OMNeT++

Discrete Event Simulation System

Version 3.2

User Manual

by András Varga

Last updated: March 29, 2005

Document History

Date	Author	Change
2005/10	AV	updated for the OMNeT++ 3.2 release
2005/03	AV	updated for the OMNeT++ 3.1 release
2004/12	AV	updated for the OMNeT++ 3.0 release
2003/06	AV	Mentioned Grace and ROOT in section "Visualizing...". Added section "Using STL in message classes".
2003/06	AV	OMNeT++ 2.3 released
2003/04-06	AV	"Design of OMNeT++" chapter revised, extended, and renamed to "Customization and Embedding". Added "Interpreting Cmdenv output" section to the "Running the Simulation" chapter. Added section about Akaroa in "Running the Simulation" chapter. Expanded section about writing shell scripts to control the simulation. Added background info about RNGs and warning about old RNG in "Class Library" chapter; revised/extended "Deriving new classes" section in same chapter. Bibliography converted to Bibtex, expanded and cleaned up; citations added to text. "Parallel Simulation" chapter: contents removed until new PDES implementation gets released. Revised and reorganized NED chapter. Section about message sending/receiving and other simple module related functions moved to chapter "Simple Modules"; cMessage treatment from "Simulation Library" merged with message subclassing chapter into new chapter "Messages". Deprecated cPacket. Removed sections "Simulation techniques" and "Coding conventions", and their useful fragments were incorporated elsewhere. Added/created sections about message transmission modeling, and using global variables. Added sections explaining how to implement broadcasts and retransmissions. Revised section about dynamic module creation. Deprecated putaside-queue, receiveNew(), receiveOn(). Added section "Object ownership management"; removed section on "Using shared objects".
2003/03	AV	OMNeT++ 2.3b2 released
2003/02	AV	OMNeT++ 2.3b1 released
2003/01	AV	Added chapter about message subclassing; revised chapter about running the simulation and incorporated new Cmdenv options; added new distributions and clarified many details in NED expr. handling section
Summer 2002	Ulrich Kaage	Converted from Word to LaTeX
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 network simulator. The simulator can be used for:

- traffic modeling of telecommunication networks
- protocol modeling
- modeling queueing networks
- modeling multiprocessors and other distributed hardware systems
- validating hardware architectures
- evaluating performance aspects of complex software systems
- . . . modeling any other system where the discrete event approach is suitable.

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 through 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 their own parameters. Parameters can be used to customize module behaviour and to parameterize the model's topology.

Modules at the lowest level of the module hierarchy encapsulate behaviour. These modules are termed simple modules, and they are programmed in C++ using the simulation library.

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

The simulator as well as user interfaces and tools are portable: they are known to work on Windows and on several Unix flavours, using various C++ compilers.

OMNeT++ also supports parallel distributed simulation. OMNeT++ can use several mechanisms for communication between partitions of a parallel distributed simulation, for example MPI or named pipes. The parallel simulation algorithm can easily be extended or new ones plugged in. Models do not need any special instrumentation to be run in parallel – it is just a matter of configuration. OMNeT++ can even be used for classroom presentation of parallel simulation algorithms, because simulations can be run in parallel even under the GUI which provides detailed feedback on what is going on.

OMNEST is the commercially supported version of OMNeT++. OMNeT++ is only free for academic and non-profit use – for commercial purposes one needs to obtain OMNEST licenses from Omnest Global, Inc.

1.2 Organization of this manual

The manual is organized the following way:

- The chapters 1 and 2 contain introductory material
- The second group of chapters, 3, 4 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 chapters 9 and 11 elaborate the topic further, by explaining how one can customize the network graphics and how to write NED source code comments from which documentation can be generated.
- The following chapters, 7, 8 and 10 deal with practical issues like building and running simulations and analyzing results, and present the tools OMNeT++ provides to support these tasks.
- Chapter 12 is devoted to the support of distributed execution.
- Finally, Chapter 13 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.
- Appendix A provides a reference of the NED language.

1.3 Credits

OMNeT++ has been developed by András Varga (andras@omnetpp.org, andras.varga@omnest.com).

In the early stage of the project, several people have contributed to OMNeT++. Although most contributed code is no longer part of the OMNeT++, nevertheless I'd like to acknowledge the work of the following people. First of all, I'd like thank Dr György Pongor (pongor@hit.bme.hu), my advisor at the Technical University of Budapest who initiated the OMNeT++ as a student project.

My fellow student Ákos Kun started to program the first NED parser in 1992-93, but it was abandoned after a few months. The first version of nedc was finally developed in summer 1995, by three exchange students from TU Delft: Jan Heijmans, Alex Paalvast and Robert van der Leij. nedc was first called JAR after their initials until it got renamed to nedc. nedc was further developed and refactored several times until it finally retired and got replaced by nedtool in OMNeT++ 3.0. The second group of Delft exchange students (Maurits André, George van Montfort, Gerard van de Weerd) arrived in fall 1995. They performed some testing of the simulation library, and wrote some example simulations, for example the original version of Token Ring, and simulation of the NIM game which survived until OMNeT++ 3.0. These student exchanges were organized by Dr. Leon Rothkranz at TU Delft, and György Pongor at TU Budapest.

The diploma thesis of Zoltán Vass (spring 1996) was to prepare OMNeT++ for parallel execution over PVM to OMNeT++. This code has been replaced with the new Parallel Simulation Architecture in OMNeT++ 3.0. Gábor Lencse (lencse@hit.bme.hu) was also interested in parallel simulation, namely a method called Statistical Synchronization (SSM). He implemented the FDDI model (practically unchanged until now), and added some extensions into NED for SSM. These extensions have been removed since then (OMNeT++ 3.0 does parallel execution on different principles).

The P^2 algorithm and the original implementation of the k-split algorithm was programmed in fall 1996 by Babak Fakhamzadeh from TU Delft. k-split was later reimplemented by András.

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 and ChangeLog files contain acknowledgements.

Between summer 2001 and fall 2004, the OMNeT++ CVS was hosted at the University of Karlsruhe. Credit for setting up and maintaining the CVS server goes to Ulrich Kaage. Ulrich can also be credited with converting the User Manual from Microsoft Word format to LaTeX, which was a huge undertaking and great help.

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 by passing messages to each another. 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.

Model structure is described in OMNeT++'s NED language.

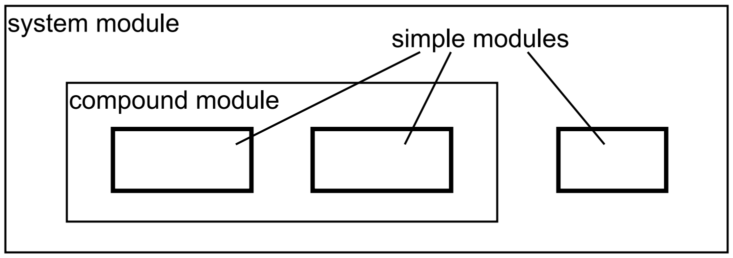


Figure 2.1: Simple and compound modules

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).

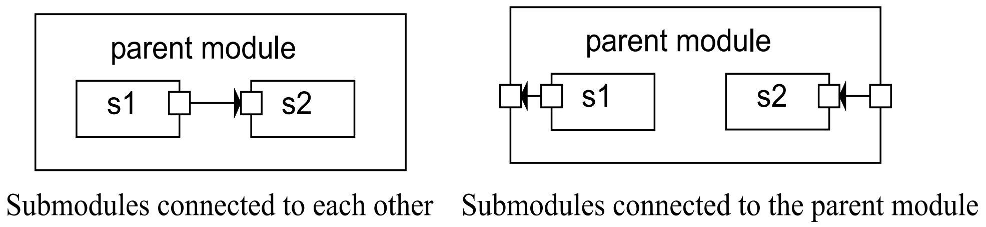

parent module
s1
s2
Submodules connected to each other
parent module
s1
s2
Submodules connected to the parent module

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 the outside world.

2.1.4 Modeling of packet transmissions

Connections can be assigned three parameters, which facilitate the modeling of communication networks, but can be useful in other models too: *propagation delay*, *bit error rate* and *data rate*, all three being optional. One can specify link parameters individually for each connection, or define link types and use them throughout the whole model.

