, opp_makemake will refuse overwriting it. You can force overwriting the old makefile with the -f option:

```
opp_makemake -f
```

If you have problems, check the path definitions (locations of include files and libraries etc.) in the configure script and correct them if necessary. Then re-run configure to commit the changes to all makefiles, the opp_makemake script etc.

You can specify the user interface (Cmdenv/Tkenv) with the -u option (with no -u, Tkenv is the default):

```
opp_makemake -u Tkenv
```

The name of the output file is set with the -o option (the default is the name of the directory):

```
opp_makemake -o fddi-net
```

If some of your source files are generated from other files (for example, you use machine-generated NED files), write your make rules into a file called makefrag. When you run opp_makemake, it will automatically

insert makefrag into the resulting makefile. With the -i option, you can also name other files to be included into makefile.

If you want better portability for your models, you can generate Makefile.in instead of Makefile with opp_makemake's -m option. You can then use autoconf-like configure scripts to generate the Makefile.

7.2.3 Multi-directory models

In the case of a large project, your source files may be spread across several directories. You have to decide whether you want to use static linking, shared or run-time loaded libraries. Here we discuss static linking.

In each subdirectory (say trafgen/ and router/), run

```
opp_makemake -n
```

The -n option means no linking is necessary, only compiling has to be done.

In your toplevel source directory, run

```
opp_makemake trafgen/ router/
```

This results in recursive makefiles: when you build the simulation, make will descend into trafgen/ and router/, run make in both, then it will link an executable with the object files in the two directories.

You may need to use the -I option if you include files from other directories. The -I option is for both C++ and NED files. In our example, you could run

```
opp_makemake -n -I../router
```

in the trafgen/ directory and vica versa.

If you're willing to play with shared and run-time loaded libraries, several opp_makemake options and the [General]/load-libs= ini file option leave you enough room to do so.

7.2.4 Static vs shared OMNeT++ system libraries

Default linking uses the shared libraries. One reason you would want static linking is that debugging the OMNeT++ class library is more trouble with shared libraries. Another reason might be that you want to run the executable on another machine without having to worry about setting the LD_LIBRARY_PATH variable (which should contain the name of the directory where the OMNeT++ shared libraries are).

If you want static linking, find the

```
build_shared_libs=yes
```

line in the configure.user script and change it to

```
build_shared_libs=no
```

Then you have to re-run the configure script and rebuild everything:

```
./configure
make clean
make
```

7.3 Using Win32 with MSVC

7.3.1 Prerequisite: install Tcl/Tk

Download and install Tcl/Tk. You need at least version 8.0p1, but it's better to download the latest version.

7.3.2 Installing OMNeT++

The installation process is not described here in detail. The readme files in the distribution should give you enough (and up-to-date) guidance to go through the installation.

What's important is that as the result of the installation, you should get the executables and the libraries in the bin/ and lib/ subdirectories within the top-level OMNeT++ directory.

7.3.3 Building the samples from the MSVC IDE

Unfortunately MSVC doesn't like the.cc extension, so first you have to rename the.cc files to.cpp. You can do that with samples/cc2cpp.bat.

Start the MSVC IDE and open the workspace (.dsw) file. Then if you choose Build from the menu, the simulation executable should build. If you encounter any problems, read the MSVC-related readme file in the distribution – it should contain more up-to-date information than this manual.

To change from Tkenv to Cmdenv or vica versa, choose Build|Set active configuration from the menu and select one of 'Debug-Tkenv', 'Release-Tkenv', 'Debug-Cmdenv', 'Release-Cmdenv', then re-link the executable.

If you have big models, you'll probably have to increase the stack size. You'll find the setting under Project|Settings –> 'Link' tab –> choose 'Output' from combo –> Stack allocations, Reserve. Be aware that if you don't specify anything here, MSVC defaults to 1MB – way too small.
If you need to modify the names of the Tcl/Tk libs (because you installed a Tcl/Tk version other than 8.2), see Project|Settings –> 'Link' tab –> choose 'Input' from combo –> Libraries.

The Tcl/Tk install program normally sets the TCL_LIBRARY environment variable needed by Tcl applications. However, if you see the "can't find a usable init.tcl..." error message when you start a simulation program (or Gned or Plove), then that didn't happen and you have to set the variable yourself.

7.3.4 Creating project files for your simulations

1. Start by copying & renaming one of the .dsp files from the samples directory. It already contains the Tkenv/Cmdenv configurations, etc.

2. Rename all .cc files to .cpp (ren *.cc *.cpp) and add them to the project.

3. Add the .ned files to the project and set custom build option for them:

   ```
   Description: NED Compiling $(InputPath)
   Command: nedc -s _n.cpp $(InputPath)
   Outputs: $(InputPath)_n.cpp
   ```

   Hint: you can select all.ned files together, and 'All configurations' from the combo at the left of the Settings dialog, and then you have to type this settings only once.

4. For each.ned file, add a corresponding _n.cpp file.

   Hint: if you compile the.ned files (choose 'Compile' from the menu), the _n.cpp *files will be created, and you can select them all at once in the 'Add files' dialog.*

5. Make sure to turn off exception handling and RTTI (they interfere with the coroutine library), and set the necessary reserved stack size.

6. Note: for Tkenv, link with sim_std.lib, envir.lib, tkenv.lib and the Tcl/Tk libraries (link as Win32 Console app...). For Cmdenv, you need to link with sim_std.lib, envir.lib, and cmdenv.lib.
   It is planned to create wizards in the future to ease some of these steps.

7.3.5 Using Plove

If you want to use Plove, you should download and install Gnuplot. You'll also need a couple of Unix tools like grep and awk, the easiest way to get them is to download and install the Cygwin package from www.cygnus.com. When you have everything installed, start Plove and set the appropriate configuration in Options|External programs. If you entered everything correctly, Plove should work.

A usual caveat is that Gnuplot expects forward slashes in filenames and Plove supplies backslashes or vica versa (there are multiple incompatible builds of Gnuplot on NT); if you suspect this might be the problem, reverse the slash/backslash setting in Options|External programs.

7.4 Hints for using Borland C++ and other compilers

7.4.1 Building OMNeT++

OMNeT++ currently doesn't support the Borland C++. This doesn't mean that the sources won't build (most probably they will), but I am unable to maintain the respective makefiles.

However, the next sections contain some hints how to build simulation programs once you got the libraries compiled.

7.4.2 Setting up a project file

What you will need to have in your project file:

- your simple module C++ sources;
- your NED files;
- for each NED file, the C++ file it will compile into (the _n.cc file). Place the .ned file under the _n.cc file in the project tree hierarchy.
- the OMNeT++ libraries: sim_std.lib, envir.lib, plus cmdenv.lib or tkenv.lib, depending on which user interface you want to link in. You also need the Tcl and Tk libraries if you're using Tkenv.

The project options have to be set up like this:

- Compile as a 32-bit flat console application. None of the special libraries (OWL, MFC, Class Library, OCF etc) are needed.
- You have to turn off exception handling, it conflicts with the coroutine library somehow. In the IDE: Options|Project –> C++ Options –> Exception Handling/RTTI –> clear [ ] Enable exceptions. It must be done both when compiling the libraries and when compiling simulation applications.
- Borland C++ does not recognize the.cc extension as C++. You have to teach it: Options|Tools –> select CppCompile –> Edit –> Advanced –> add the.cc extension to the Translate From and Default For entries. Do the same with the EditText tool.

- You also have to teach Borland C++ how to handle.ned files. Select Options|Tools –> New. Fill in the dialog as follows:

```
Name: NEDCompile

Path: ..\..\src\nedc\nedc.exe

Command Line: $NOSWAP $CAP MSG(BORL2MSG) $EDNAME

Menu Text: NED Compile

Help Hint: OMNeT++ NED compiler
```

  Select Advanced, and fill in the dialog:

```
Type: Translator
Translate From:.ned
Translate To:.cc
Default For:.ned
```

- If you're going to build a LARGE model, be sure to increase the stack size in Options|Project options|Linker|32-bit Linker|Reserved stack size. The default is 0x1000000 (1MB), which is hardly enough for OMNeT++ simulations. Increase it to 64MB for example: 0x40000000. If the simulation exceeds the stack size configured here, you'll get nice exceptions, General Protection Faults and the like.

Chapter 8

Running The Simulation

8.1 Command line switches

An OMNeT++ executable accepts the following command line switches:

-h	The program prints a short help message and the networks contained in the executable and exits.
-f<*fileName*>	Specify the name of the configuration file. The default is omnetpp.ini. Multiple -f switches can be given; this allows you to partition your configuration file. For example, one file can contain your general settings, another one most of the module parameters, another one the module parameters you change often.
-l<*fileName*>	Load a shared object (.so file on Unix). Multiple -l switches are accepted. Your .so files may contain module code etc. By dynamically loading all simple module code and compiled network description (_n.o files on Unix) you can even eliminate the need to re-link the simulation program after each change in a source file. (Shared objects can be created with gcc -shared...)
-r<*runs*>	Only recognized by simulations linked with Cmdenv. It specifies which runs should be executed (e.g. -r2,4,6-8). This option overrides the runs-to-execute= option in the [Cmdenv] section of the ini file (see later).

All other options are read from the configuration file.

An example of running an OMNeT++ executable with the -h flag:

```
C:\OMNETPP\PROJECTS\FDDI>fddi.exe  -h

OMNeT++ Discrete Simulation, TUB Dept. of Telecommunications,  1990-97

Networks in this program:
 1. NRing
 2. FDDI1

End run of OMNeT++
```

8.2 The configuration file: omnetpp.ini

8.2.1 Sections and entries

The configuration file (also called ini file, because it has an .ini extension) contains options that control how the simulation is executed and can also contain settings of model parameters. The ini file is a text file consisting of entries grouped into different sections. The following sections can exist:

```
[General]
[Cmdenv], [Tkenv],...
[Parameters]
[OutVectors]
[DisplayStrings]
[Machines]
[Slaves]
[Run 1], [Run 2], [Run 3],...
```

'#' and ';' denote comments. A sample ini file:

```
# omnetpp.ini

[General]
ini-warnings = false
network = token
distributed = no
snapshot-file = token.sna
output-vector-file = token.vec
log-parchanges = no
parchange-file = token.pch
random-seed = 1
sim-time-limit = 1000ms
cpu-time-limit = 180s
total-stack-kb = 2048

[Cmdenv]
runs-to-execute = 1-3,5
module-messages = yes
verbose-simulation = no
display-update = 100ms

[Parameters]
token.num_stations = 3
token.num_messages = 10000

[Run 1]
token.wait_time = 10ms

[Run 2]
token.wait_time = 30ms
```

Parameters that were set to input value in the NED file are searched for in the ini file.

OMNeT++ can execute several simulation runs automatically one after another. If multiple runs are selected, option settings and parameter values can be given either individually for each run, or together for all runs, depending in which section the option or parameter appears.

This is summarized in the following table:

What	If set for all runs together	If set for individual runs
general settings	`[General]`	`[Run 1]`, `[Run 2]` etc.
user interface-specific settings	`[Cmdenv]`, `[Tkenv]` etc.	`[Run 1]`, `[Run 2]` etc.
module parameter values	`[Parameters]`	`[Run 1]`, `[Run 2]` etc.
output vector configuration	`[OutVectors]`	`[Run 1]`, `[Run 2]` etc.
graphical appearance	`[DisplayStrings]`	`[Run 1]`, `[Run 2]` etc.
logical - physical machine mappings	`[Machines]`	not possible
with distributed execution settings for slave processes	`[Slaves]`	`[Run 1]`, `[Run 2]` etc.

The most important options of the [General] section are the following.

- The ini-warnings option can be used for ”debugging” ini files: if enabled, it lists which options were searched for but not found.
- The network option selects the model to be set up and run.
- The length of the simulation can be set with the sim-time-limit and the cpu-time-limit options (the usual time units such as ms, s, m, h, etc. can be used).
- The warnings option enables/disables run-time warnings; it is recommended to be turned on while debugging.
- The distributed flag selects between normal and parallel execution.
- The output file names can be set with the following options: snapshot-file, output-vector-file, output-scalar-file, parchange-file. (For the last file to be written, one must explicitly enable parameter change logging with the log-parchanges option).
- The load-libs entry can be used to load shared objects (containing simple modules, compiled NED code etc) at run-time. Each setting has a meaningful default value.

Almost any of the above options can also be specified individually for each run. Per-run settings (if they exist) have priority over globally set one.

8.2.2 Splitting up the configuration file

OMNeT++ supports file inclusion in ini files. This feature allows you to partition large ini files to logical units, fixed and varying part etc.

An example:

```
# omnetpp.ini
...
include parameters.ini
include per-run-pars.ini
...
```

8.2.3 Module parameters in the configuration file

Values for module parameters go into the [Parameters] or the [Run 1], [Run 2] etc. sections of the ini file. The run-specific settings take precedence over the overall settings. Parameters that were assigned a (non-input) value in the NED file are not influenced by ini file settings.

Wildcards (*,?) can be used to supply values to several model parameters at a time. Filename-style (glob) and not regex-style pattern matching is used. Character ranges use curly braces instead of square brackets to avoid interference with the notation of module vectors: {a-zA-Z}. If a parameter name matches several wildcards-patterns, the first matching occurrence is used.

An example ini file:

```
# omnetpp.ini

[Parameters]
token.num_stations = 3
token.num_messages = 10000

[Run 1]
token.stations[*].wait_time = 10ms

[Run 2]
token.stations[0].wait_time = 5ms
token.stations[*].wait_time = 1000ms
```

8.2.4 Configuring output vectors

As a simulation program is evolving, it is becoming capable of collecting more and more statistics. The size of output vector files can easily reach a magnitude of several ten or hundred megabytes, but very often, only some of the recorded statistics are interesting to the analyst.