Propagation delay is the amount of time the arrival of the message is delayed by when it travels through the channel.

Bit error rate speficifies the probability that a bit is incorrectly transmitted, and allows for simple noisy channel modelling.

Data rate is specified in bits/second, and it is used for calculating transmission time of a packet.

When data rates are in use, the sending of the message in the model corresponds to the transmission of the first bit, and the arrival of the message corresponds to the reception of the last bit. This model is not always applicable, for example protocols like Token Ring and FDDI do not wait for the frame to arrive in its entirety, but rather start repeating its first bits soon after they 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.

2.1.5 Parameters

Modules can have parameters. Parameters can be assigned either in the NED files or the configuration file omnetpp.ini.

Parameters may be used to customize simple module behaviour, and for parameterizing the model topology.

Parameters can take string, numeric or boolean values, or can contain XML data trees. 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.

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 3.

2.2 Programming the algorithms

The simple modules of a model contain algorithms as C++ functions. The full flexibility and power of the programming language can be used, supported by the OMNeT++ simulation class library. The simulation programmer can choose between event-driven and process-style description, and can freely use object-oriented concepts (inheritance, polymorphism etc) and design patterns to extend the functionality of the simulator.

Simulation objects (messages, modules, queues etc.) are represented by C++ classes. They have been designed to work together efficiently, creating a powerful simulation programming framework. The following 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 classes are also specially instrumented, allowing one to traverse objects of a running simulation and display information about them such as name, class name, state variables or contents. This feature has made it possible to create a simulation GUI where all internals of the simulation are visible.

2.3 Using OMNeT++

2.3.1 Building and running simulations

This section provides insight into working with OMNeT++ in practice: Issues such as model files, compiling and running simulations are discussed.

An OMNeT++ model consists of the following parts:

- NED language topology description(s) (`.ned` files) which describe the module structure with parameters, gates etc. NED files can be written using any text editor or the GNED graphical editor.
- Message definitions (`.msg` files). You can define various message types and add data fields to them. OMNeT++ will translate message definitions into full-fledged C++ classes.
- 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 in 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, `.msg` files are translated into C++ code using the `opp_msgc`. program. Then all C++ sources are compiled, and linked with the simulation kernel and a user interface library to form a simulation executable. NED files can either be also translated into C++ (using `nedtool`) and linked in, or loaded dynamically in their original text forms when the simulation program starts.

Running the simulation and analyzing the results

The simulation executable is a standalone program, thus it can be run on other machines without OMNeT++ or the model files being present. When the program is started, it reads a configuration file (usually called `omnetpp.ini`). This file contains settings that control how the simulation is executed, 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. It is not expected that someone will process the result files using OMNeT++ alone: output files are text files in a format which can be read into math packages like Matlab or Octave, or imported into spreadsheets like OpenOffice Calc, Gnumeric or MS Excel (some preprocessing using `sed`, `awk` or `perl` might be required, this will be discussed later). All these external programs provide rich functionality for statistical analysis and visualization, and it is outside the scope of OMNeT++ to duplicate their efforts. This manual briefly describes some data plotting programs and how to use them with OMNeT++.

Output scalar files can be visualized using the Scalars tool. It can draw bar charts, x-y plots (e.g. throughput vs offered load), or export data via the clipboard for more detailed analysis into spreadsheets and other programs.

User interfaces

The primary purpose of user interfaces is to make the internals of the model visible to the user, to control simulation execution, and possibly allow the user to 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' of the model's behaviour. The graphical user interface can also be used to demonstrate a model's operation.

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 separate from the place of their actual use. This enables the user to 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 is to be 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 in the distribution

If you installed the source distribution, the omnetpp directory on your system should contain the following subdirectories. (If you installed a precompiled distribution, some of the directories may be missing, or there might be additional directories, e.g. containing software bundled with OMNeT++.)

The simulation system itself:

```
omnetpp/         OMNeT++ root directory
  bin/           OMNeT++ executables (GNED, nedtool, 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.
    manual/      manual in HTML
    tictoc-tutorial/  introduction into using OMNeT++
    api/         API reference in HTML
    nedxml-api/  API reference for the NEDXML library
    src/         sources of the documentation
  src/           OMNeT++ sources
    nedc/        nedtool, message compiler
    sim/         simulation kernel
      parsim/    files for distributed execution
      netbuilder/files for dynamically reading NED files
    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
    scalars      output scalar analyzer and plotting tool
    nedxml/      NEDXML library
    utils/       makefile-creator, documentation tool, etc.
  test/          regression test suite
    core/        regression test suite for the simulation library
    distrib/     regression test suite for built-in distributions
    ...
```

Sample simulations are in the `samples` directory.

```
  samples/     directories for sample simulations
    aloha/     models the Aloha protocol
    cqn/       Closed Queueing Network
    ...
```

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 contain integration components for Microsoft Visual C++, etc.

Chapter 3

The NED Language

3.1 NED overview

The topology of a model is specified using the NED language. The NED language facilitates the modular description of a network. This means that a network description may consist of a number of component descriptions (channels, simple/compound module types). The channels, simple modules and compound modules of one network description can be reused in another network description.

Files containing network descriptions generally have a `.ned` suffix. NED files can be loaded dynamically into simulation programs, or translated into C++ by the NED compiler and linked into the simulation executable.

The EBNF description of the language can be found in Appendix A.

3.1.1 Components of a NED description

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

- import directives
- channel definitions
- simple and compound module definitions
- network definitions

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

3.1.3 Identifiers

Identifiers are the names of modules, channels, networks, submodules, parameters, gates, channel attributes and functions.

Identifiers must be composed of letters of the English alphabet (a-z, A-Z), numbers (0-9) and the underscore "_". Identifiers may only begin with a letter or the underscore. If you want to begin an identifier with a digit, prefix the name you'd like to have with an underscore, e.g. `_3Com`.

If you have identifiers that are composed of several words, the convention is to capitalize the beginning of every word. Also, it is recommended that you begin the names of modules, channels and networks with a capital letter, and the names of parameters, gates and submodules with a lower-case letter. Underscores are rarely used.

3.1.4 Case sensitivity

The network description and all identifiers in it are case sensitive. For example, `TCP` and `Tcp` are two different names.

3.1.5 Comments

Comments can be placed anywhere in the NED file, with the usual C++ syntax: comments begin with a double slash '//', and last until the end of the line. Comments are ignored by the NED compiler.

NED comments can be used for documentation generation, much like JavaDoc or Doxygen. This feature is described in Chapter 11.

3.2 The import directive

The `import` directive is used to import declarations from another network description file. After importing a network description, one can use the components (channels, simple/compound module types) defined in it.

When a file is imported, only the declaration information is used. Also, importing a `.ned` file does not cause that file to be compiled with the NED compiler when the parent file is NED compiled, i.e., one must compile and link all network description files – not only the top-level ones.

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

Example:

```
import "ethernet";   // imports ethernet.ned
```

3.3 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.

The syntax:

```
channel ChannelName
    //...
endchannel
```

Three attributes can be assigned values in the body of the channel declaration, all of them are optional: `delay`, `error` and `datarate`. `delay` is the propagation delay in (simulated) seconds; `error` is the bit error rate that speficifies the probability that a bit is incorrectly transmitted; and `datarate` is the channel bandwidth in bits/second, used for calculating transmission time of a packet. The attributes can appear in any order.

The values should be constants.

Example:

```
channel LeasedLine
    delay 0.0018  // sec
    error 1e-8
    datarate 128000  // bit/sec
endchannel
```

3.4 Simple module definitions

Simple modules are the basic building blocks for other (compound) modules. Simple module types are identified by names. By convention, module names begin with upper-case letters.

A simple module is defined by declaring its parameters and gates.

Simple modules are declared with the following syntax:

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

3.4.1 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 module called `TrafficGen` may have a parameter called `numOfMessages` that determines how many messages it should generate.

Parameters are identified by names. By convention, parameter names begin with lower-case letters.

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 `xml`. If the parameter type is omitted, `numeric` is assumed.

Example:

```
simple TrafficGen
    parameters:
        interarrivalTime,
        numOfMessages : const,
        address : string;
    gates: //...
endsimple
```

Parameters are assigned from NED (when the module is used as a building block of a larger compound module) or from the config file `omnetpp.ini`. `omnetpp.ini` is described in Chapter 8.

Random parameters and const

Numeric parameters can be set to return random numbers, uniformly distributed or from various distributions. For example, setting a parameter to `truncnormal(2,0.8)` would return a new random number from the truncated normal distribution with mean 2.0 and standard deviation 0.8 every time the parameter is read from the simple module (from C++ code). For example, this is useful for specifying interarrival times for generated packets or jobs.

You may want the initial parameter value to be chosen randomly, but not to change it afterwards. This can be achieved with declaring the parameter to be `const`. `const` parameters will be evaluated only once at the beginning of the simulation then set to a constant value.

It is recommended to mark every parameter with `const` unless you really want to make use of the random numbers feature.

XML parameters

Sometimes modules need more complex input than simple module parameters can describe. Then you'd put these parameters into an external config file, and let the modules read and process the file. You'd pass the file name to the modules in a string parameter.

These days, XML is increasingly becoming a standard format for configuration files as well, so you might as well describe your configuration in XML. From the 3.0 version, OMNeT++ contains built-in support for XML config files.

OMNeT++ wraps the XML parser (LibXML, Expat, etc.), reads and DTD-validates the file (if the XML document contains a DOCTYPE), caches the file (so that if you refer to it from several modules, it'll still be loaded only once), lets you pick parts of the document via an XPath-subset notation, and presents the contents to you in a DOM-like object tree.

This machinery can be accessed via the NED parameter type `xml`, and the `xmldoc()` operator. You can point `xml`-type module parameters to a specific XML file (or to an element inside an XML file) via the `xmldoc()` operator. You can assign `xml` parameters both from NED and from `omnetpp.ini`.

3.4.2 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 input and output gates. Messages are sent through output gates and received through input gates.

Gates are identified by their names. By convention, gate names begin with lower-case letters.

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 from zero.

Examples:

```
simple NetworkInterface
    parameters: //...
    gates:
        in:  fromPort, fromHigherLayer;
        out: toPort, toHigherLayer;
endsimple

simple RoutingUnit
    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.

3.5 Compound module definitions

Compound modules are modules composed of one or more submodules. Any module type (simple or compound module) can be used as a submodule. Like simple modules, compound modules can also have gates and parameters, and they can be used wherever simple modules can be used.

It is useful to think about compound modules as "cardboard boxes" that help you organize your simulation model and bring structure into it. No active behaviour is associated with compound modules – they are simply for grouping modules into larger components that can can be used either as a model (see section 3.6) or as a building block for other compound modules.

By convention, module type names (and so compound module type names, too) begin with upper-case letters.

Submodules may use parameters of the compound module. They may be connected with each other and/or with the compound module itself.

A compound module definition looks similar to a simple module definition: it has `gates` and `parameters` sections. There are two additional sections, `submodules` and `connections`.

The syntax for compound modules is the following:

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

All sections (`parameters`, `gates`, `submodules`, `connections`) are optional.

3.5.1 Compound module parameters and gates

Parameters and gates for compound modules are declared and work in the same way as with simple modules, described in sections 3.4.1 and 3.4.2.

Typically, compound module parameters are passed to submodules and used for initializing their parameters.

Parameters can also be used in defining the internal structure of the compound module: the number of submodules and sizes of gate vectors can be defined with the help of parameters, and parameters can also be used in defining the connections inside the compound module. As a practical example, you can create a `Router` compound module with a variable number of ports, specified in a `numOfPorts` parameter.

Parameters affecting the internal structure should always be declared `const`, so that accessing them always yields the same value. Otherwise, if the parameter was assigned a random value, one could get a different value each time the parameter is accessed during building the internals of the compound module, which is surely not what was meant.

Example:

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

3.5.2 Submodules

Submodules are defined in the `submodules:` section of a compound module declaration. Submodules are identified by names. By convention, submodule names begin with lower-case letters.

Submodules are instances of a module type, either simple or compound – there is no distinction. The module type must be known to the NED compiler, that is, it must have appeared earlier in the same NED file or have been imported from another NED file.

It is possible to define vectors of submodules, and the size of the vector may come from a parameter value.

When defining submodules, you can assign values to their parameters, and if the corresponding module type has gate vectors, you have to specify their sizes.

Example:

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

Module vectors

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

3.5.3 Submodule type as parameter

Sometimes it is convenient to make the name of a submodule type a parameter, so that one can easily 'plug in' any module there.

For example, assume the purpose of your simulation study is to compare different routing algorithms. Suppose you programmed the needed routing algorithms as simple modules: `DistVecRoutingNode`, `AntNetRouting1Node`, `AntNetRouting2Node`, etc. You have also created the network topology as a compound module called `RoutingTestNetwork`, which will serve as a testbed for your routing algorithms. Currently, `RoutingTestNetwork` has `DistVecRoutingNode` hardcoded (all submodules are of this type), but you want to be able to switch to other routing algorithms easily.

NED gives you the possibility to add a string-valued parameter, say `routingNodeType` to the `RoutingTestNetwork` compound module. Then you can tell NED that types of the submodules inside `RoutingTestNetwork` are not of any fixed module type, but contained in the `routingNodeType` parameter. That is all – now you are free to assign any of the `"DistVecRoutingNode"`, `"AntNetRouting1Node"` or `"AntNetRouting2Node"` string constants to this parameter (you can do that in NED, in the config file (`omnetpp.ini`), or even enter it interactively), and your network will use the routing algorithm you chose.

If you specify a wrong value, say `"FooBarRoutingNode"` when you have no `FooBarRoutingNode` module implemented, you'll get a runtime error at the beginning of the simulation: *module type definition not found*.

Inside the `RoutingTestNetwork` module you assign parameter values and connect the gates of the routing modules. To provide some degree of type safety, NED wants to make sure you didn't misspell parameter or gate names and you used them correctly. To be able to such checks, NED requires some help from you: you have to name an existing module type (say `RoutingNode`) and promise NED that all modules you're going you specify in the `routingNodeType` parameter will have (at least) the same parameters and gates as the `RoutingNode` module. [1]

All the above is achieved via the `like` keyword. The syntax is the following:

```
module RoutingTestNetwork
    parameters:
        routingNodeType: string; // should hold the name
                                 // of an existing module type
    gates: //...
    submodules:
        node1: routingNodeType like RoutingNode;
        node2: routingNodeType like RoutingNode;
        //...
    connections nocheck:
        node1.out0 --> node2.in0;
```

[1]If you like, the above solution somewhat similar to polymorphism in object-oriented languages – `RoutingNode` is like a "base class", `DistVecRoutingNode` and `AntNetRouting1Node` are like "derived classes", and the `routingNodeType` parameter is like a "pointer to a base class" which may be downcast to specific types.