In OMNeT++, you can control how cOutVector objects record data to disk. You can turn output vectors on/off or you can assign a result collection interval. Output vector configuration is given in the [OutVectors] section of the ini file, or in the [Run 1], [Run 2] etc sections individually for each run. By default, all output vectors are turned on.

Entries configuring output vectors can be like that:

```
module-pathname.objectname.enabled = yes/no
module-pathname.objectname.interval = start..stop
module-pathname.objectname.interval = ..stop
module-pathname.objectname.interval = start..
```

The object name is the string passed to cOutVector in its constructor or with the setName() member function.

```
cOutVector eed("End-to-End Delay",1);
```

Start and stop values can be any time specification accepted in NED and config files (e.g. *10h 30m 45.2s*).

As with parameter names, wildcards are allowed in the object names and module path names.

An example:

```
#
# omnetpp.ini
#

[OutVectors]
*.interval = 1s..60s
*.End-to-End Delay.enabled = yes
```

```
*.Router2.*.enabled = yes
*.enabled = no
```

The above configuration limits collection of all output vectors to the 1s..60s interval, and disables collection of output vectors except all end-to-end delays and the ones in any module called Router2.

8.2.5 Module parameter logging

It is possible to log all the changes to module parameters into a text file. This can be useful when the simulation contains run-time tuning of one or more module parameters and one wants to have the trajectory documented.

Module parameter logging must be explicitly enabled from the header file if one wants to use it:

```
[General]
log-parchanges = yes
parchange-file = token.pch
```

The format of the parameter change file is similar to the that of the output vector file.

If a parameter is taken by reference by several modules, any change to the parameter will appear in the file under the name of the top-level parameter, no matter which module actually changed it and under what name.

8.2.6 Display strings

Display strings control the modules' graphical appearance in the Tkenv user interface. Display strings can be assigned to modules, submodules and gates (a connection's display string is stored in its "from" gate). Display strings can be hardcoded into the NED file or specified in the configuration file. (Hardcoded display strings take precedence over the ones given in ini files.) Format of display string are documented in the Display String section (4.9.8).

Display strings can appear in the [DisplayStrings] section of the ini file. They are expected as entries in one of the following forms:

```
\textit{moduletype} = "..."
\textit{moduletype.submodulename} = "..."

\textit{moduletype.inputgatename} = "..."
\textit{moduletype.submodulename.outputgatename} = "..."
```

As with parameter names, wildcards are allowed in module types, submodule and gate names.

8.2.7 Specifying seed values

As is was pointed out earlier, it is of great importance that different simulation runs and different random number sources within one simulation run use non-overlapping sequences of random numbers.

In OMNeT++, you have three choices:

1. Automatic seed selection.
2. Specify seeds in the ini file (with the help of the seedtool program, see later)
3. Manually set the seed from within the program.

If you decide for automatic seed selection, do not specify any seed value in the ini file. For the random number generators, OMNeT++ will automatically select seeds that are 1,000,000 values apart in the sequence. If you have several runs, each run is started with a fresh set of seeds that are 1,000,000 values apart from the seeds used for previous runs. Since the generation of new seed values is costly, OMNeT++ has a table of precalculated seeds (256 values); if they are all used up, OMNeT++ starts from the beginning of the table again.

Automatic seed selection may not be appropriate for you for several reasons. First, you may need more than 256 seeds values; or, if you use variance reduction techniques, you may want to use the same seeds for several simulation runs. In this case, there is a standalone program to generate appropriate seed values (seedtool will be discussed in the next section), and you can specify the seeds explicitly in the ini file.

The following ini file explicitly initializes two of the random number generators, and uses different seed values for each run:

```
[Run 1]
gen0-seed = 1768507984
gen1-seed = 33648008

[Run 2]
gen0-seed = 1082809519
gen1-seed = 703931312
...
```

If you want the same seed values for all runs, you will write something like this:

```
[General]
gen0-seed = 1768507984
gen1-seed = 33648008
```

All other random number generators (2,3,...) will have their seeds automatically assigned. As a third way, you can also set the seed values from the code of a simple module using genk_randseed(), but I see no reason why you would want to do so.

8.2.8 List of all ini file options

The exact meaning of the different entries are:

Entry	Description
[General]	
ini-warnings = yes	Helps debugging of the ini file. If turned on, OMNeT++ prints out the name of the entries it that it wanted to read but they were not in the ini file.
network =	The name of the network to be simulated.
distributed = no	Parallel execution or not.
parallel-system = MPI	MPI or PVM. Defaults to MPI.
snapshot-file = omnetpp.sna	Name of the snapshot file. The result of each snapshot() call will be appended to this file.
output-vector-file = omnetpp.vec	Name of output vector file.
output-scalar-file = omnetpp.sca	Name of output scalar file.
pause-in-sendmsg = no	Only makes sense with step-by-step execution. If enabled, OMNeT++ will split send() calls to two steps.
warnings = yes	Globally turns on/off simulation runtime warnings. It is advisable to leave this turned on.

log-parchanges = no	Specifies whether changes of module parameters should be logged to file. *Not supported after OMNeT++ 2.1.*
parchange-file = omnetpp.pch	File to save parameter changes to. *Not supported after OMNeT++ 2.1.*
sim-time-limit = 1000ms	Duration of the simulation in simulation time.
cpu-time-limit= 180s	Duration of the simulation in real time.
random-seed = 542	Random number seed for generator 0. Should not be zero.
total-stack-kb = 8192	Specifies the total stack size (sum of all coroutine stacks) in kilobytes. You need to increase this value if you get the "Cannot allocate coroutine stack..." error.
load-libs =	Name of shared libraries (.so files) to load after startup. You can use it to load simple module code etc. Example: `load-libs=../x25/x25.so../lapb/lapb.so`
netif-check-freq=	Used with parallel execution.
gen0-seed = 3567 gen1-seed = 4535 ...	Seeds for the given random number generator. They should be nonzero.
outputvectormanager-class= cFileOutputVectorManager	Part of the Envir plugin mechanism: defines the name of the output vector manager class to be used to record data from output vectors. The class has to implement the cOutputVectorManager interface defined in envirext.h.
outputscalarmanager-class= cFileOutputScalarManager	Part of the Envir plugin mechanism: defines the name of the output scalar manager class to be used to record data passed to recordScalar(). The class has to implement the cOutputScalarManager interface defined in envirext.h.
snapshotmanager-class= cFileSnapshotManager	Part of the Envir plugin mechanism: defines the name of the class to handle streams to which snapshot() writes its output. The class has to implement the cSnapshotManager interface defined in envirext.h.

[Cmdenv]	
runs-to-execute=1,3-4,6	Specifies which simulation runs should be executed
module-messages = yes/no	Globally enables/disables ev-style messages in simple modules (e.g. ev << "sending\n";).
verbose-simulation = yes/no	Enables/disables printing banners for each event ("Event #1234, T=..." stuff.)
display-update = 100ms	If there was no display from the simulation execution (both the above options are disabled), OMNeT++ can print out regular messages of the progress. The interval is understood in simulation time.
extra-stack = 16384	Specifies the extra amount of stack (bytes) that is reserved for each *activity()* simple module when the simulation is linked with Cmdenv.

[Tkenv]	
default-run = 1	Specifies which run Tkenv should set up automatically after startup. If there's no default-run= entry or the value is 0, Tkenv will ask which run to set up.
use-mainwindow = yes	Enables/disables writing *ev* output to the Tkenv main window.
print-banners = yes	Enables/disables printing banners for each event.

breakpoints-enabled = yes	Specifies whether the simulation should be stopped at each breakpoint() call in the simple modules.
update-freq-fast = 10	Number of events executed between two display updates when in *Fast* execution mode.
update-freq-express = 500	Number of events executed between two display updates when in *Express* execution mode.
animation-delay = 0.3s	Delay between steps when you slow-execute the simulation.
animation-enabled = yes	Enables/disables message flow animation.
animation-msgnames = yes	Enables/disables displaying message names during message flow animation.
animation-msgcolors = yes	Enables/disables using different colors for each message kind during message flow animation.
animation-speed = 1.0	Specifies the speed of message flow animation.
extra-stack = 32768	Specifies the extra amount of stack (bytes) that is reserved for each *activity()* simple module when the simulation is linked with Tkenv.

[Slaves]	
write-slavelog = yes	Enables/disables writing to the slave.log file
slavelog-file = slave.log	Specifies an alternative filename for slave.log.
module-messages = yes	Specifies whether module messages are printed or not.
errmsgs-to-console = yes	Specifies whether error messages should be sent to and displayed at the 'console' segment.
infomsgs-to-console = no	Specifies whether info messages should be sent to and displayed at the 'console' segment.
modmsgs-to-console = no	Specifies whether module *ev* output should be sent to and displayed at the 'console' segment.

8.3 Choosing good seed values: the seedtool utility

For selecting good seeds, the seedtool program can be used (it is in the utils directory). When started without command-line arguments, the program prints out the following help:

```
seedtool - part of OMNeT++, (c) 1992-2001 Andras Varga, TU Budapest
See the license for distribution terms and warranty disclaimer.

A tool to help select good random number generator seed values.
Usage:
  seedtool i seed         - index of 'seed' in cycle
  seedtool s index        - seed at index 'index' in cycle
  seedtool d seed1 seed2  - distance of 'seed1' and 'seed2' in cycle
  seedtool g seed0 dist   - generate seed 'dist' away from 'seed0'
  seedtool g seed0 dist n - generate 'n' seeds 'dist' apart, starting at
                            'seed0'
  seedtool t              - generate hashtable
  seedtool p              - print out hashtable
```

The last two options, p and t were used internally to generate a hash table of pre-computed seeds that greatly speeds up the tool. For practical use, the g option is the most important. Suppose you have 4 simulation runs that need two independent random number generators each and you want to start their seeds at least 10,000,000 values apart. The first seed value can be simply 1. You would type the following command:

```
C:\OMNETPP\UTILS> seedtool g 1 10000000 8
```

The program outputs 8 numbers that can be used as random number seeds:

```
1768507984
33648008
1082809519
703931312
1856610745
784675296
426676692
1100642647
```

You would specify these seed values in the ini file.

8.4 Repeating or iterating simulation runs

TBD Intro, and multiple simulation runs in omnetpp.ini vs controlling script.

Variations over parameter values

You don't need to generate the whole omnetpp.ini from program if you use include files. You can have a fixed omnetpp.ini which contains the line

```
include parameters.ini
```

and then generate parameters.ini by program for each run.

Here's the "runall" script of Joel Sherrill's *File System Simulator* as an example:

```
#! /bin/bash
#
# This script runs multiple variations of the file system simulator.
#
all_cache_managers="NoCache FIFOCache LRUCache PriorityLRUCache..."
all_schedulers="FIFOScheduler SSTFScheduler CScanScheduler..."

for c in ${all_cache_managers}; do
  for s in ${all_schedulers}; do
  (
    echo "[Parameters]"
    echo "filesystem.generator_type = \"GenerateFromFile\""
    echo "filesystem.iolibrary_type = \"PassThroughIOLibrary\""
    echo "filesystem.syscalliface_type = \"PassThroughSysCallIface\""
    echo "filesystem.filesystem_type = \"PassThroughFileSystem\""
    echo "filesystem.cache_type = \"${c}\""
    echo "filesystem.blocktranslator_type = \"NoTranslation\""
    echo "filesystem.diskscheduler_type = \"${s}\""
    echo "filesystem.accessmanager_type = \"MutexAccessManager\""
    echo "filesystem.physicaldisk_type = \"HP97560Disk\""
  ) >algorithms.ini

  ./filesystem
  done
done
```

And omnetpp.ini includes algorithms.ini.

Variations over seed value (multiple independent runs)

The same technique can be used if you want several runs with different random seeds. This code should do 500 runs with independent seeds (suppose one run doesn't use more than 10 million random values):

```
#! /bin/bash

for seed in ‘seedtool g 1 10000000 500‘
do
   (
     echo "[General]"
     echo "random-seed = ${seed}"
     echo "output-vector-file = xcube-${seed}.vec"
   ) > parameters.ini
   ./xcube
done
```

omnetpp.ini should include parameters.ini.

Other languages for writing the control script

The above examples use the Unix shell, but you have quite a number of options in what language to implement the controlling script. Some ideas:

- shell (mentioned above)
- Perl
- Tcl
- Octave (suggested by Richard Lyon, see the contrib/octave directory for examples)
- DOS/Win32 batch (maybe this is not such a good idea...)

8.5 User interfaces of simulation executables

The user interface is separated from the simulation kernel; the two parts interact through a well-defined interface. This construction makes it possible to implement several types of user interfaces, without changing the simulation kernel. Also, the same simulation model can be executed with different user interfaces, without any change in the model files themselves. The user would test and debug the simulation with a powerful graphical user interface, and finally run it with a simple and fast user interface that supports batch execution.

User interfaces takes the form of libraries (.a file or .so on UNIX, .lib or .dll file on NT). The libraries are interchangeable. When the user creates a simulation executable, he can pick one of the user interface libraries that he links in.

Two user interfaces have been implemented:

- Cmdenv: command-line user interface for batch execution
- Tkenv: graphical, windowing user interface (Tcl/Tk)

The following sections contain more detailed descriptions about each user interface.

8.5.1 Cmdenv: the command-line user interface

The command line user interface is a small, portable and fast user interface that compiles and runs on all platforms whether it is UNIX, DOS, or WinNT console. Cmdenv is designed primarily for batch execution.