```
        //...
endmodule
```

The `RoutingNode` 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 types that will be substituted (e.g. `DistVecRoutingNode`, `AntNetRouting1Node`,etc.) do not need to be declared in the NED files.

The `like` phrase lets you 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.

3.5.4 Assigning values to submodule parameters

If the module type used as submodule has parameters, you can assign values to them in the `parameters` section of the submodule declaration. As a value you can use a constant (such as `42` or `"www.foo.org"`), various parameters (most commonly, parameters of the compound module), or write an arbitrary expression containing the above.

It is not mandatory to mention and assign all parameters. Unassigned parameters can get their values at runtime: either from the configuration file (`omnetpp.ini`), or if the value isn't there either, the simulator will prompt you to enter it interactively. Indeed, for flexibility reasons it is often very useful not to "hardcode" parameter values in the NED file, but to leave them to `omnetpp.ini` where they can be changed more easily.

Example:

```
module CompoundModule
    parameters:
        param1: numeric,
        param2: numeric,
        useParam1: bool;
    submodules:
        submodule1: Node
            parameters:
                p1 = 10,
                p2 = param1+param2,
                p3 = useParam1==true ? param1 : param2;
        //...
endmodule
```

The expression syntax is very similar to C. Expressions may contain constants (literals) and parameters of the compound module being defined. Parameters can be passed by value or by reference. The latter means that the expression is evaluated at runtime each time its value is accessed (e.g. from simple module code), opening up interesting possibilities for the modeler. You can also refer to parameters of the already defined submodules, with the syntax `submodule.parametername` (or `submodule[index].parametername`).

Expressions are described in detail in section 3.7.

The `input` keyword

When a parameter does not receive a value inside NED files or in the configuration file (`omnetpp.ini`), the user will be prompted to enter its value at the beginning of the simulation. If you plan to make use of interactive prompting, you can specify a prompt text and a default value.

The syntax is the following:

```
   parameters:
      numCPUs = input(10, "Number of processors?"), // default value, prompt
      processingTime = input(10ms), // prompt text
      cacheSize = input;
```

The third version is actually the same as leaving out the parameter from the list of assignments, but you can use it to make it explicit that you do not want to assign a value from the NED file.

3.5.5 Defining sizes of submodule gate vectors

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 Node
    gates:
        in: inputs[];
        out: outputs[];
endsimple

module CompoundModule
    parameters:
        numPorts: const;
    submodules:
        node1: Node
            gatesizes:
                inputs[2], outputs[2];
        node2: Node
            gatesizes:
                inputs[numPorts], outputs[numPorts];
        //...
endmodule
```

`gatesizes` is not mandatory. If you omit `gatesizes` for a gate vector, it will be created with zero size.

One reason for omitting `gatesizes` is that you'll want to use the `gate++` ("extend gate vector with a new gate") notation later in the `connections` section.

3.5.6 Conditional parameters and gatesizes sections

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

Example:

```
module Chain
    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 disjoint 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.

3.5.7 Connections

The compound module definition specifies how the gates of the compound module and its immediate submodules are connected.

You can connect two submodules or a submodule with its enclosing compound module. (For completeness, you can also connect two gates of the compound module on the inside, but this is rarely needed). This means that NED does not permit connections that span multiple levels of hierarchy – this restriction enforces compound modules to be self-contained, and thus promotes reusability. Gate directions must also be observed, that is, you cannot connect two output gates or two input gates.

Only one-to-one connections are supported, so a particular gate may only be used occur in one connection. One-to-many and many-to-one connections can be achieved using simple modules that duplicate messages or merge message flows. The rationale is that wherever such fan-in or fan-out occurs in a model, it is usually associated with some processing anyway that makes it necessary to use simple modules.

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

Example:

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

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

The *gate++* notation allows you to extend a gate vector with new gates, without having to declare the vector size in advance with `gatesizes`. This feature is very convenient for connecting nodes of a network:

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

```
endsimple

module SmallNet
    submodules:
        node: Node[6];
    connections:
        node[0].out++ --> node[1].in++;
        node[0].in++ <-- node[1].out++;

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

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

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

        node[4].out++ --> node[5].in++;
        node[4].in++ <-- node[5].out++;
endmodule
```

A connection:

- may have attributes (delay, bit error rate or data rate) or use a named channel;
- may occur inside a for-loop (to create multiple connections);
- may be conditional.

These connection types are described in the following sections.

Single connections and channels

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

```
    node1.outGate --> node2.inGate;
```

You can specify a channel by its name:

```
    node1.outGate --> Fiber --> node2.inGate;
```

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

One can also specify the channel parameters directly:

```
    node1.outGate --> error 1e-9 delay 0.001 --> node2.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
    node1.outGate[i] --> node2[i].inGate
endfor;
```

The result of the above loop connection can be illustrated as depicted in Fig. 3.1.

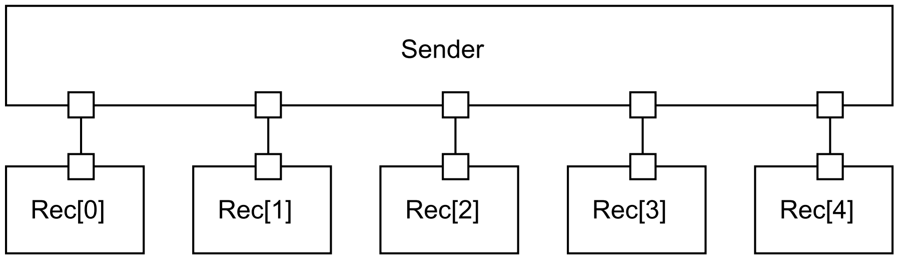


Figure 3.1: Loop connection

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

One can create nested loops by specifying more than one indices in the `for` statement, with the first variable forming the outermost loop.

```
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;
```

Conditional connections

Creation of a connection can be made conditional, using the `if` keyword:

```
for i=0..n do
    node1.outGate[i] --> node2[i].inGate if i%2==0;
endfor;
```

The `if` condition is evaluated for each connection (in the above example, for each *i* value), and the decision is made individually each time whether to create the the connection or not. In the above example we connected every second gate. Conditions may also use random variables, as shown in the next section.

The nocheck modifier

By default, NED requires that all gates be connected. Since this check can be inconvenient at times, it can be turned off using the `nocheck` modifier.

The following example generates a random subgraph of a full graph.

```
module RandomConnections
    parameters: //..
    gates: //..
    submodules: //..
    connections nocheck:
        for i=0..n-1, j=0..n-1 do
            node[i].out[j] --> node[j].in[i] 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.

3.6 Network definitions

Module module declarations (compound and simple module declarations) just define module types. To actually get a simulation model that can be run, you need to write a *network definition*.

A network definition declares a simulation model as an instance of a previously defined module type. You'll typically want to use a compound module type here, although it is also possible to program a model as a self-contained simple module and instantiate it as a "network".

There can be several network definitions in your NED file or NED files. The simulation program that uses those NED files will be able to run any of them; you typically select the desired one in the config file (`omnetpp.ini`).

The syntax of a network definition is similar to that of a submodule declaration:

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

Here, `WirelessLAN` is the name of previously defined compound module type, which presumably contains further compound modules of types `WirelessHost`, `WirelessHub`, etc.

Naturally, only module types without gates can be used in network definitions.

Just as in submodules, you do not need to assign values to all parameters. Unassigned parameters can get their values from the config file (`omnetpp.ini`) or will be interactively prompted for.

3.7 Expressions

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

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.

When an expression is used for a parameter value, it is evaluated each time the parameter value is accessed (unless the parameter is declared `const`, see 3.4.1). This means that a simple module querying a non-const parameter during simulation may get different values every time (e.g. if the value involves a random variable, or it contains other parameters taken by reference). Other expressions (including `const` parameter values) are evaluated only once.

XML-type parameters can be used to conveniently access external XML files or parts of them. XML-type parameters can be assigned with the `xmldoc()` operator, also described in this section.

3.7.1 Constants

Numeric and string constants

Numeric constants are accepted in their usual decimal or scientific notations.

String constants

String constants use double quotes.

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:
    propagationDelay = 560ms, // 0.560s
    connectionTimeout = 6m 30s 500ms, // 390.5s
    recoveryIntvl = 0.5h; // 30 min
```

The following units can be used:

Unit	Meaning
`ns`	nanoseconds
`us`	microseconds
`ms`	milliseconds
`s`	seconds
`m`	minutes (60s)
`h`	hours (3600s)
`d`	days (86400s)

3.7.2 Referencing parameters

Expressions can use the parameters of the enclosing compound module (the one being defined) and of submodules defined earlier in NED file. The syntax for the latter is `submod.param` or `submod[index].param`.

There are two keywords that you can use with a parameter name: `ancestor` and `ref`. The first one (`ancestor` *param*) means that if compound module doesn't have such a parameter, further modules up in the module hierarchy will be searched for the parameter. `ancestor` is considered bad practice because it violates the encapsulation principle and can only be checked at runtime. It is provided for the rare case when it is really needed.

`ref` *param* takes the parameter by reference, meaning that runtime changes to the parameter will propagate to all modules which take that parameter by reference. Like `ancestor`, `ref` should also be used very sparingly. One possible use is tuning a model at runtime, in search for an optimum: one defines a parameter at the highest level of the model, and lets other modules take it by reference – then if you change the parameter value at runtime (manually or from a simple module), it will affect the whole model. In another setup, reference parameters may be used to propagate status values to neighbouring modules.

3.7.3 Operators

The operators supported in NED are similar to C/C++ operators, with the following differences:

- ^ is used for power-of (and not bitwise XOR as in C)
- ## is used for logical XOR (same as != between logical values), and # is used for bitwise XOR
- the precedence of bitwise operators (&, |, #) have been raised to bind stronger than relational operations. This precedence is usually more convenient than the C/C++ one.

All values are represented as `double`s. For the bitwise operators, `double`s are converted to `unsigned long` [2] using the C/C++ builtin conversion (type cast), the operation is performed, then the result is converted back to `double`. Similarly, for the logical operators &&, || and ##, the operands are converted to `bool` using the C/C++ builtin conversion (type cast), the operation is performed, then the result is converted back to `double`. For modulus (%), the operands are converted to `long`.

Here's the complete list of operators, in order of decreasing precendence:

Operator	**Meaning**
-, !, ~	unary minus, negation, bitwise complement
^	power-of
*, /, %	multiply, divide, modulus
+, -	add, subtract
«, »	bitwise shift
&, |, #	bitwise and, or, xor
== != >, >= <, <=	equal not equal greater, greater or equal less, less or equal
&&, ||, ##	logical operators and, or, xor
?:	the C/C++ "inline if"

3.7.4 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.

The following example describes a router with several ports and one routing unit. We assume that gate vectors `in[]` and `out[]` have the same size.

```
module Router
    gates:
        in: in[];
        out: out[];
    submodules:
        port: PPPInterface[sizeof(in)]; // one PPP for each input gate
            parameters: interfaceId = 1+index; // 1,2,3...
        routing: RoutingUnit;
            gatesizes:
                in[sizeof(in)];  // one gate pair for each port
                out[sizeof(in)];
    connections:
        for i = 0..sizeof(in)-1 do
            in[i] --> port[i].in;
            out[i] <-- port[i].out;
```

[2] In case you are worried about `long` values being not accurately represented in `double`s, this is not the case. IEEE-754 `double`s have 52 bit mantissas, and integer numbers in that range are represented without rounding errors.

```
            port[i].out --> routing.in[i];
            port[i].in <-- routing.out[i];
        endfor;
endmodule
```

3.7.5 The `xmldoc()` operator

The `xmldoc()` operator can be used to assign XML-type parameters, that is, point them to XML files or to specific elements inside XML files.

`xmldoc()` has two flavours: one accepts a file name, the second accepts a file name plus an XPath-like expression which selects an element inside the XML file. Examples:

```
xmlparam = xmldoc("someconfig.xml");
xmlparam = xmldoc("someconfig.xml", "/config/profile[@id='2']");
```

OMNeT++ supports a subset of the XPath 1.0 specification; details are documented below.

From the C++ code you'd access the XML element like this:

```
cXMLElement *rootelement = par("xmlparam").xmlValue();
```

The `cXMLElement` class provides a DOM-like access to the XML document. You can then navigate the document tree, extract the information you need, and store it in variables or your internal data structure. `cXMLElement` is documented in Chapter 6.

You can also read XML parameters from omnetpp.ini:

```
[Parameters]
**.interface[*].config = xmldoc("conf.xml")
```

or

```
[Parameters]
**.interface[*].config = xmldoc("all-in-one.xml", "/config/interfaces/interface[2]")
```

3.7.6 XML documents and the XPath subset supported

`xmldoc()` with two arguments accepts a path expression to select an element within the document. The expression syntax is similar to XPath.

If the expression matches several elements, the first element (in preorder depth-first traversal) will be selected. (This is unlike XPath, which selects all matching nodes.)

The expression syntax is the following:

- An expression consists of *path components* (or "steps") separated by "`/`" or "`//`".
- A path component can be an element tag name, "`*`", "`.`" or "`..`".
- "`/`" means child element (just as e.g. in `/usr/bin/gcc`); "`//`" means an element any levels under the current element.
- "`.`", "`..`" and "`*`" mean current element, parent element, and an element with any tag name, respectively.

- Element tag names and "`*`" can have an optional predicate in the form "`[position]`" or "`[@attribute='val`
  Positions start from zero.

- Predicate of the form "`[@attribute=$param]`" are also accepted, where *`$param`* can be one of: `$MODULE_FULLPATH`, `$MODULE_FULLNAME`, `$MODULE_NAME`, `$MODULE_INDEX`, `$MODULE_ID`, `$PARENTMODUL`
  `$PARENTMODULE_FULLNAME`, `$PARENTMODULE_NAME`, `$PARENTMODULE_INDEX`, `$PARENTMODULE_ID`, `$GRANDPARENTMODULE_FULLPATH`, `$GRANDPARENTMODULE_FULLNAME`, `$GRANDPARENTMODULE_NAME`, `$GRANDPARENTMODULE_INDEX`, `$GRANDPARENTMODULE_ID`.$^{New!}$

Examples:

- `/foo` – the root element which must be called `<foo>`
- `/foo/bar` – first `<bar>` child of the `<foo>` root element
- `//bar` – first `<bar>` anywhere (depth-first search!)
- `/*/bar` – first `<bar>` child of the root element which may have any tag name
- `/*/*/bar` – first `<bar>` child two levels below the root element
- `/*/foo[0]` – first `<foo>` child of the root element
- `/*/foo[1]` – second `<foo>` child of the root element
- `/*/foo[@color='green']` – first `<foo>` child which has attribute "color" with value "green"
- `//bar[1]` – a `<bar>` element anywhere which is the second `<bar>` among its siblings
- `//*[@color='yellow']` – any element anywhere which has attribute "color" with value "yellow"
- `//*[@color='yellow']/foo/bar` – first `<bar>` child of first `<foo>` child of a "yellow-colored" element anywhere

Path support allows you put all your XML configuration into a single XML document, when you would otherwise end up with lots of small XML files. For example, consider the following `sample.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <traffic-profile id="low">
        ...
    </traffic-profile>
    <traffic-profile id="medium">
        ...
    </traffic-profile>
    <traffic-profile id="high">
        ...
    </traffic-profile>
</root>
```

In one simulation you can configure module parameters as `xmldoc("sample.xml", "//traffic-profile[@id`
in another run as `xmldoc("sample.xml", "//traffic-profile[@id='medium']")`, and so on.

3.7.7 Functions

In NED expressions, you can use the following 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.

It is possible to add new ones, see 3.7.9.

3.7.8 Random values

Expressions may contain random variates from different distributions. Such parameters, unless declared as `const`, return different values each time they are evaluated.

If the parameter is declared as `const`, it is only evaluated once at the beginning of the simulation, and subsequent queries on the parameter will always return the same value.

Random variate functions use one of the random number generators (RNGs) provided by OMNeT++. By default this is generator 0, but you can specify which one is to be used.

OMNeT++ has the following predefined distributions:

Function	**Description**
	Continuous distributions
`uniform(a, b, `*`rng=0`*`)`	uniform distribution in the range [a,b)
`exponential(mean, `*`rng=0`*`)`	exponential distribution with the given mean
`normal(mean, stddev, `*`rng=0`*`)`	normal distribution with the given mean and standard deviation
`truncnormal(mean, stddev, `*`rng=0`*`)`	normal distribution truncated to nonnegative values
`gamma_d(alpha, beta, `*`rng=0`*`)`	gamma distribution with parameters alpha>0, beta>0
`beta(alpha1, alpha2, `*`rng=0`*`)`	beta distribution with parameters alpha1>0, alpha2>0
`erlang_k(k, mean, `*`rng=0`*`)`	Erlang distribution with k>0 phases and the given mean
`chi_square(k, `*`rng=0`*`)`	chi-square distribution with k>0 degrees of freedom
`student_t(i, `*`rng=0`*`)`	student-t distribution with i>0 degrees of freedom
`cauchy(a, b, `*`rng=0`*`)`	Cauchy distribution with parameters a,b where b>0
`triang(a, b, c, `*`rng=0`*`)`	triangular distribution with parameters a<=b<=c, a!=c
`lognormal(m, s, rng=0)`	lognormal distribution with mean m and variance s>0
`weibull(a, b, `*`rng=0`*`)`	Weibull distribution with parameters a>0, b>0
`pareto_shifted(a, b, c, `*`rng=0`*`)`	generalized Pareto distribution with parameters a, b and shift c
	Discrete distributions
`intuniform(a, b, `*`rng=0`*`)`	uniform integer from a..b

`bernoulli(p, `*`rng=0`*`)`	result of a Bernoulli trial with probability 0<=p<=1 (1 with probability p and 0 with probability (1-p))
`binomial(n, p, `*`rng=0`*`)`	binomial distribution with parameters n>=0 and 0<=p<=1
`geometric(p, `*`rng=0`*`)`	geometric distribution with parameter 0<=p<=1
`negbinomial(n, p, `*`rng=0`*`)`	binomial distribution with parameters n>0 and 0<=p<=1
`poisson(lambda, `*`rng=0`*`)`	Poisson distribution with parameter lambda

If you do not specify the optional `rng` argument, the functions will use random number generator 0.