Cmdenv uses simply executes all simulation runs that are described in the configuration file. If one run stops with an error message, subsequent ones will still be executed.

Cmdenv recognizes the following ini file options:

```
[Cmdenv]
runs-to-execute = 1,4-6,8
module-messages = no
verbose-simulation = no
display-update = 100ms
```

The first one specifies which runs (described in the [Run 1], [Run 2] etc. sections) should be executed. If the value is missing, Cmdenv executes all runs that have ini file sections; if no runs are specified in the ini file, Cmdenv does one run. The -r command line option overrides this ini file setting.

The second and the third are yes/no settings and control the amount of screen output during simulation. The fourth one is in effect when the other two are disabled (that is, there would be no display at all from the simulation execution); it prints out progress messages at the specified frequency.

Portability: all platforms.

8.5.2 Tkenv: graphical user interface on Unix/NT

Features

Tkenv is a portable graphical windowing user interface. Tkenv supports interactive execution of the simulation, tracing and debugging. Tkenv is recommended in the development stage of a simulation or for presentation and educational purposes, since it allows one to get a detailed picture of the state of simulation at any point of execution and to follow what happens inside the network. The most important feaures are:

- message flow animation
- graphical display of statistics (histograms etc.) and output vectors during simulation execution
- separate window for each module's text output
- scheduled messages can be watched in a window as simulation progresses
- event-by-event, normal and fast execution
- labeled breakpoints
- inspector windows to examine and alter objects and variables in the model
- simulation can be restarted
- snapshots (detailed report about the model: objects, variables etc.)

Tkenv makes it possible to view simulation results (output vectors etc.) during execution. Results can be displayed as histograms and time-series diagrams. This can speed up the process of verifying the correct operation of the simulation program and provides a good environment for experimenting with the model during execution. When used together with gdb or xxgdb, Tkenv can speed up debugging a lot.

Portability: Tkenv is built with Tcl/Tk. Tkenv should work on all platforms that Tcl/Tk has been ported to: Unix/X, Win32, Macintosh.

You can get more information about Tcl/Tk in the Web pages listed in the Reference.

Simulation running modes in Tkenv

Tkenv has the following modes for running the simulation :

- Step
- Run
- Fast run
- Express run

The running modes have their corresponding buttons on Tkenv's toolbar.

In **Step** mode, you can execute the simulation event-by-event.

In **Run** mode, the simulation runs with all tracing aids on. Message animation is active and inspector windows are updated after each event. Output messages are displayed in the main window and module output windows. You can stop the simulation with the Stop button on the toolbar. You can fully interact with the user interface while the simulation is running: you can open inspectors etc.

In **Fast** mode, animation is turned off. The inspectors and the message output windows are updated after each 10 events (the actual number can be set in Options|Simulation options and also in the ini file). Fast mode is several times faster than the Run mode; the speedup can get close to 10 (or the configured event count).

In **Express** mode, the simulation runs at about the same speed as with Cmdenv, all tracing disabled. Module output is not recorded in the output windows any more. You can interact with the simulation only once in a while (1000 events is the default as I recall), thus the run-time overhead of the user interface is minimal. You have to explicitly push the Update inspectors button if you want an update.

Inspectors

In Tkenv, objects can be viewed through inspectors. To start, choose Inspect|Network from the menu. Usage should be obvious; just use double-clicks and popup menus that are brought up by right-clicking. In Step, Run and Fast Run modes, inspectors are updated automatically as the simulation progresses. To make ordinary variables (int, double, char etc.) appear in Tkenv, use the WATCH() macro in the C++ code.

In list dialogs, entries begin with text like "ptr0x8000ab7e". Yes, it is really the object pointer; knowing it is extremely useful if you're running the simulation under a debugger such as gdb.

Configuring Tkenv

In case of nonstandard installation, it may be necessary to set the OMNETPP_TKENV_DIR environment variable so that Tkenv can find its parts written in Tcl script.

The default path from where the icons are loaded can be changed with the OMNETPP_BITMAP_PATH variable, which is a semicolon-separated list of directories and defaults to "*omnetpp-dir*/bitmaps;.;./bitmaps".

The ini file options accepted by Tkenv are:

```
[Tkenv]
use-mainwindow = yes
print-banners = yes
breakpoints-enabled = yes
update-freq-fast = 10
update-freq-express = 500
animation-delay = 0.3s
```

The above options can also be set from within Tkenv itself, from a configuration dialog box.

Embedding TCL code into the executable

A significant part of Tkenv is written in TCL, in several .tcl script files. The default location of the scripts is passed compile-time to tkapp.cc, and it can be overridden at run-time by the OMNETPP_TKENV_DIR environment variable. The existence of a separate script library can be inconvenient if you want to carry standalone simulation executables to different machines. To solve the problem, there is a possibility to compile the script parts into Tkenv as a large string constant.

The details: the tcl2c program (its C source is there in the Tkenv directory) is used to translate the .tcl files into C code (tclcode.cc), which gets included into tkapp.cc. On Unix, this feature is enabled in Tkenv's makefile; it is documented there exactly how. On Win95/NT, one has to manually compile tcl2c.c into tcl2c.exe, run it to produce tclcode.cc and then compile tkapp.cc without providing the OMNETPP_TKENV_DIR external define. The latter will cause tkapp.cc to include and use tclcode.cc.

8.5.3 In Memoriam...

There used to be other user interfaces which have been removed from the distribution.

- **TVEnv**. A Turbo Vision-based user interface, the first interactive UI for OMNeT++. (Turbo Vision was an excellent character-graphical windowing environment, originally shipped with Borland C++ 3.1.)
- **XEnv**. A GUI written in pure X/Motif. It was an experiment, written before I stumbled into Tcl/Tk and discovered its immense productivity in GUI building. XEnv never got too far because it was really very-very slow to program in Motif...

8.6 Typical problems

8.6.1 Stack problems

"Stack violation (*FooModule* stack too small?) in module *bar.foo*"

OMNeT++ detected that the module has used more stack space than it has allocated. You should increase the stack for FooModule. You can call the stackUsage() from finish() to find out actually how much stack the module used.

"Error: Cannot allocate *nn* bytes stack for module *foo.bar*"

If you get the above message, you have to increase the total stack size (the sum of all coroutine stacks). You can do so in omnetpp.ini:

```
[General]
total-stack-kb = 2048 # 2MB
```

There is no penalty if you set total-stack-kb too high. I recommend to set it to a few K less than the maximum process stack size allowed by the operating system (ulimit -s; see next section).

"Segmentation fault"

On Unix, if you set the total stack size higher, you may get a segmentation fault during network setup (or during execution if you use dynamically created modules) for exceeding the operating system limit for maximum stack size. For example, in Linux 2.0.x, the stack can be at most 8192K (that is, 8MB). The ulimit syscall and utility program can be used to modify the resource limits, but you can only increase if you're root. Furthermore, resource limits are inherited by child processes. The following statement worked out for me under Linux to get a shell with a 64M stack limit:

```
$ su root
Password:
# ulimit -s 65536
# su andras
$ ulimit -s
65536
```

If you do not want to go through the above process at each login, you can change the limit in the PAM configuration files. In Redhat Linux (maybe other systems too), add the following line to `/etc/pam.d/login`:

```
session    required    /lib/security/pam_limits.so
```

and the following line to `/etc/security/limits.conf`:

```
*    hard    stack    65536
```

A more drastic solution is to recompile the kernel with a larger stack limit. Edit `/usr/src/linux/include/linux/sched.h` and increase `_STK_LIM` from `(8*1024*1024)` to `(64*1024*1024)`.

Finally, it you're tight with memory, you can switch to Cmdenv. Tkenv increases the stack size of each module by about 32K so that user interface code that is called from a simple module's context can be safely executed. Cmdenv does not need that much extra stack.

8.6.2 Memory allocation problems

For investigating memory allocation problems, try using Cmdenv, and uncomment the `#defines` in `src/envir/cmdenv/heap.cc`:

`HEAPCHECK`	checks heap on new/delete
`COUNTBLOCKS`	counts blocks on heap and tells it if none left
`ALLOCTABLE`	remembers pointers and reports heap contents if only LASTN blocks remained
`DISPLAYALL`	reports every new/delete
`DISPSTRAYS`	reports deleting of pointers that were not registered by operator new or that were deleted since then
`BKPT`	calls a function at a specified new/delete; you can set a breakpoint to that function

If COUNTBLOCKS is turned on, you should see the [heap.cc-DEBUG:ALL BLOCKS FREED OK] message at the end of the simulation. If you do not see it, it means that some blocks have not been freed up properly, that is, your simulation program is likely to have memory leaks.

8.7 Execution speed

If your simulation program is tested and runs OK, you'll probably want to run it as fast as possible. Here's a table that could help where to begin optimizing.

The measurements were made on one version of the FDDI model (you can find it in the samples directory); we simulated 10 milliseconds. We used Cmdenv. The machine was a 100Mhz Intel Pentium with 32MB RAM. The simulation program was compiled with Borland C++ 3.1 (no particular optimization) and run on DOS 6.22. Disk caching was installed (SmartDrive read/write caching, 8MB cache).

Settings	Execution time	Details

all screen output on;
full heapcheck
	7 min 50 sec	Setting in omnetpp.ini:
		`verbose-simulation = yes`
		`module-messages = yes`
		The `#defines` in `envir/cmdenv/heap.cc` were all enabled. This means full heapcheck with each allocation, tracking of all allocated blocks etc.

no screen output at all; full heapcheck	5 min 50 sec	All screen output were #ifdef'ed out from source; also, the omnetpp.ini contained the `verbose-simulation = no` line. The heapcheck defines were turned on.
all screen output on; no heapcheck	2 min	We turned off heapcheck (we commented out the defines in heap.cc) and turned back on the screen output. We used the same omnetpp.ini: setting as with first case.
screen output redirected to file; no heapcheck	15.5 sec	Same as previous configuration, except that we run the program with `fddi > output.txt`
screen output redirected to NUL; no heapcheck	13 sec	Same as previous configuration, except that we run the program with `fddi > NUL`

screen output turned off from ini file;
no heapcheck
	7.5 sec	We did not only redirect but also disabled screen output. Setting in omnetpp.ini:
		`verbose-simulation = no`
		`module-messages = no`

no screen output generation; no heapcheck	4.5 sec	We #ifdef'ed out all printouts from the simple module sources and also turned off any messages from omnetpp.ini.

The moral is that heap checks and screen output greatly influences speed, so once you do not need them (debugging is over), throw them out. You also gain a lot by putting #ifdef lines around your debugging code. And of course, program with care.

Chapter 9

Analyzing Simulation Results

9.1 Plotting output vectors with Plove

9.1.1 Plove features

Typically, you'll get output vector files as a result of a simulation. Data written to cOutVector objects from simple modules go to output vector files. Normally, you use Plove to look into output vector files and plot vectors in it.

Plove is a handy tool for plotting OMNeT++ output vectors. It uses Gnuplot to do the actual work. You can specify the drawing style (lines, dots etc) for each vector as well as set the most frequent drawing options like axis bounds, scaling, titles and labels etc. You can save the gnuplot graphs to files (postscript, latex, pbm etc) with a click. Plove can also generate standalone shell scripts that plot output vectors in much the same way Plove does itself. These scripts can be used for batch processing or to debug filters (see later). Plove does not take away any of gnuplot's flexibility – you can embed your own gnuplot commands to customize the output.

Filtering the results before plotting is possible. Filters can do averaging, truncation of extreme values, smoothing, they can do density estimation by calculating histograms etc. Some filters are built in, and you can easily create new filters or modify the existing ones. Filters can be incorporated in one of three ways: as awk expressions, as awk programs and as external filter programs. Filters can be parameterized. Multiple filters for the same vector is not currently supported; also, you cannot currently feed several vectors into a single filter.

Plove does not create temporary files, so you don't need to worry about disk space: if the output vector is there, Plove can plot it for you. Moreover, it can also work with gzipped vector files without extracting them – just make sure you have zcat.

Plove never modifies the output vector files themselves.

On startup, Plove automatically reads the .ploverc file in your home directory. The file contains general gnuplot settings, the filter configuration etc. (that is, the stuff from the Options menu).

Portability: Plove works fine on Unix and (with some limitations) on Win95/NT.

9.1.2 Usage

First, you load an output vector file (.vec) into the left pane. You can also load gzipped vector files (.vec.gz) without having to decompress them. You can copy vectors from the left pane to the right pane by clicking the right arrow icon in the middle. The large PLOT button will plot the *selected* vectors in the right pane. Selection works as in Windows: dragging and shift+left click selects a range, and ctrl+left click

selects/deselects individual items. To adjust drawing style, change vector title or add filter, push the Options... button. This works for several selected vectors too. Plove accepts nc/mc-like keystrokes: F3, F4, F5, F6, F8, grey '+' and grey '*'.

The left pane works as a general storage for vectors you're working with. You can load several vector files, delete vectors you don't want to deal with, rename them etc. All this will not affect the vector files on disk. In the right pane, you can duplicate vectors if you want to filter the vector and also keep the original. If you set the right options for a vector but temporarily do not want it to hang around in the right pane, you can put it back into the left pane for storage.

9.1.3 Writing filters

Filters get an output vector on their standard input (as plain text, with the timestamp being the second and the value being the third field on each line), do some processing to it and write the result to the standard output.

Filters can be incorporated in one of three ways: as awk expressions, as awk programs or as external programs. An 'awk expression' filter means assembling and launching a command like this:

```
cat foobar.vec | awk '{$3 = <expression>; print}' | ...
```

An awk program filter means running the following command:

```
cat foobar.vec | awk '{<program>}' | ...
```

The third type of filters is used like this:

```
cat foobar.vec | <program> <parameters> | ...
```

Before the filter pipeline is launched, the following substitutions are performed on the awk scripts:

```
t --> $2
x --> $3
```

The parameters of the form `$(paramname)` are also replaced with their actual value.

Thus, if you want to add 1 to all value, you can use the awk expression filter x+1. It will turn into:

awk '{$3 = $3+1}; print'.

When you want to shift the vector by a used-defined DT time, you can create the following awk program filter:

```
{t += $(DT); print}
```

Do not forget the print statement, or your filter will not output anything and the gnuplot graph will be empty.

Filters are automatically saved into and loaded from the `~/.ploverc` file.

TBD add example scripts

9.2 Format of output vector files

An output vector file contains several series of data produced during simulation. The file is textual, it looks like this:

```
mysim.vec:
vector 1   "subnet[4].term[12]"  "response time"  1
1  12.895  2355.66666666
1  14.126  4577.66664666
vector 2   "subnet[4].srvr"  "queuelen+queuingtime"  2
2  16.960  2.00000000000.63663666
1  23.086  2355.66666666
2  24.026  8.00000000000.44766536
```

There are label lines (beginning with vector) and data lines.

A vector line introduces a new vector. Its columns are: vector ID, module of creation, name of cOutVector object, multiplicity of data (single numbers or pairs will be written).

Lines beginning with numbers are data lines. The columns: vector ID, current simulation time, and one or two double values.

9.3 Working without Plove

9.3.1 Extracting vectors from the file

You can use the Unix grep tool to extract a particular vector from the file. As the first step, you must find out the ID of the vector. You can find the appropriate vector line with a text editor or you can use grep for this purpose:

```
% grep "queuelen+queuingtime" vector.vec
```

Or, you can get the list of all vectors in the file by typing:

```
% grep ^vector vector.vec
```

This will output the appropriate vector line:

```
vector 6  "subnet[4].srvr"  "queuelen+queuingtime"  2
```

Pick the vector ID, which is 6 in this case, and grep the file for the vector's data lines:

```
grep ^6 vector.vec > vector6.vec
```

Now, vector6.vec contains the appropriate vector. The only potential problem is that the vector ID is there at the beginning of each line and this may be hard to explain to some programs that you use for post-processing and/or visualization. This problem is eliminated by the OMNeT++ splitvec utility (written in awk), to be discussed in the next section.

9.3.2 Using splitvec

The splitvec script (part of OMNeT++) breaks the vector file into several files which contain one vector each:

```
% splitvec mysim.vec
```

creates several files: mysim1.vec, mysim2.vec etc.

```
mysim1.vec:
# vector 1  "subnet[4].term[12]"  "response time"  1
12.895  2355.66666666
14.126  4577.66664666
23.086  2355.66666666

mysim2.vec:
# vector 2  "subnet[4].srvr"  "queuelen+queuingtime"  2
16.960  2.00000000000.63663666
24.026  8.00000000000.44766536
```

As you can see, the vector ID is gone.

The files can be further processed with math packages, or read by analysis or spreadsheet programs which provide numerous ways to display data as diagrams, do calculations on them etc. One could use for example Gnuplot, Matlab, Excel, etc.

9.3.3 Visualization under Unix

Two programs are in common use: Gnuplot and Xmgr. Both are free and both have their good and bad sides; will briefly discuss them. There are innumerable tutorials and documentation about them on the Web; some of them you will find among the References.

Both programs can eat files produced by splitvec. Both programs can produce output in various forms: on screen, in Postscript format, printer files, Latex output etc. For DTP purposes, Postscript seems to be the most appropriate. On Windows, the easiest way is to copy the picture to the clipboard from the Gnuplot window's system menu.

Gnuplot has an interactive command interface. To get the vectors in mysim1.vec and mysim4.vec plotted in the same graph, you can type:

```
plot "mysim1.vec" with lines, "mysim4.vec" with lines
```

To adjust the y range, you would type:

```
set yrange [0:1.2]
replot
```

There are several commands to adjust ranges, plotting style, labels, scaling etc. Gnuplot can also plot 3D graphs. Gnuplot is also available for DOS, Windows and other platforms. Gnuplot also has a simple graphical interactive user interface called PlotMTV. However, we recommend that you use OMNeT++'s Plove tool, described in an earlier section.

Xmgr is an X/Motif based program, with a menu-driven graphical interface. You load the appropriate file by selecting in a dialog box. The icon bar and menu commands can be used to customise the graph. Some say that Xmgr can produce nicer output that Gnuplot and it is easier to use. Xmgr cannot do 3D and only runs on Unixes with X and Motif installed. Xmgr also has a batch interface so you can use it from scripts too.

Chapter 10

Parallel Execution

10.1 OMNeT++ support for parallel execution

10.1.1 Introduction to Parallel Discrete Event Simulation

OMNeT++ supports parallel execution of large simulations. The following paragraphs provide a very brief (and thus not very accurate) picture of the problems and methods of parallel discrete event simulation (PDES). Interested readers – and those who are thinking about doing PDES with OMNeT++ – are strongly encouraged to look into the literature.

For parallel execution, the model is to be partitioned to several segments that will be simulated independently on different hosts or processors. Each segment will have its own local Future Event Set, thus they will maintain local simulation times. The main issue with parallel simulations is keeping segments synchronized in order to avoid violating causality of events. Without synchronization, a message sent by one segment could arrive in another segment when the simulation time in the receiving segment has already passed the timestamp (arrival time) of the message. This would break causality of events in the receiving segment.

There are mainly three different methods used for synchronizing segments:

1. **Conservative synchronization** exploits knowledge about when segments send messages to other segments, and uses 'null' messages to propagate this info to other segments. This may speed up simulation, since e.g. if a segment knows it won't receive any messages from other segments until $t + \Delta t$ simulation time, it may advance until $t + \Delta t$ without the need for external synchronization. Conservative syncronization requires modifications to existing models, i.e., inserting code which sends out the 'null' messages. Conservative simulation tends to converge to sequential simulation (slowed down by communication between segments) if there's not enough parallelism in the model, or parallelism is not exploited by sending enough 'null' messages.

2. **Optimistic synchronization** allows incausalities to occur, but detects and repairs them. Repairing involves rollbacks to a previous state, sending out anti-messages to cancel messages sent out during the period that is being rolled back, etc. Optimistic synchronization is extremely difficult to implement, because it requires periodic state saving and the ability to restore previous states. In any case, implementing optimistic synchronization in OMNeT++ would require – in addition to a more complicated simulation kernel – writing significantly more complex simple module code from the user. Optimistic synchronization may be slow in cases of excessive rollbacks.

3. **Statistical synchronization** is a compromise where segments do not exchange individual messages but distributions of the traffic flow characteristics. While conservative and optimistic synchronization are exact methods (they produce exactly the same results are the corresponding sequential simulation

would), this is certainly not true for statistical synchronization where the results may contain error introduced by the statistical nature of the synchronization. Statistical synchronization does not require changes to existing models, only the insertion of extra modules, called *"statistical interfaces"*, therefore it is significantly easier to implement than either conservative or optimistic. In addition to easier implementation, there is a potential for much larger speedup than with conservative or optimistic, because the method is much less sensitive to communication delay between processors running the segments. Therefore, for parallel simulation on a cluster of workstations, statistical synchronisation may be the only feasible method.

10.1.2 OMNeT++ support for parallel simulation

The simulation kernel makes it possible to send messages from one segment to another. A message can contain arbitrarily complex data structures; these are transferred transparently, even between hosts of different architectures. The simulation kernel provides a simple synchronization mechanism (*syncpoints*, available through the syncpoint() call) that can ensure that causality is kept when sending messages between segments. Syncpoints correspond to *null messages* found in the literature.

Message sending and syncpoints enable one to implement conservative PDES and also Statistical Synchronization. The simulation class library contains objects that explicitly support the implementation of models using Statistical Synchronization.

High level debugging is supported by saving the textual output from remote segments to a log file and/or relaying them to a single console.

OMNeT++ supports flexible partitioning of the model. In the NED language, by using *machine parameters* you can specify *logical hosts* for different modules at any level of the module hierarchy of the network. You map logical hosts to physical ones in the ini file; if you map several logical hosts into the same physical machine, they will be merged into a single OMNeT++ process.

One may choose between using the MPI (Message Passing Interface) and the PVM3 (Parallel Virtual Machine Version 3) libraries for communication between hosts. Both libraries are portable and widely used in university and research environments. MPI is newer though and considered to be the successor of PVM. You can find MPI and PVM readings in the Reference.

10.1.3 Syncpoints

Overview

When running a simulation in parallel, different segments of the model execute as independent UNIX processes, typically on separate hosts. Since the hosts can be of different speed and the simulated model segments can be of different complexity, at a given moment the model times of different segments will differ: some segments are ahead of the others and some lag behind. Suppose that a message is sent from segment A to segment B which is ahead of A in model time. If B processed the message, causality would break. This should never happen.

The solution built in OMNeT++ is the following. Segment A must know in advance when it will send the next message to segment B and announce it with the `syncpoint()` call. The simulation kernel sends the syncpoint to segment B. When segment B's model time reaches the specified time, segment B's simulation kernel blocks execution until the promised message arrives from A. Then the simulation continues, typically but not necessarily with the message that has just been received from A.

In the reverse case when A is ahead of B, A's message arrived at B before it has reached the syncpoint. In this case, there is no problem and the syncpoint is just an unnecessary precaution. B just inserts the message in its future event set, clears the syncpoint and continues execution.

The syncpoint API

The syncpoint() call takes two arguments. The first is the model time when (or more precisely: when of

after when) the simple module will send a message to another simple module in a different segment. The second argument is a gate given with its number or its name. The gate implicitly specifies the destination segment to synchronize with.

```
syncpoint(t, "outgate");
```

Details of the syncpoint implementation

If the destination module is in the same segment, the call is ignored. (This makes it possible to run models designed to execute in parallel as a single process, without any modification.) Each segment keeps a list of syncpoints sent to it (time + gate), ordered by time. Simulation executes normally until it comes to an event that has a time *definitely past* the first syncpoint in the list. That event is not processed, but the segment goes into a blocked state. While the segment is blocked, it listens for messages arriving from other segments. (In the actual implementation, passive wait is used so a blocked segment doesn't use much CPU time.) Each message that arrives deletes the first syncpoint in the list that matches its gate. The segment goes out of the blocked state when – because of deletions – the first syncpoint in the list is no longer past the event in question. Then the simulation goes on normally, either with the newly arrived message (or the earliest of them) or the original event. A message that arrives outside of the blocked state also causes deletion of the first matching syncpoint in the list; this case corresponds to the reverse case when the sender segment is ahead of the receiving segment in model time.

Deadlock

It is possible to cause deadlock with carelessly placed syncpoints. Suppose that segment A declares a syncpoint at 10s with segment B, but it will actually send a message only at 10.5s. If segment B does the same to segment A, a nice deadlock is created. OMNeT++ makes no effort to detect or prevent such deadlocks; it is entirely the simulation programmer's task to take care that deadlocks do not occur.

10.2 Configuring a simulation for parallel execution

10.2.1 Configuring OMNeT++

Choosing between MPI and PVM

You have let OMNeT++ know if you want to use MPI or PVM. This can be configured in the [General] section of the ini file, via te parallel-system= entry. Its value can be "PVM" or "MPI"; it defaults to "MPI".

```
; file: omnetpp.ini
;...
[General]
parallel-system = MPI
```

Mapping logical machines to physical ones

The on: phrases in the NED descriptions specify the logical machine(s) on which the module is run. The machine parameters are mapped to physical machines in the [Machines] section of the configuration file:

```
; file: omnetpp.ini
;...
[Machines]
node1 = whale.hit.bme.hu
node4 = whale.hit.bme.hu
node2 = puppis.hit.bme.hu
```

```
node3 = dolphin.hit.bme.hu
;...
```

Configuration of the slaves

Slave processes can be configured in the [Slaves] section of the configuration file:

```
; file: omnetpp.ini
[Slaves]
write-slavelog=
slavelog-file=
module-messages=
errmsgs-to-console=
infomsgs-to-console=
modmsgs-to-console=
```

Screen input/output of the slaves is re-routed to the console. However, any file I/O is done in the local file system of each host.

10.2.2 Setting up PVM

The PVM virtual machine

The `pvmhosts` file is used by PVM to describe what computers will participate in the virtual machine, where the executables (in our case, the OMNeT++ programs) are located on each computer, what working directories should be set etc.

It is advisable to have a common, shared directory mounted on all participating hosts; this eliminates the tedious work of having to copy files to all hosts again and again.

If using OMNeT++, it is a good idea to write separate pvmhosts files for each simulation program. Since simulation programs are typically in separate directories, the pvmhosts file in each directory can name that directory as executables directory and working directory for each host. This way, there is no need to create soft links or explicitly name directories in the OMNeT++ ini files.

Each line in the pvmhosts file describes one host. An example line (this all should be a *single line*!):

```
whale ip=whale.hit.bme.hu lo=andras
   dx=/home/andras/pvm/pvm
   ep=/home/andras/omnetpp/projects/fddi
   wd=/home/andras/omnetpp/projects/fddi
```

To start PVM with this configuration:

```
cd ~/omnetpp/projects/fddi
pvm pvmhosts
```

Configuration and running

The user must have PVM installed on the hosts he is going to run segments on.