Examples:

```
intuniform(0,10)/10  // one of: 0, 0.1, 0.2, ..., 0.9, 1
exponential(5)       // exponential with mean=5 (thus parameter=0.2)
2+truncnormal(5,3)   // normal distr with mean 7 truncated to >=2 values
```

The above distributions are implemented with C functions, and you can easily add new ones (see section 3.7.9). Your distributions will be treated in the same way as the built-in ones.

3.7.9 Defining new functions

To use user-defined functions, one has to code the function in C++. The C++ function must take 0, 1, 2, 3, or 4 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
```

If your function takes parameters that are `int` or `long` or some other type which is not `double`, you can create wrapper function that takes all doubles and does the conversion. In this case you have to register the wrapper function with the `Define_Function2()` macro which allows a function to be registered with a name different from the name of the function that implements it. You can do the same if the return value differs from `double`.

```
#include <omnetpp.h>

long factorial(int k)
{
  ...
}

static double _wrap_factorial(double k)
{
  return factorial((int)k);
}

Define_Function2(factorial, _wrap_factorial, 1);
```

3.8 Parameterized compound modules

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

3.8.1 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:
        rteProcessingDelay, rteBuffersize,
        numOfPorts: const;
    gates:
        in: inputPorts[];
        out: outputPorts[];
    submodules:
        localUser: Application;
        routing: RoutingUnit
            parameters:
                processingDelay = rteProcessingDelay,
                buffersize = rteBuffersize;
            gatesizes:
                input[numOfPorts+1],
                output[numOfPorts+1];
        portIf: PPPNetworkInterface[numOfPorts]
            parameters:
                retryCount = 5,
                windowSize = 2;
    connections:
        for i=0..numOfPorts-1 do
            routing.output[i] --> portIf[i].fromHigherLayer;
            routing.input[i] <-- portIf[i].toHigherLayer;
            portIf[i].toPort --> outputPorts[i];
            portIf[i].fromPort <-- inputPorts[i];
        endfor;
        routing.output[numOfPorts] --> localUser.input;
        routing.input[numOfPorts] <-- localUser.output;
endmodule
```

Example 2: Chain

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

```
module Chain
    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

One can use conditional connections to build a binary tree. The following NED code loops through all possible node pairs, and creates the connections needed for a binary tree.

```
simple BinaryTreeNode
    gates:
        in: fromupper;
        out: downleft;
        out: downright;
endsimple

module BinaryTree
    parameters:
        height: const;
    submodules:
        node: BinaryTreeNode [ 2^height-1 ];
    connections nocheck:
        for i = 0..2^height-2, j = 0..2^height-2 do
            node[i].downleft --> node[j].fromupper if j==2*i+1;
            node[i].downright --> node[j].fromupper if j==2*i+2;
        endfor;
endmodule
```

Note that not every 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.

An alert reader might notice that there is a better alternative to the above code. Each node except the ones at the lowest level of the tree has to be connected to exactly two nodes, so we can use a single loop to create the connections.

```
module BinaryTree2
    parameters:
        height: const;
    submodules:
        node: BinaryTreeNode [ 2^height-1 ];
    connections nocheck:
        for i=0..2^(height-1)-2 do
            node[i].downleft --> node[2*i+1].fromupper;
            node[i].downright --> node[2*i+2].fromupper;
        endfor;
endmodule
```

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 && uniform(0,1)<connectedness;
        endfor;
endmodule
```

Note the use of the nocheck modifier here too, to turn off error messages given by the network setup code for unconnected gates.

3.8.2 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, you can resort to specifying constant submodule/gate vector sizes and explicitly listing all connections, like you would do it in most existing simulators.

3.8.3 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. 3.2).

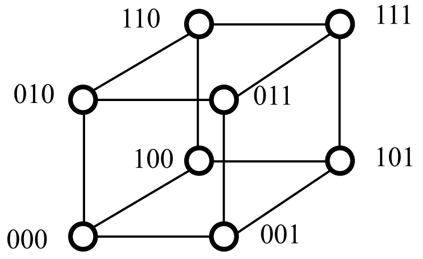


Figure 3.2: Hypercube topology

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.

3.9 Large networks

There are situations when using hand-written NED files to describe network topology is inconvenient, for example when the topology information comes from an external source like a network management program.

In such case, you have two possibilities:

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.

3.9.1 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 relatively 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 resulting NED files can either be translated by `nedtool` into C++ and compiled in, or loaded dynamically.

3.9.2 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, using dynamic module creation (to be described later, in section 4.11). The code which you need to write would be similar to the `*_n.cc` files output by `nedtool`.

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.

3.10 XML binding for NED files

To increase interoperability, NED files (and also message definition files) have an XML representation. Any NED file can be converted to XML, and any XML file which corresponds to the NED DTD can be converted to NED. [3]

XML is well suited for machine processing. For example, stylesheet transformations (XSLT) can be used to extract information from NED files, or the other way round, create NED files from external info present in XML form. One practical application of XML is the `opp_neddoc` documentation generation tool which is described in Chapter 11.

The `nedtool` program (which also translates NED to C++ code) can be used to convert between NED and XML.

Converting a NED file to XML:

[3]DTD stands for Document Type Descriptor, and it defines a "grammar" for XML files. More info can be found on the W3C web site, www.w3.org.

```
nedtool -x wireless.ned
```

It generates `wireless_n.xml`. Several switches control the exact content and details of the resulting XML as well as the amount of checks made on the input.

Converting the XML representation back to NED:

```
nedtool -n wireless.xml
```

The result is `wireless_n.ned`.

Using nedtool as NED compiler to generate C++ code:

```
nedtool wireless.ned
```

The resulting code is more compact than the one created by `nedtool`'s predecessor `nedc`. As a result, `nedtool`-created `_n.cc` C++ files compile much faster.

You can generate C++ code from the XML format as well:

```
nedtool wireless.xml
```

Chapter 4

Simple Modules

Simple modules are the active components in the model. Simple modules are programmed in C++, using the OMNeT++ class library. The following sections contain a short introduction to discrete event simulation in general, explain how its concepts are implemented in OMNeT++, and give an overview and practical advice on how to design and code simple modules.

4.1 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.

4.1.1 Discrete Event Simulation

A *Discrete Event System* is a system where state changes (events) happen at discrete instances in 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. (Other systems can be modelled e.g. with continuous simulation models.)

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 "interesting" 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 which refer to how long the simulation program has been running and how much CPU time it has consumed.

4.1.2 The event loop

Discrete event simulation maintains the set of future events in a data structure often called FES (Future Event Set) or FEL (Future Event List). 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 differ considerably 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 canceling timeouts.

The simulation stops when there are no 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 user will typically want to record statistics into output files.

4.1.3 Simple modules in OMNeT++

In OMNeT++, events occur inside simple modules. Simple modules encapsulate C++ code that generates events and reacts to events, in other words, implements the behaviour of the model.

The user creates simple module types 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 packed with simulation-related functionality, doesn’t do anything useful by itself – you have to redefine some virtual member functions to make it do useful work.