To set up a simulation for distributed execution, the user must:

1. set the `PVM_ROOT` environment variable

2. link the simulation executable with `sim_pvm` instead of `sim_std` (You can do it by setting `PVM_SUPPORT` to `yes` in a `opp_makemake`-generated makefile.)
3. set `distributed=true` in the `[General]` section of the configuration file.
4. specify the logical-hosts-to-physical-machines mapping in the `[Machines]` section
5. copy the simulation executable and the configuration file to each host if they have physically different disks
6. start `pvm` with an appropriate `pvmhosts` file
7. start the simulation executable on the host which is supposed to be the console. That process will start up the program on the other hosts too and do the simulation.

The first machine is called ”console” or “master”, the others are called “slaves”.

If there are problems...

PVM programs in general are more difficult to get running than ordinary programs. Wrong settings in the PVM configuration files can cause various problems, for example. Also, parallel programs are a lot harder to test and debug.

What can you do if your distributed OMNeT++ simulation won’t work?

- First of all, check the `pvmhosts` file to see if PVM looks for the executables in the right directories on all hosts and the working directories are right (typically, the same directory as the executable’s).
- In the ini file, enable writing the `slave.log` files for the slave processes and check what is written into them.
- You can try enabling the `SINGLE_HOST` define in the `sim/pvm/pvmmod.cc` source file. This will make OMNeT++ run all segments of the distributed simulation on the local host, making things a lot easier to manage.
- Also, try the defining `PVM_DEBUG` at the same place: it enables a lot of `ev.printf()`s in the code interfacing with PVM, so it is easier to spot where the problems are.
- PVM itself also has an environment variable which, if set, causes the PVM library to print out debugging information. However, this is very low-level information, it will rarely be useful.

10.2.3 Setting up MPI

TBD...

10.3 Statistical synchronization

10.3.1 The description of the Statistical Synchronization Method (SSM)

Similarly to other parallel discrete event simulation methods, the model to be simulated - which is more or less a precise representation of a real system - is divided into segments, where the segments usually describe the behaviour of functional units of the real system. The communication of the segments can be represented by sending and receiving various messages. The simulators of the segments are executed by separate processors.

The communication of these segments is simulated with appropriate interfaces. The messages generated in a given segment and to be processed in a different segment are not transmitted there, but the output interfaces collect the statistical data of them. If the input interfaces generate messages for the segments according to the statistical characteristics of the messages collected by the proper output interfaces, the segments with their input- and output interfaces can be simulated separately, giving statistically correct results. The events in one segment have not the same effect in other segments as in the original model, so the results collected during the SSM are not exact. The precision depends on the segmentation, on the accuracy of statistics collection and regeneration, and on the frequency of the statistics exchange among the processors.

Segmentation

The segments of the simulator are executed by separate processors, they have their own, independent virtual times. Because the interactions among segments are performed by the statistical parameters of these interactions, the segmentation should be done so, that the overwhelming majority of the interactions should happen within the segments and not among them. This speeds up the so-called inter-segment transients and improves the accuracy as well.

Timing of statistics exchange

Asynchronous statistics exchange means, that whenever a statistical result collection in an output interface is ready, it is applied - after mapping and correction - in the proper input interface. This is clearly more efficient, than the so-called synchronous statistics exchange, which means, that we delay the application of collected values until all the output interfaces get ready with the result collection. Frequent statistics exchange makes the inter-segment transient faster, but the lower sample numbers makes the estimation - and the whole simulation - less precise.

To learn more about SSM, see [PON92] and [PON93].

10.3.2 Using SSM in OMNeT++

OMNeT++ directly supports the implementation of statistical interface with the following classes:

`cLongHistogram, cDoubleHistogram, cPSquare, cPar.`

Chapter 11

The Design of OMNeT++

11.1 Structure of an OMNeT++ executable

Consider the following diagram:

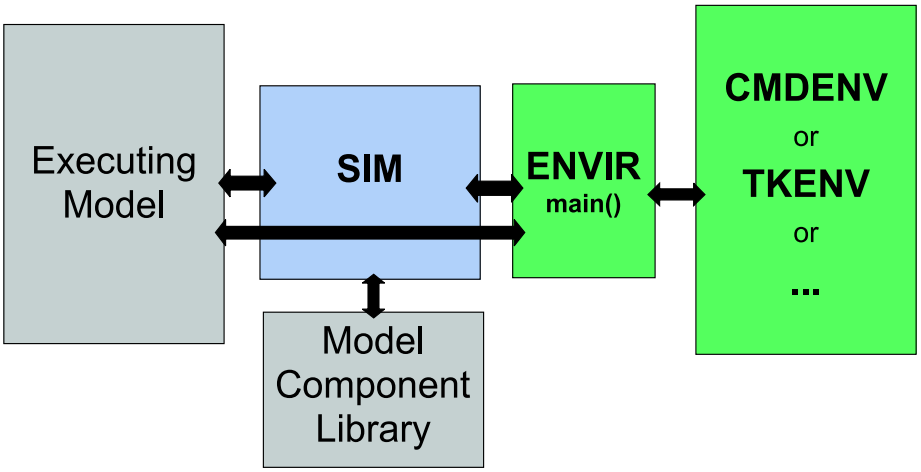

Figure 11.1: Architecture of OMNeT++ simulation programs

A simulation program contains the simulated network (with its simple and compound modules etc.), SIM, ENVIR and exactly one of CMDENV and TKENV. SIM contains the simulation class library and the simulation kernel. The model only interacts with SIM[1]. ENVIR contains code that's common for all user interfaces, and provides infrastructure like ini file handling for them. `main()` is also in ENVIR. Specific user interface code is contained in CMDENV and TKENV. The above components are also physically separated: they are in separate source directories and form separate library files (`libsim_std.a`, `libenvir.a` etc.)

The simulation program may contain several linked-in model components: networks, simple module types, compound module types, channel types etc. Any network (but only one at a time) can be set up for simulation which has all necessary components linked in.

[1] the only exception is textual module output: it is sent directly to ENVIR: ev<<"hello"; ev.printf(" world");

11.2 Embedding OMNeT++

Embedding is a special issue. You probably do not want to keep the appearance of the simulation program, so you do not want Cmdenv and Tkenv. You may or may not want to keep ENVIR.

What you'll absolutely need for a simulation to run is the SIM package. You can keep ENVIR if its philosophy and the infrastructure it provides (`omnetpp.ini`, certain command-line options etc.) fit into your design. Then the embedding program will take the place of Cmdenv and Tkenv.

If ENVIR does not fit your needs (for example, you want the model parameters to come from a database not from `omnetpp.ini`), then you have replace it. Your ENVIR replacement (the embedding program, practically) must implement the `cEnvir` member functions from `envir/cenvir.h`, but you have full control over the simulation.

Normally, code that sets up a network or builds the internals of a compound module comes from compiled NED source. You may not like the restriction that your simulation program can only simulate networks whose setup code is linked in. No problem; your program can contain pieces of code like what is generated by nedc and then it can build any network whose components (primarily the simple modules) are linked in. It is even possible to write an integrated environment where you can put together a network using a graphical editor and right after that you can run it, without intervening NED compilation and linkage.

11.3 The simulation kernel

The source code for the simulation kernel of OMNeT++ and the library classes reside in the sim directory.

Almost all objects are derived from `cObject` which provides a common interface for them.

11.3.1 The central object: cSimulation simulation

The `cSimulation` class stores a network and manages simulation. There is only one instance, a global object called `simulation`. The object has two basic roles:

- as a vector of modules
- holds global variables (for example, the message queue).

11.3.2 Module classes

Base class for module classes: `cModule`. Two derived classes: `cCompoundModule`, `cSimpleModule`. User simple modules are derived from `cSimpleModule`.

A `cModule` has: array of parameters, array of gates + member functions to set up and query parameters and gates.

`cSimpleModule` adds: put-aside queue, list of local objects + the virtual function `activity()` + member functions like `send()`, `receive()` etc.

Gates are represented by the `cGate` objects. Connections are not real objects: their attributes (delay, error, datarate) are managed by the connection's source gate.

11.3.3 Global registration lists

There are global objects holding lists of components available in an OMNeT++ executable. These lists are:

List object	Macro that creates a member. Class of members	Function

`cHead networks;`	`Define_Network()`
	
	`cNetworkType`
		List of available networks.
		A cNetworkType object holds a pointer to a function that can build up the network.
		`Define_Network()` macros occur in the code generated by the NEDC compiler.

`cHead modtypes;`	`Define_Module(), Define_Module_Like(),`
	
	`cModuleType`
		List of available module types.
		A `cModuleType` object knows how to create a module of a specific type. If it is compound, it holds a pointer to a function that can build up the inside.
		Usually, `Define_Module()` macros for compound modules occur in the code generated by the NEDC compiler; for simple modules, the `Define_Module()` lines are added by the user.

`cHead classes;`	`Register_Class()`
	
	`ClassRegister`
		List of available classes of which the user can create an instance.
		A `cClassRegister` object knows how to create an (empty) object of a specific class.
		The list is used by the `createOne()` function that can create an object of any (registered) type from a string containing the class name. (E.g. `ptr = createOne( "cArray")` creates an empty array.)
		`createOne()` is used by the PVM extension.
		`Register_Class()` macros are present in the simulation source files for existing classes; has to be written by the user for new classes.

`cHead functions;`	`Define_Function()`
	
	`cFunctionType`
		List of mathematical functions.
		A `cFunctionType` object holds a pointer to the function and knows how many arguments it takes.

`cHead linktypes;`	`Define_Link()`
	
	`cLinkType`
		List of link types.
		A `cLinkType` object knows how to create `cPar` objects representing the delay, error and datarate attributes for a channel.
		`Define_Link()` macros occur in the code generated by the NEDC compiler, one for each channel definition.

`cHead locals;`	`-`
	
	`any object`
		This is only 'dummy' object, it stands for the current module's local object list

`cHead superhead;`	`-`
	
	`cHead`
		List of all other lists.

11.3.4 The coroutine package

The coroutine package is in fact two coroutine packages.

There is a platform-independent coroutine package that creates all coroutine stacks inside the main stack.

It was taken from [KOF85]. It allocates stack by deep-deep recursions and then plays with setjmp() and longjmp() to switch from one another. Its drawback is that under 16-bit Intel platforms (DOS real mode and Win16), stack is limited to 64K which is not very much.

The other package allocates stack by malloc() and uses a short assembly code to initialize it for the first use. Then it also uses setjmp() and longjmp(). This is implemented under DOS + BC3.1, and also RISC6000 where the original setjmp() / longjmp() behaved in an unfriendly way and the portable coroutine package could not be used.

The coroutines are represented by the cCoroutine class. cSimpleModule has cCoroutine as one a base class.

11.3.5 Object ownership/contains relationships

Ownership: Exclusive right and duty to delete the child objects. Ownership works through cObj's ownerp/prevp/nextp and firstchildp/lastchildp pointers.

'Contains' relationship: Only for container classes, e. g. cArray or cQueue. Keeping track of contained objects works with another mechanism, *not* the previously mentioned ptrs. (E.g., cArray uses a vector, cQueue uses a separate list).

The two mechanisms are *independent*.

What cObject does:

- Owner of a new object can be explicitly given; if omitted, defaultOwner() will be used.
- An object created through the copy constructor will have the same owner as original and does not dup() or take objects owned by the original.
- Destructor calls free() for owned objects (see later)

Rules for derived classes:

- Objects contained as data members: the enclosing object should own them.

Rules for container objects derived from cObject:

- they use the functions: take(obj), drop(obj), free(obj)
- when an object is inserted, if takeOwnership() is true, should take ownership of object by calling take(obj). takeOwnership() defaults to true!
- when an object is removed, they should call drop(obj) for it if they were the owner.
- copy constructor copies should dup() and take ownership of objects that were owned by the original.
- destructor doesn't need not call free() for objects: this will be done in cObject's destructor.

The class cHead is special case: it behaves as a container, displaying objects it owns as contents.

11.4 The user interface

The source code for the user interface of OMNeT++ resides in the envir directory (common part) and in the cmdenv, tkenv directories.

The classes in the user interface are *not* derived from cObject, they are completely separated form the simulation kernel.

11.4.1 The main() function

The main() function of OMNeT++ simply sets up the user interface and runs it. Actual simulation is done in cEnvir::run() (see later).

11.4.2 The cEnvir interface

The cEnvir class has only one instance, a global object called ev:

```
cEnvir ev;
```

cEnvir basically is only an interface, its member functions hardly contain any code. cEnvir maintains a pointer to a dynamically allocated simulation application object (derived from TOmnetApp, see later) which does all actual work.

cEnvir member functions deal with four basic tasks:

- I/O for module activities; actual implementation is different for each user interface (e.g. stdin/stdout for Cmdenv, windowing in Tkenv)
- setting up and running the simulation application
- provides functions called by simulation kernel objects to get information (for example, get module parameter settings from the configuration file)
- provides functions called by simulation kernel objects to notify the user interface of some events. This is especially important for windowing user interfaces (Tkenv), because the events are like this: an object was deleted so its inspector window should be closed; a message was sent so it can be displayed; a breakpoint was hit.

11.4.3 Implementation of the user interface: simulation applications

The base class for simulation application is TOmnetApp. Specific user interfaces such as TCmdenv, TOmnetTkApp are derived from TOmnetApp.

TOmnetApp's member functions are almost all virtual.

- Some of them implement the cEnvir functions (described in the previous section)
- Others implement the common part of all user interfaces (for example: reading options from the configuration files; making the options effective within the simulation kernel)
- The run() function is pure virtual (it is different for each user interface).