These member functions are the following:

- void `initialize()`
- void `handleMessage(cMessage *msg)`
- void `activity()`
- 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 `handleMessage()` and `activity()` functions are called during event processing. This means that the user will implement the model's behavior in these functions. `handleMessage()` and `activity()` implement different event processing strategies: for each simple module, the user has to redefine exactly one of these functions.

`handleMessage()` is a method that is called by the simulation kernel when the module receives a message. `activity()` is a coroutine-based solution which implements the process interaction approach (coroutines are non-preemptive (i.e. cooperative) threads). Generally, it is recommended that you prefer `handleMessage()` to `activity()` – mainly because `activity()` doesn't scale well. Later in this chapter we'll discuss both methods including their advantages and disadvantages.

Modules written with `activity()` and `handleMessage()` can be freely mixed within a simulation model.

The `finish()` functions are called when the simulation terminates successfully. The most typical use of `finish()` is the recording of statistics collected during simulation.

4.1.4 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 by 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 mathematically should be. For example, addition is not an associative operation when it comes to floating point calculations: $(x + y) + z! = x + (y + z)$! (See [Gol91]). This means that it is generally not a good idea to rely on arrival times of two events being the same unless they are calculated in exactly the same way.

One may suggest introducing a small *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*). This approach, however, would be more likely to cause confusion than actually cure the problem.

4.1.5 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 that.

4.2 Packet transmission modeling

4.2.1 Delay, bit error rate, data rate

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 (*ber*) is the probability that a bit is incorrectly transmitted. Thus, the probability that a message of *n* bits length is transferred without bit errors is:

$P_{nobiterror} = (1 - ber)^{l}ength$

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. 4.1).

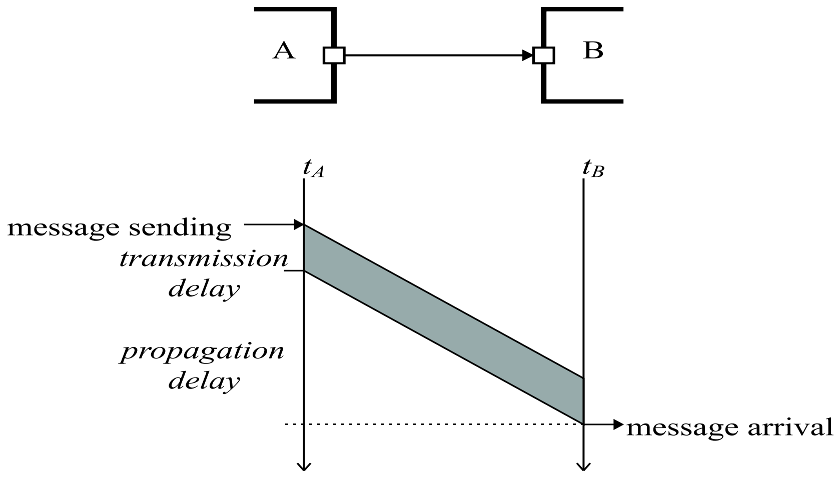


Figure 4.1: Message transmission

The above model may not be suitable to model all protocols. In Token Ring and FDDI, stations start to repeat bits before the whole frame arrives; in other words, frames “flow through” the stations, being delayed only a few bits. In such cases, the data rate modeling feature of OMNeT++ cannot be used.

If a message travels along a route, passing through successive links and compound modules, the model behaves as if each module waited until the last bit of the message arrives and only started its transmission afterwards. (Fig. 4.2).

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.

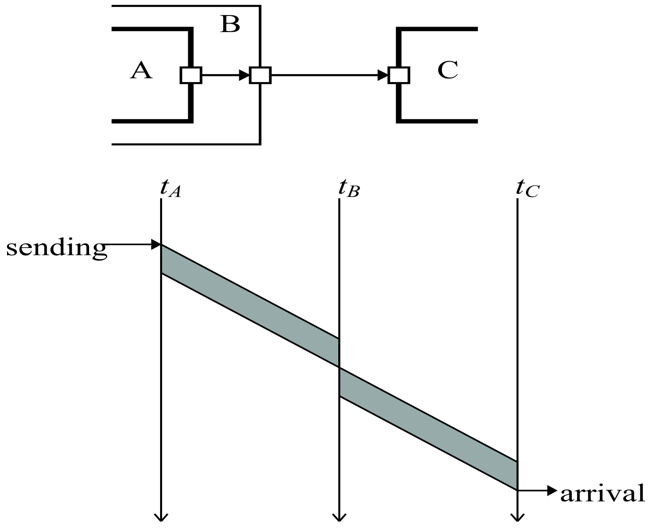


Figure 4.2: Message sending over multiple channels

4.2.2 Multiple transmissions on links

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

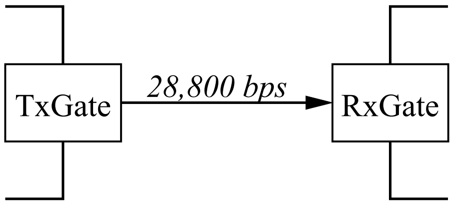


Figure 4.3: Connection with a data rate

While a message is under transmission, other messages have to wait until the transmission is completed. You can still send messages while the gate is busy, but the beginning of the modeled message transmission will be delayed, just as if the gate had an internal queue for the messages waiting to be transmitted.

The OMNeT++ class library provides 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 directly connected to the simple module’s output gate but is the second one in the route, you have to check the second gate’s busy condition.

Implementation of message sending

Message sending is implemented like this: the arrival time and the bit error flag of a message are calculated immediately after the `send()` (or a similar) function is invoked. That is, if the message travels through several links before 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.

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 chose one of two paths:

- write a sender module such that it schedules events for when the gate finishes its current transmission and sends 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 sent are buffered at the destination module (or the remote end of the link) until they are 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 signals the arrival of the packet. In some implementations, output gates also have packet queues where packets will be buffered until the channel is ready (available for transmission).

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

In OMNeT++ one can implement *point-to-point transmitter* modules with packet queues if needed. For example, the INET Framework follows this approach.

4.3 Defining simple module types

4.3.1 Overview

As mentioned before 4.1.3, a simple module is nothing more than a C++ class which has to be subclassed from `cSimpleModule`, with one or more virtual member functions redefined to define its behavior.

The class has to be registered with OMNeT++ via the `Define_Module()` macro. The `Define_Module()` line should always be put into `.cc` or `.cpp` files and not header file (`.h`), because the compiler generates code from it. [1]

The following `HelloModule` is about the simplest simple module one could write. (We could have left out the `initialize()` method as well to make it even smaller, but how would it say Hello then?) Note `cSimpleModule` as base class, and the `Define_Module()` line.

[1]For completeness, there is also a `Define_Module_Like()` macro, but its use is discouraged and might even be removed in future OMNeT++ releases.

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

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

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

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

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

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

```
// file: HelloModule.ned
simple HelloModule
    gates:
        in: in;
endsimple
```

4.3.2 Constructor

Simple modules are never instantiated by the user directly, but rather by the simulation kernel. This implies that one cannot write arbitrary constructors: the signature must be what is expected by the simulation kernel. Luckily, this contract is very simple: the constructor must be public, and must take no arguments:

```
  public:
    HelloModule();  // constructor takes no arguments
```

`cSimpleModule` itself has two constructors:

1. `cSimpleModule()` – one without arguments
2. `cSimpleModule(size_t stacksize)` – one that accepts the coroutine stack size

The first version should be used with `handleMessage()` simple modules, and the second one with `activity()` modules. (With the latter, the `activity()` method of the module class runs as a coroutine which needs a separate CPU stack, usually of 16..32K. This will be discussed in detail later.) Passing zero stack size to the latter constructor also selects `handleMessage()`.

Thus, the following constructor definitions are all OK, and select `handleMessage()` to be used with the module:

```
HelloModule::HelloModule() {...}
HelloModule::HelloModule() : cSimpleModule() {...}
```

It is also OK to omit the constructor altogether, because the compiler-generated one is suitable too.