TOmnetApp's data members:

- a pointer to the object holding configuration file contents (type `cInifile`);
- the options and switches that can be set from the configuration file (these members begin with `opt_`)

Concrete simulation applications:

- add new configuration options
- provide a `run()` function
- implement functions left empty in `TOmnetApp` (like `breakpointHit()`, `objectDeleted()`).

11.5 Writing inspectors for TkEnv

TBD

Appendix A

OPNET and OMNeT++

A.1 Comparison of OPNET and OMNeT++

OPNET^{TM} (from MIL3 Inc.) is a state-of-the art commercial simulation program for the modeling of communication systems. OPNET is designed to enable full-detail modeling: every tool is given to implement nonstandard protocols or behaviour.

A quote from the OPNET brochure:

- *OPNET presents an advanced graphical user interface that supports multi-windowing, makes use of menus and icons, and runs under X Windows. Supported platforms include popular engineering workstations from SUN, DEC, HP and Silicon Graphics. (Windows NT version also exists.)*
- *Graphical object-oriented editors for defining topologies and architectures directly parallel actual systems, allowing an intuitive mapping between a system and its model. OPNET's hierarchical approach simplifies the specification and representation of large and complex systems.*
- *The process editor provides a powerful and flexible language to design models of protocols, resources, applications, algorithms, queuing policies, and other processes. Specification is performed in the Proto-C language, which combines a graphical state-transition diagram approach with a library of more than 300 communication- and simulation-specific functions. The full generality and power of the C language is also available.*
- *OPNET simulations generate user-selected performance and behavioral data. Simulation results can be plotted as time series graphs, scatter plots, histograms, and probability functions. Standard statistics and confidence intervals are easily generated and additional insight can be obtained by applying mathematical operators to the collected data.*
- *OPNET provides an advanced animation capability for visualising simulation events. Both automatic and user-customised animations can be displayed interactively during or after a simulation. Animations can depict messages flowing between objects, control flow in a process, paths of mobile nodes, and dynamic values such as queue size or resource status.*
- *OPNET provides open system features including: interfaces to standard languages; the ability to take advantage of third-party libraries; an application program interface; access to databases and data files such as those generated by network analysers; and PostScript and TIFF export for desktop publishing. OPNET users are guided by a comprehensive documentation set and are backed by outstanding technical support.*

OPNET is very well designed and built commercial simulation software. The author of OMNeT++ has worked for the Hungarian distributor of OPNET for over three years and he has gained significant experience with the software. He has taken part in several computer network simulation projects for major

Hungarian companies and also delivered OPNET training. He has also written simulation models for a VSAT system in OPNET.
Following is a comparison of the features that concern general-purpose computer systems simulation (and are not specific to computer network simulation) and that are present both in OMNeT++ and OPNET.

Model hierarchy levels

OPNET	OMNeT++
network level (subnetwork nesting possible) node level (no nesting) process level (no nesting)	arbitrary levels of submodule nesting

Topology description method

OPNET provides two tools for defining module topology: graphical editors to design network and node level models, and EMA (External Model Access), an API for building model files from C programs. These tools correspond to OMNeT++'s tools in the following way:

	OPNET	OMNeT++
Graphical	graphical editor within the IDE	graphical editor: GNED
High-level	-	NED language
Low-level	EMA	C++ output of NED compilation

There is no high-level textual model description in OPNET (like NED is in OMNeT++). This means that one has either to use the graphical editor or write lengthy C code using the EMA API.

The OPNET graphical model editor can only create fixed (non-parameterized) topologies.

There's a significant difference between how EMA and OMNeT++'s NED are used. OPNET's EMA generates model files. EMA applications are standalone programs: one writes the EMA C code, compiles and runs it, and the EMA executable will generate a model file that can be read into the graphical editor or loaded by simulation programs. EMA cannot be used from within a simulation program. In contrast, the compiled NED code of OMNeT++ becomes part of the simulation program and it builds the model without having to run external programs; this means that you can have a single simulation executable that can be used to perform simulation studies on networks with different topologies.

Module parameters

	OPNET	OMNeT++
Expressions	no expressions are allowed: only literals or exact copy of another parameter	arbitrary expressions using other parameters
Parameter passing	by value	parameters can be passed by value or by reference, and be changed during simulation
Usage	by process models only	by process models; also to define flexible topologies

In OPNET, module parameter values can be passed only "as is".

Packet streams or gates

	OPNET	OMNeT++

Identification	Packet streams are numbered from 0; no names can be assigned.	Gates are identified by names. Gate vectors are supported. In the code, gates can be referenced by ID for greater speed.
Directionality	Packet streams are unidirectional.	Gates are uni-directional.

Flexible topologies

OPNET	OMNeT++
not really supported[1]	in the NED file, parameters can define submodule types, count of submodules, gates and describe connections

Tracing, animation and interactive simulation

	OPNET	OMNeT++
Tracing and debugging	powerful command line debugger (ODB)	separate window for each module's output, single-steps, run until, inspectors, snapshot, etc. (Tkenv)
Animation	mostly used in record/ playback mode; animation spec. must be given in advance (via anim. probes)	interactive execution with message-flow animation, statistics animation etc. (Tkenv)
Interactive simulation	not supported	strongly supported via object inspectors and watches. (Tkenv)

Random numbers

	OPNET	OMNeT++
Distributions provided	many built-in distributions (through algorithms)	four built-in distributions, as C functions
Additional distributions	through histograms	as C functions (algorithms); or through histograms
Random number generation	one random number generator, no support for seed selection	several independent random number generators; tool to support selecting good seed values

OPNET has many built-in distributions implemented with algorithms (C functions). Additional distributions are supported as histograms. There is only one common source of random numbers. OPNET has no aid for selecting seed values that produce long non-overlapping random number sequences.

OMNeT++, only four basic distributions are provided. They are implemented as C functions. Additional distributions can be added by the user, and they are treated exactly in the same way as built-in ones. Defining and using distributions in histogram form is also supported. OMNeT++ provides several random number generators, and also a tool for selecting good seed values.

Process description method

[1] If really necessary, it can be done through C programming (writing EMA code) and running external program to create a separate model file for each case.

	OPNET	OMNeT++
Method	finite state machine (graphical spec. only)	both process-style (coroutine-based) and finite state machine (textual spec. only)

Direct (non-scheduled) process interaction

	OPNET	OMNeT++
Method	”forced interrupt”	member function call of other module

Dynamic module creation

	OPNET	OMNeT++
What can be created	only processes within an existing module	simple modules; connections; compound modules with arbitrarily complex, parameterized topologies

Object-oriented concepts

	OPNET	OMNeT++
Language	C	C++
Objects	C API functions operating on object-like data structures; no support for inheritance[2], polymorphism or the like	full flexibility of C++: inheritance, polymorphism etc; built-in object-oriented mechanisms

Statistics collection and run-time analysis

OPNET	OMNeT++
writing observations to output file; ”probes” to select statistics to be collected; only off-line analysis (analysis of output files) is supported	writing observations to output files (roughly equivalent to OPNET’s solution); run-time processing: basic measures (mean etc); distribution estimation with histograms; quantiles (P^2 algorithm); support for detecting the end of the transient period and sufficient result accuracy

Parallel execution

OPNET	OMNeT++
not supported	supported by PVM and MPI; arbitrary synchronization can be used

Openness

[2]The graphical user interface of OPNET (from version 3.0) contains an ”inheritance mechanism” for models. This is no real inheritance in the object-oriented sense because it just means that parameter values can be changed or fixed down, parameters renamed, merged etc. There is no mention about changing the behaviour of a module (that is, anything like C++’s virtual functions).

	OPNET	OMNeT++
Input file formats	binary model files[3]; textual parameter files	text files
Output file formats	binary files[4]	text files
Availability of source	not available (only the source of the shipped models is available)	available
Embedding simulations into other software product	not supported and also not possible (the main() function cannot be supplied by the user etc.)	supported. Embedding application becomes a new "user interface" based on Envir (1); or embedding application replaces Envir (2).

A.2 Quick reference for OPNET users

This section is intended to help OPNET users learn OMNeT++ faster.

OPNET	OMNeT++
network, subnetwork, node	Compound modules
module, process	An OMNeT++ simple module corresponds to an OPNET module with its process.
interrupts, invocations, states	When using `handleMessage()`: interrupt = event, invocation = call to `handleMessage()`, state = FSM state or the value of the state vars stored in the class When using modules with `activity()`, this means a little different way of thinking from OPNET's. In OMNeT++, you write a simple module as you would write an operating system process or a thread, thus there's no need to distinguish 'states' or speak about 'invocations'. Within the simulation kernel, an 'invocation' corresponds to a `transferTo(`*`module`*`)` call. An OMNeT++ module accepts messages (and simulation time advances) within `receive...(...)` calls; `wait()` is just a `scheduleAt()` followed by a `receive()`. An OPNET interrupt is the event being processed. In this sense, OMNeT++ messages returned by `receive()` correspond to OPNET interrupts.
endsim interrupt	The `finish()` virtual member functions of the simple modules are called at the end of the simulation run. You can redefine `finish()` to write statistics etc.

`op_ima_obj_attr_get(...)`	`foo = par("foo");` `foo = module->par("foo");`
`op_ima_sim_attr_get(...)`	There are no simulation attributes. You can use the parameters of the top-level module instead: `foo = simulation.systemModule()->par("foo");`

[3] Can be read and analyzed by EMA programs.
[4] Can be exported to text files from the main OPNET program.

`op_prg_odb_print_minor(...)` `op_prg_odb_print_major(...)`	`ev « ”hello!” « endl;` `ev.printf(...);`
`op_sim_end(...)`	`simulation.error(”Your fault! error %d”,ec);`

`op_subq_....()`	Create a queue object and then manipulate it with its member functions.
	`cQueue queue;`
	`queue.insert( msg );`
	`if (!queue.empty())`
	`  msg = queue.pop();`

`List`
`op_prg_list_...()`
	`cLinkedList list;`
	`list.insert( ptr );`
	`if (!list.empty())`
	`ptr = list.pop();`

`Topology` `op_rte_...()`	The `cTopology` class offers similar functionality, and you can expect greater speed than with OPNET’s routing functions.

`Packet`
`op_pk_create(... )`
`op_pk_destroy( )`
	Use the cMessage class.
	`cMessage *msg = new cMessage;`
	`delete msg;`

packet fields
`op_pk_nfd_set(... )`
`op_pk_nfd_get_(... )`
`op_pk_fd_set(... )`
`op_pk_fd_get(... )`
	Message parameters. A parameter has both name and index.
	`msg->par("foo") = foo;`
	`msg->addPar("new-foo") = foo;`
	`int foo = msg->par("foo");`
	`int fooindex = msg->parList().find("foo");`
	`msg->par(fooindex) = foo;`

packet field modeled size	Message parameters do not have associated modelled bit sizes. Message length can be used instead.
	`msg->addPar("dest_addr") = dest_addr;`
	`msg->addLength( 32 );`

packet formats	There are no explicit packet formats in OMNeT++. However, you can write function to create messages with specific fields and length:
	`cMessage *createEthernetFrame()`
	`{`
	`  cMessage *msg = new cMessage;`
	`  msg->setKind(PACKET);`
	`  msg->addPar("source");`
	`  msg->addPar("destination");`
	`  msg->addPar("protocol");`
	`  msg->setLength( 8*16 );`
	`  return msg;`
	`}`

packet encapsulation	As in OPNET, message parameters can be assigned object pointers, thus also message pointers.
	However, there is also direct support encapsulation:
	`msg->encapsulate(innermsg)`
	`innermsg = msg->encapsulatedMsg();`
	`innermsg = msg->decapsulate();`

ICI	ICIs are also represented by cMessage objects, naturally with zero length.
	If it is important to distinguish between packets and ICIs, you can use the message kind field:
	`#define PACKET 0`
	`#define ICI 1`
	`cMessage *pk = new cMessage;`
	`pk->setKind(PACKET);`
	`cMessage *ici = new cMessage;`
	`ici->setKind(ICI);`

ICI formats	See packet formats.
ICI attributes	See packet fields.

packet and ICI in the same interrupt	You can use encapsulation. At the sender side:
	`cMessage *ici, *pk;`
	`ici->encapsulate(pk);`
	`send(ici,"out-gate");`
	The receiver side:
	`ici = receive();`
	`pk = ici->decapsulate();`

op_pk_send(...)	send(msg, "out-gate");
	send(msg, "gate-vector'', index);
	send(msg, gate_id);

op_pk_send_delayed(...)	sendDelayed(...)
op_pk_deliver(...)	sendDirect(...)

op_pk_schedule_self(...)	scheduleAt(simTime()+timeout, msg);
op_ev_cancel(...)	cancelEvent(msg);

op_dist_load(...)
op_dist_outcome(...)
	To generate random numbers from analytical distributions, use:
	uniform(...)
	intuniform(...)
	exponential(...)
	normal(...)
	truncnormal(...)
	For custom distributions you can use the histogram classes. Histograms can load distribution data from file.
	cDoubleHistogram hist;
	FILE *f = fopen("distribution.dat");
	hist.loadFromFile(f);
	fclose(f);
	double rnd = hist.random();

output vectors	The cOutVector class can be used.
	cOutVector eed("End-to-end delay");
	double d = msg->creationTime() - simTime();
	eed.record(d);

output scalars	Output scalar file exists. You can write into it with recordScalar(): recordScalar("average delay", avg_delay);

op_topo_parent()	cModule *parent = parentModule();
op_topo_child_...(...)	cSubModuleIterator

`op_topo_..._assoc_(...)`	`gate(i)/gate(name),`
	`gate->toGate()/fromGate()`
	`gate->destinationGate()/sourceGate()`
	`gate->ownerModule()`

`op_pro_create(...)`	See dynamic module creation. Note that this is a more powerful tool than OPNET's dynamic processes in that you can also create compound modules.

Prohandle	Module ID. Given the module pointer, you can obtain module ID by
	`int id = mod->id();`
	`\end{Verbat}`
	`And you can obtain module pointer from the ID:`
	`\begin{Verbatim}`
	`cModule *mod = simulation.module(id);`
	An invalid ID is negative.

`op_pro_invoke(...)`	Dynamically created modules do not need to be invoked, they live their own life. To dispatch messages to them, you can use `sendDirect(...)`
`op_pro_destroy(... )` `op_pro_destroy( self )`	`deleteModule( module );` `deleteModule();`

module memory, parent-to-child memory, argument memory to dynamic processes	Parent module can set pointers (void* data members) in the dynamically created module object any time, thus also right after creating it (parent-to-child memory), right before sending a packet to it (argument memory), and the pointer can refer to memory managed by the parent module (module memory).
	An example for argument memory. Suppose the child module class has a public data member named argmem:
	`class ChildModule : public cSimpleModule {`
	`  ...`
	`  public`
	`  void *argmem;`
	`  ...`
	`};`
	The parent module code would be:
	`childmod->argmem = argument_memory_ptr;`
	`sendDirect( msg, childmod, 0.0, "in" );`
	Child module code would be:
	`msg = receive();`
	`argument_memory_ptr = argmem;`

`op_pro_valid(...)`	Given the module id: `int valid = (id>=0) && simulation.exist(id);`

Environment files	Configuration files. Default is `omnetpp.ini`. Multiple ini files and ini file inclusion are also supported.
Process Editor	Your favourite text editor. Or *vi* :-).
Network Editor, Node Editor	Any editor to write NED files. GNED. Not very sophisticated yet though.
Simulation Tool	Use the `[Run 1]`, `[Run 2]` etc. sections in `omnetpp.ini` do describe several runs with different parameters. To create loops on different variables, you can use a shell script that creates a short ini file with the variable parameters, and include that file in `omnetpp.ini`.
probes, Probe Editor	From the ini file, you can turn on/off cOutVector objects individually as well as assign result collection interval to them.
Analysis Tool	Plove
EMA	Where you would normally use EMA, OMNeT++ NED files with parameterized topology are often enough. Otherwise, you have two choices: a) write a program to generate NED files. Text-processing languages like perl and awk are great tools for that. b) write the network-building code in C++. You can look at the output of nedc for some idea how to do it.

Appendix B

PARSEC and OMNeT++

B.1 What is PARSEC?

PARSEC is a very successful simulation language, with strong support for parallel simulation. PARSEC bears some similarity to OMNeT++ in that it is also based on threads/coroutines. The language and the software has been developed at the Parallel Computing Laboratory of the University of California at Los Angeles (UCLA), under the leadership of Prof. Rajive Bagrodia. PARSEC has been used in a number of simulation projects, for example in simulation of mobile radio networks in a military environment.

It is best to quote the PARSEC User Manual, Release 1.1 (August 1998):

PARSEC (for PARallel Simulation Environment for Complex programs) is a C-based discrete event simulation language. It adopts the process interaction approach to discrete event simulation. An object (also referred to as a physical process) or a set of objects in the physical system is represented by a logical process [a thread – roughly equivalent to an OMNeT++ simple module –Andras]. *Interactions among physical processes (events) are modeled by timestamped message exchanges among the corresponding logical processes.*
One of the important distinguishing features of PARSEC is its ability to execute a discrete-event simulation model using several different asynchronous parallel simulation protocols on a variety of parallel architectures. [...] Thus, with few modifications, a PARSEC program may be executed using the traditional sequential (Global Event List) simulation protocol or one of many parallel [...] protocols.
In addition, PARSEC provides powerful message receiving constructs that result in shorter and more natural simulation programs. [...]
The PARSEC language has been derived from the Maisie language, but with several improvements, both in the syntax of the language and in its execution environment.

The PARSEC web site is at http://pcl.cs.ucla.edu/.

PARSEC is ***not*** open source. It seems that the source code is only available to research collaborators.

B.2 What is inside the PARSEC package?

When you download and install the PARSEC distribution, basically you find:

- pcc (the PARSEC compiler), and
- 2 variants of the PARSEC runtime library

This shows that PARSEC is strictly a simulation (and parallel programming) language which is restricted to the area of entities, messages, and the tasks centered around message sending and receiving. It is difficult to

compare to OMNeT++ which is more of a complete simulation environment. (The OMNeT++ simulation library alone covers a much wider range of functionality than PARSEC as a whole.)

The primary strength of PARSEC is its parallel simulation support. The manual only describes conservative PDES, but optimistic algorithms are also supported. However, the distribution of Parallel PARSEC is limited to research collaborators (personal communication from Richard A. Meyer, Nov. 2001).

B.3 PARSEC vs. the OMNeT++ simulation kernel

This section gives a brief overview of PARSEC, with special attention to the differences compared to OMNeT++.

PARSEC is compared against the core functionality of the OMNeT++ simulation kernel, that is, message sending/receiving and the coroutines (`activity()`). Other parts of the OMNeT++ simulation kernel (e.g. statistics classes) and other parts of the OMNeT++ package have no equivalent in PARSEC.

Sample PARSEC code

The PARSEC is a programming language based on C (*not* C++!). PARSEC programs, in addition to normal C code, contain special syntactic constructs, so they do not compile as C. One has to invoke the PARSEC compiler (`pcc`) on the source code in order to translate it into C code that uses the PARSEC runtime library.

The main advantage of this solution is that the PARSEC language is clean and really elegant.

Let us see a bit of PARSEC code:

```
#include <stdio.h>
...

message job {
   int id;
   int count;
};

message add_to_your_sorc {
   ename id;\\
};
...

entity driver(int argc, char **argv) {
...
}
```

Entities and messages

In the above PARSEC code fragment, two constructs stand out at once: `message` and `entity`.

The `message` constructs define message types, and they are translated to C structs by pcc.

Entities correspond to OMNeT++'s simple modules. (PARSEC has no equivalent of OMNeT++'s compound modules.) The body of the entity contains the algorithm. Entities are implemented with coroutines or threads much like OMNeT++'s `activity()`-based simple modules; the entity body is equivalent to the `activity()` function. (PARSEC has no equivalent of `handleMessage()`-based simple modules.)

`ename` is a data type that holds entity references.

Problems with splitting up the entity body

During programming, the code of an entity may become so large that it is no longer feasible to keep it within a single function body. In OMNeT++ you can solve the problem by distributing the simple module

class's `activity()` code into new member functions which are called from activity(), and moving the some local variables of activity() into the module class so that they can also be accessed by the new member functions.

The above approach doesn't work in PARSEC, because PARSEC is C-based and entities are not C++ classes. Of course one may call ordinary C functions from the entity body, but the necessary parameters must be passed in the argument list (or as pointers to data structures).

Another solution in PARSEC is to use a construct called *friend functions* (not to be confused with C++ friend functions). PARSEC's friend functions may access the local variables of the entity (quite strange in C, but much like an inner procedure in Pascal...). However, the PARSEC documentation does not recommend using friend functions (they are slow); it says they are provided for Maisie compatibility.

The driver entity

The driver entity is special; in a way it is similar to the C main() function. PARSEC starts the simulation by creating and running a driver entity. The main task of the driver is to create all other entities in the simulation and provide them with information they need (parameter values, etc). The latter is done by sending out messages with the necessary parameters to all entities that need it.

PARSEC does not have a high-level topology description language like NED in OMNeT++; instead, the driver entity is hand-coded most of the time. (There was no mention of tools that could generate the driver entity based on some higher-level description.).

OMNeT++ compound modules have no equivalent in PARSEC. All entities are at the same level, there's no way to express hierarchy.

PARSEC has no notion of module gates, and there are no connections (in the OMNeT++ sense) among the entities. This means that when sending messages, the receiving entity must be explicitly named. Since the program contains no explicit topology information, an entity initially has no information about its communication partners (it knows no enames except its own). The usual practice is that the driver entity sends the necessary enames in an initialization message to each entity. (For illustration, see the add_to_your_sorc message type from the above code fragment. The message name itself is quite descriptive.)

The consequence of the lack of compound modules and module gates is that it is a complicated and tricky task to set up networks with but the most trivial topology. It is also very difficult to write reusable simulation components without well-defined interfaces and structuring (compound modules).

C++ syntax not allowed

It is not possible to use *any* C++ constructs in PARSEC programs. This means it is also impossible to use any C++ class libraries in PARSEC programs; only C libraries can be used.

This limitation comes from pcc itself: the parser inside pcc is written for C, and as such, it cannot parse C++ syntax. It is irrelevant whether you use a C or C++ compiler to compile pcc's output. One exception is that pcc accepts //-style comments.

Message sending

Messages can be sent to other entities with the *send* construct:

```
send message to dest-entity after delay;
```

For example, creating a new message of type Request with the parameters 10 and self (the current entity) and sending it to entity2 entity after a delay looks like this:

```
send Request{10,self} to entity2 after 5;
```

This PARSEC construct is totally equivalent in functionality to OMNeT++'s sendDirect(*message*, *delay*, *dest-module* [,*dest-gate*]) call. Since PARSEC has no equivalent of OMNeT++ gates, OMNeT++'s other send() functions which send messages through a gate are not present in PARSEC.

Message receiving constructs

PARSEC entities accept messages with the *receive* construct. *Receive* has many forms: the elegance and power of the PARSEC language stems from the *receive* construct. Some illustrative examples:

```
receive (Request req) {
...
} or receive (Release rel) {
...
} or timeout in (5) { /*"in": timeout with high priority*/
...
}
```

It is possible to add guards to the receive branches:

```
receive (Request req) when (req.units<=units) {
...
} or timeout after (5) { /*"after": timeout with low priority*/
...
}
```

These constructs have to be explicitly programmed in OMNeT++ using while loops with receive() calls and if/switch statements in its body. The reason OMNeT++ doesn't have this sort of syntax and functionality is that it is impossible to express with plain C/C++: one cannot avoid the need for a special precompiler. Having to use a precompiler, however, causes some inconvenience during program development, and in practice, there isn't as much need for this sort of complex receive constructs that would justify making it mandatory to use a precompiler for every source file.

One may wonder what happens to the messages which have arrived already but have not been accepted by the entity yet because they had no matching *receive* branch. PARSEC stores those messages in what it calls the *message buffer* of the entity. PARSEC's message buffer is practically the same as the put-aside queue in OMNeT++.

Guards may contain the special expressions `qhead(`*`msgtype`*`)`, `qempty(`*`msgtype`*`)`, `qlength(`*`msgtype`*`)` which refer to the messages in the message buffer. The programmer perceives as if each message type had a separate message buffer:

```
receive (Request req) when (qhead(Request).units<=units);
receive (Request req) when (qempty(Release) && req.units<=units);
```

Note that the `qhead()`, `qempty()` and `qlength()` operations seem to be all you can do with the message buffer, while in OMNeT++ you have free access to the put-aside queue through the cQueue member functions.

PARSEC also has a *hold* statement which is functionally equivalent to OMNeT++'s wait():

```
hold(5);
```

Cancelling messages

PARSEC has no support for cancelling messages, that is, there is no equivalent to OMNeT++'s `cancelEvent()` method. The reason is probably that message cancellation is difficult to handle in certain parallel simulation algorithms. However, this doesn't relieve the pain that such functionality would often be needed in practice (e.g. when implementing timeouts).

The PARSEC team recommends various workarounds like keeping a list of valid (or cancelled) timers and checking messages against that.

Simulation clock

The PARSEC simulation clock is of integer type: optionally, unsigned int or long long (long long is *not* a standard ANSI C/C++ type). The time unit is not specified by PARSEC: 1 may mean 1 nanosecond, 1 second or 1 hour. OMNeT++ uses double, with the time value to be interpreted as seconds.

It is probably application-specific which is the better choice, but in the case of a large simulation model put together from components written with different time granularity in mind, double seems a better choice because it is relatively insensitive to the choice of the time unit.

Random number generation

PARSEC provides platform-independent random number generators via the `pc_erand()`, `pc_nrand()`, etc. library functions.

Thread/coroutine handling

Symbol names in the PARSEC runtime library give the impression that the thread/coroutine implementation is quite similar in OMNeT++ and the single-processor implementation of PARSEC. Both simulators use a setjmp/longjmp-based coroutine library. (Although it's possible that future versions of OMNeT++ will use the Fibers API on Win32 platforms.)

The worst problem with coroutines/threads is that if you create too many of them, you'll need a lot of memory. With the current engineering workstations, it is practically impossible to create more than a few times ten thousand entities in PARSEC (this is requires a few hundred megabytes of memory).

It is possible to specify coroutine stack sizes in both PARSEC and OMNeT++. One advantage of OMNeT++ is that it can measure how much stack space a module actually uses during its operation (`stackUsage()` function), so it is relatively easy to find the optimal stack size. In PARSEC, this can only be done by trial and error (if the program crashes, a stack size was too small).

If memory requirements would grow too high due to the large number of coroutines, in OMNeT++ it is possible to rewrite modules to use `handleMessage()`, thus eliminating the need for a separate corourine stack. PARSEC has no equivalent of `handleMessage()`, so coroutine stacks cannot be eliminated.

Comparison of PARSEC and OMNeT++ as parallel simulation tools

PARSEC was created to be a parallel simulation (and parallel programming) language. It provides strong support for a wide range of conservative and optimistic PDES algorithms.

In contrast, OMNeT++ was created to be a generic simulation package, and as such, it offers *some* support for conservative PDES and Statistical Synchronization.

PARSEC runs on both shared memory multiprocessors and distributed memory systems. On a multiprocessor, NT native threads are used on Win32 platforms and the pthread library on Unix systems. MPI is used for communication between nodes on a distributed memory system.

OMNeT++ only supports distributed memory systems, using PVM or MPI for communication.

Only the sequential version of PARSEC is available for the public; the distribution of Parallel PARSEC is limited to research collaborators.

B.4 Feature summary

Feature	OMNeT++	PARSEC
Programs, components:		
graphical model editor	GNED	-
result analysis/plotting	Plove	-
interactive execution, tracing	Tkenv	-
parameter file	omnetpp.ini	-
random numbers support	Seedtool	-

Model structure		
encapsulation/grouping	compound modules	-
connections	yes (optionally: delay, data rate, bit error rate)	-
topology description	via NED, nedc	- (manually from the driver entity)
Simulation methodology		
Precompiler	- (no need, code is standard C++)	pcc (PARSEC compiler)
C++ support	based on C++	- (language based on C)
alternative to coroutines/threads	handleMessage()	-
complex message receiving constructs	- (timeout only)	yes: filter by message type, timeout, guards, etc.
message types	via subclassing cMessage or via cMessage + pars	via the message construct
module gates, sending via gates	yes	- (direct sending only)
module parameters	yes	-
dynamic module (entity) creation	yes (also compound modules)	yes
Simulation library		
statistics/histogram classes	yes (cStdDev, 3 histogram classes, P^{22} , k-split)	-
routing support	yes (cTopology)	-
FSM support	yes (FSM macros)	-
support for output files	yes (cOutVector, recordScalar(),...)	-
container classes	yes (cQueue, cArray,...)	-
Parallel simulation		
conservative	yes	yes
optimistic	-	yes
statistical synchronization	yes	possible, but no support

B.5 Correspondence between PARSEC and OMNeT++

PARSEC	**OMNeT++**
entity	simple module (`cSimpleModule`)
message	message (`cMessage`, `cPacket`,...)
message buffer of the entity	put-aside queue
send *message* to *entity* after *delay*	`sendDirect(` *`message, delay, module`* `[,`*`destgate`*`])`
send *message* to self after *delay*	`scheduleAt(` *`message`*`, simTime()+`*`delay`*`)`
n/a (PARSEC has no equivalent of OMNeT++ gates)	`send(`*`message,gate`*`)` `sendDelayed(`*`message,gate,delay`*`)`
hold(*delay*)	`wait(`*`delay`*`)`
receive (*msgtype msg*) {... }	*`msg`* `= receive()`

receive (*msgtype msg*) {... } or timeout after (*delay*) {... }	`msg = receive( delay )`
more complex *receive* constructs	`while { msg=receive(); if (...)... }`

Appendix C

NED Language Grammar

The NED language, the network topology description language of OMNeT++ will be given using the extended BNF notation.

Space, horizontal tab and new line characters counts as delimiters, so one or more of them is required between two elements of the description which would otherwise be unseparable. '//' (two slashes) may be used to write comments that last to the end of the line. The language only distinguishes between lower and upper case letters in names, but not in keywords.

In this description, the {xxx...} notation stands for one or more xxx's separated with spaces, tabs or new line characters, and {xxx,,,} stands for one or more xxx's, separated with a comma and (optionally) spaces, tabs or new line characters.

For ease of reading, in some cases we use textual definitions. The *networkdescription* symbol is the sentence symbol of the grammar.

notation	meaning
[a]	0 or 1 time a
{a}	a
{a,,,}	1 or more times a, separated by commas
{a...}	1 or more times a, separated by spaces
a|b	a or b
'a'	the character a
bold	keyword
italic	identifier

```
networkdescription ::=
    { definition... }

definition    ::=
      include
    | channeldefinition
    | simpledefinition
    | moduledefinition
    | networkdefinition

include ::=
    INCLUDE { fileName ,,, } ;

channeldefinition ::=
```

```
    CHANNEL channeltype
     [ DELAY numericvalue ]
     [ ERROR numericvalue ]
     [ DATARATE numericvalue ] $^******$
    ENDCHANNEL

simpledefinition ::=
    SIMPLE simplemoduletype
     [ machineblock ]
     [ paramblock ]
     [ gateblock ]
    ENDSIMPLE [ simplemoduletype ]

moduledefinition ::=
    MODULE compoundmoduletype
     [ machineblock$^*$ ]
     [ paramblock ]
     [ gateblock ]
     [ submodblock ]
     [ connblock ]
    ENDSIMPLE [ compoundmoduletype ]

moduletype ::=
    simplemoduletype | compoundmoduletype

machineblock ::=
    MACHINES: { machine ,,, } ;

paramblock ::=
    PARAMETERS: { parameter ,,, } ;

parameter ::=
    parametername
    | parametername : CONST [ NUMERIC ]
    | parametername : STRING
    | parametername : BOOL
    | parametername : CHAR
    | parametername : ANYTYPE

gateblock ::=
    GATES:
     [ IN: { gate ,,, } ; ]
     [ OUT: { gate ,,, } ; ]
gate ::=
    gatename [ '[]' ]

submodblock ::=
    SUBMODULES: { submodule... }

submodule ::=
    { submodulename : moduletype [ vector ]
     [ on_block$^*$... ]
     [ substparamblock... ]
     [ gatesizeblock... ] }
```

```
  | { submodulename : parametername [ vector ] LIKE moduletype
      [ on_block$^*$... ]
      [ substparamblock... ]
      [ gatesizeblock... ] }

on_block$^*$ ::=
    ON [ IF expression ]: { on_machine ,,, } ;

substparamblock    ::=
    PARAMETERS [ IF expression ]:
      { substparamname = substparamvalue,,, } ;

substparamvalue ::=
    ( [ ANCESTOR ] [ REF ] name )
    | parexpression

gatesizeblock ::=
    GATESIZES [ IF expression ]:
      { gatename vector ,,, } ;

connblock ::=
    CONNECTIONS [ NOCHECK ]: { connection ,,, } ;

connection ::=
     normalconnection | loopconnection

loopconnection ::=
    FOR { index... } DO
      { normalconnection ,,, } ;
    ENDFOR

index ::=
    indexvariable '=' expression ''...'' expression

normalconnection ::=
     { gate { --> | <-- } gate [ IF expression ]}
   | {gate --> channel --> gate [ IF expression ]}
   | {gate <-- channel <-- gate [ IF expression ]}

channel ::=
     channeltype
    | [ DELAY expression ] [ ERROR expression ] [ DATARATE expression ]
        $^******$

gate ::=
    [ modulename [vector]. ] gatename [vector]

networkdefinition ::=
    NETWORK networkname : moduletype
     [ on_block ]
     [ substparamblock ]
    ENDNETWORK

vector ::=    '[' expression ']'
```

```
parexpression ::=
    expression | otherconstvalue

expression    ::=
      expression + expression
    | expression - expression
    | expression * expression
    | expression / expression
    | expression % expression
    | expression ^ expression
    | expression == expression
    | expression != expression
    | expression < expression
    | expression <= expression
    | expression > expression
    | expression >= expression
    | expression ? expression : expression
    | expression AND expression
    | expression OR expression
    | NOT expression
    | '(' expression ')'
    | functionname '(' [ expression ,,, ] ')' $^***$
    | - expression
    | numconstvalue
    | inputvalue
    | [ ANCESTOR ] [ REF ] parametername
    | SIZEOF$^****$ '(' gatename ')'
    | INDEX$^*****$

numconstvalue ::=
    integerconstant | realconstant | timeconstant

otherconstvalue ::=
      'characterconstant'
    | ''stringconstant''
    | TRUE
    | FALSE

inputvalue ::=
    INPUT '(' default , ''prompt-string'' ')'

default ::=
    expression | otherconstvalue
```

* used with distributed execution

** used with the statistical synchronization method

*** max. three arguments. The function name must be declared in the C++ sources with the Define_Function macro.

**** Size of a vector gate.

***** Index in submodule vector.

****** Can appear in any order.

References

Simulation-related

[JAIN91] Jain, Raj: *The Art of Computer Systems Performance Analysis*. Wiley, New York, 1991.

[BFS86] Bratley P., Fox, B. L. and Schrage, L. E.: *A Guide to Simulation*. Springer-Verlag, New York, 1986.

[JCH85] Jain, Raj and Chlamtac, Imrich: *The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms without Storing Observations*, Communications of the ACM, 28(10), 1076-1085, 1985.

[PON91] Pongor, György: *OMNET: An Object-Oriented Network Simulator*. 1991 ??

[PON92] Pongor, György: *Statistical Synchronization: A Different Approach of Parallel Discrete Event Simulation*. Lappeenranta University of Technology, Data Communications Laboratory, Lappeenranta, Finland, 1992

[PON93] Pongor, György: *On the Efficiency of the Statistical Synchronization Method*. European Simulation Symposium (ESS'93), Delft, The Netherlands, Oct. 25-28, 1993

[KOF95] Kofoed, Stig: *Portable Multitasking in C++*. Dr. Dobb's Journal, November 1995. `ftp://ftp.mv.com/pub/ddj/1995/1995.11/mtask.zip`

TBD include papers of Gabor Lencse

OMNeT++-related research papers

[VAR99] *"Using the OMNeT++ Discrete Event Simulation System in Education"*. András Varga. IEEE Transactions on Education, November 1999 CD-ROM issue; abstract in vol. 42, no. 4, pp. 372, November 1999.

[VAR98a] *"K-split - On-Line Density Estimation for Simulation Result Collection"*. András Varga. In the Proceedings of the European Simulation Symposium (ESS'98). October 26-28, 1998. Nottingham, UK.

[VAR98b] *"Parameterized Topologies for Simulation Programs"*. András Varga. In the Proceedings of the Western Multiconference on Simulation (WMC'98) / Communication Networks and Distributed Systems (CNDS'98). January 11-14, 1998. San Diego, CA.

[V&F97] *"The K-Split Algorithm for the PDF Approximation of Multi-Dimensional Empirical Distributions without Storing Observations"*. András Varga and Babak Fakhamzadeh. In Proceedings of the 9th European Simulation Symposium (ESS'97), pp.94-98. October 19-22 1997, Passau, Germany.

[V&P97] *"Flexible Topology Description Language for Simulation Programs"*. András Varga and György Pongor. In Proceedings of the 9th European Simulation Symposium (ESS'97), pp.225-229. October 19-22 1997, Passau, Germany.

Former OMNeT++ documents

[OMN1] Vass Zoltán.: *PVM Extension of OMNeT++ to Support Statistical Synchronization*. Diploma Thesis, Technical University of Budapest, 1996 (in Hungarian).

[OMN2] André Maurits, George van Montfort and Gerard van de Weerd: *OMNeT++ extensions and examples*. Technical University of Budapest, Dept. of Telecommunications, 1995.

[OMN3] Jan Heijmans, Alex Paalvast, Robert van der Leij: *Network simulation using the JAR compiler for the OMNeT++ simulation system*. Technical University of Budapest, Dept. of Telecommunications, 1995.

newline [OMN4] Varga András.: *OMNeT++ - Portable User Interface for the OMNeT++ Simulation System*. Diploma Thesis, Technical University of Budapest, 1994 (in Hungarian).
[OMN5] Lencse Gábor: *Graphical Network Editor for OMNeT++*. Diploma Thesis, Technical University of Budapest, 1994 (in Hungarian).
[OMN6] Varga András.: *OMNeT++ - Portable Simulation Environment in C++*. TDK work, Technical University of Budapest, 1992 (in Hungarian).

Other simulation software

See web site

C++ language

Too many books to list.

Cyg-Win32

[CYGWIN] http://sourceware.cygnus.com/cygwin/top.html

DJGPP

[DJGPP1] Official DJGPP Home Page: http://www.delorie.com/djgpp

PVM

[PVM1] The Official PVM Home Page. http://www.epm.ornl.gov/pvm/pvm_home.html
[PVM2] http://www.sp2.uni-c.dk/PVM/PvmIntro.html
[PVM3] http://www.cse.ogi.edu/DISC/projects/mist/related-work/pvm.html

Turbo Vision

[TV1] *Borland C++ 3.1 Manuals*. Borland International, 1992.
[TV2] The TVPlus Archieve. http://wvnvm.wvnet.edu/∼u6ed4/tvhome.htm
[TV3] Sierwald, Joern: 32-bit Portable Turbo Vision. http://wvnvm.wvnet.edu/∼u6ed4/tvptsier.htm

TCL/TK

[TCLTK1] Welch, Brent: *Practical Programming in Tcl and Tk*. Prentice-Hall, 1995

[TCLTK2] HyperTcl. http://web.cs.ualberta.ca/∼wade/HyperTcl/
[TCLTK3] TCL WWW Info. http://www.sco.com/Technology/tcl/Tcl.html

Gnuplot

[GPLOT1] Brief tutorial:
http://nacphy.physics.orst.edu/DATAVIS/datavis.html

[GPLOT2] Reference:
http://www.cm.cf.ac.uk/Latex/Gnuplot/gnuplot.html

[PMTV1] PlotMTV:
http://cauchy.math.edu/workshop/Plotmtv/plotmtv.html

Xmgr

[XMGR1] Brief tutorial:
http://nacphy.physics.orst.edu/DATAVIS/xmgr.html

Index