The following constructor definition selects `activity()` to be used with the module, with 16K of coroutine stack:

```
HelloModule::HelloModule() : cSimpleModule(16384) {...}
```

4.3.3 Constructor and destructor vs initialize() and finish()

The `initialize()` and `finish()` methods will be discussed in a later section in detail, but because their apparent similarity to the constructor and the destructor is prone to cause some confusion, we'll briefly cover them here.

The constructor gets called when the module is created, as part of the model setup process. At that time, everything is just being built, so there isn't a lot things one can do from the constructor. In contrast, `initialize()` gets called just before the simulation starts executing, when everything else has been set up already.

`finish()` is for recording statistics, and it only gets called when the simulation has terminated normally. It does not get called when the simulations stops with an error message. The destructor always gets called at the end, no matter how the simulation stopped, but at that time it is fair to assume that the simulation model has been halfway demolished already.

Based on the above, the following conventions exist for these four methods:

Constructor conventions:

Set pointer members of the module class to `NULL`; postpone all other initialization tasks to `initialize()`.

`initialize()` conventions:

Perform all initialization tasks: read module parameters, initialize class variables, allocate dynamic data structures with `new`; also allocate and initialize self-messages (timers) if needed.

`finish()` conventions:

Record statistics. Do **not** `delete` anything or cancel timers – all cleanup must be done in the destructor.

destructor conventions:

Delete everything which was allocated by `new` and is still held by the module class. With self-messages (timers), use the `cancelAndDelete(msg)` function! It is almost always wrong to just delete a self-message from the destructor, because it might be in the scheduled events list. The `cancelAndDelete(msg)` function checks for that first, and cancels the message before deletion if necessary.

4.3.4 Compatibility with earlier versions

OMNeT++ versions earlier than 3.2 expected a different module class constructor, with the following signature:

```
MyModule(const char *name, cModule *parentModule, size_t stack=<stacksize>);
```

For convenience, a macro named `Module_Class_Members()` was also provided, which expanded to a default (i.e. do-nothing) constructor implementation.

In OMNeT++ 3.2, the `Module_Class_Members()` macro has been retained but expands to the new constructor definition. Thus a module which uses `Module_Class_Members()` does not need to be changed to work with OMNeT++ 3.2 or later. When compatiblity with older versions is no longer required, the macro call can simply be deleted.

Some (few) modules have hand-coded constructors instead of using `Module_Class_Members()`. These modules will produce a compile error with OMNeT++ 3.2 or later, saying *no appropriate constructor available*. The easiest way to get them working is to add `=NULL` default value to both the `name` and the `parentModule` arguments:

```
MyModule(const char *name=NULL, cModule *parentModule=NULL, size_t stack=<stacksize>);
```

Again, when compatibility with older OMNeT++ versions is no longer required, the redundant constructor arguments can be removed.

4.3.5 "Garbage collection" and compatibility

OMNeT++ versions before the 3.2 release had a feature which often was, informally and also somewhat incorrectly, called *"garbage collection"* (GC). The purpose of this feature was to mitigate the need for writing destructors, and often constructors as well by providing automatic cleanup at the end of the simulation. (It did not do anything during simulation, as the name might suggest.)

OMNeT++ (all versions) keep track of user-allocated simulation objects (typically: messages) and their ownerships. What the *"garbage collection"* feature did was that during the cleanup of the model, after each module destructor finished, it checked whether there were simulations objects left that were apparently owned by that module but not deallocated by the destructor – and if it found such objects, it invoked `delete` on them.

It worked out nicely in 90 percent of cases, but occasionally it resulted in spurious crashes which were hard to debug for users not familiar with OMNeT++ internals or lacking advanced C++ skills. [2]

Starting from OMNeT++ 3.2, this cleanup-time GC mechanism has been disabled by default (`perform-gc=` configuration option, see 8.2.6), and it generally not recommended to turn it back on. It does not do any harm to run any simulation model without GC (apart from the memory leak).

It is expected that existing modules will be updated for OMNeT++ 3.2 sooner or later, by adding proper constructors and destructors. To catalyse this process, OMNeT++ dumps the list of unreleased objects at the end of the simulation. This dump can also be turned off in the configuration (`print-undisposed=` configuration option, see 8.2.6).

4.3.6 An example

The following code is a bit longer but actually useful simple module implementation. It demonstrates several of the above concepts, plus some others which will be explained in later sections:

1. constructor, initialize and destructor conventions
2. using messages for timers
3. accessing module parameters
4. recording statistics at the end of the simulation
5. documenting the programmer's assumptions using ASSERT()

[2]These crashes occurred due to lack of information available to the GC mechanism, e.g. C++ provides no way to detect from the pointer whether an object is part of an array, or is inside a struct or class. The solution was to use pointers: pointer array, pointer as class member, etc.

```
// file: FFGenerator.h

#include <omnetpp.h>

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

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

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


// file: FFGenerator.cc

#include "FFGenerator.cc"

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

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

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

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

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

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

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

FFGenerator::~FFGenerator()
{
    cancelAndDelete(sendMessageEvent);
}
```

It also needs a NED declaration to be able to use it in NED files:

```
// file: FFGenerator.ned
simple FFGenerator
    parameters:
        sendIaTime: numeric;
    gates:
        out: out;
endsimple
```

4.3.7 Using global variables

If possible, avoid using global variables, including static class members. They are prone to cause several problems. First, they are not reset to their initial values (to zero) when you rebuild the simulation in Tkenv, or start another run in Cmdenv. This may produce surprising results. Second, they prevent you from running your simulation in parallel. When using parallel simulation, each partition of your model (may) run in a separate process, having its own copy of the global variables. This is usually not what you want.

The solution is to encapsulate the variables into simple modules as private or protected data members, and expose them via public methods. Other modules can then call these public methods to get or set the values. Calling methods of other modules will be discussed in section 4.10. Examples of such modules are the `Blackboard` in the *Mobility Framework*, and `InterfaceTable` and `RoutingTable` in the *INET Framework*.

4.4 Adding functionality to cSimpleModule

This section discusses `cSimpleModule`'s four previously mentioned member functions, intended to be redefined by the user: `initialize()`, `handleMessage()`, `activity()` and `finish()`, plus a fifth, less frequently used one, `handleParameterChange`.

4.4.1 handleMessage()

Function called for each event

The idea is that at each event (message arrival) we simply call a user-defined function. This function, `handleMessage(cMessage *msg)` is a virtual member function of `cSimpleModule` which does nothing by default – the user has to redefine it in subclasses and add the message processing code.

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 event loop inside the simulator handles both `activity()` and `handleMessage()` simple modules, and it corresponds to the following pseudocode:

```
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 a `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 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.

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 normally used to record statistics information accumulated in data members of the class at the end of the simulation.

Application area

`handleMessage()` is in most cases a better choice than `activity()`:

1. When you expect the module to be used in large simulations, involving several thousand modules. In such cases, the module stacks required by `activity()` would simply consume too much memory.
2. For modules which maintain little or no state information, such as packet sinks, `handleMessage()` is more convenient to program.
3. 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.

Example 1: Protocol models

Models of protocol layers in a communication network tend to have a common structure on a high level because fundamentally they all have to react to three types of events: to messages arriving from higher layer protocols (or apps), to messages arriving from lower layer protocols (from the network), and to various timers and timeouts (that is, self-messages).

This usually results in the following source code pattern:

```
class FooProtocol : public cSimpleModule
{
  protected:
    // state variables
    // ...

    virtual void processMsgFromHigherLayer(cMessage *packet);
    virtual void processMsgFromLowerLayer(FooPacket *packet);
    virtual void processTimer(cMessage *timer);

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// ...

void FooProtocol::handleMessage(cMessage *msg)
{
    if (msg->isSelfMessage())
        processTimer(msg);
    else if (msg->arrivedOn("fromNetw"))
        processMsgFromLowerLayer(check_and_cast<FooPacket *>(msg));
    else
```

```
        processMsgFromHigherLayer(msg);
}
```

The functions `processMsgFromHigherLayer()`, `processMsgFromLowerLayer()` and `processTimer()` are then usually split further: there are separate methods to process separate packet types and separate timers.

Example 2: Simple traffic generators and sinks

The code for simple packet generators and sinks programmed with `handleMessage()` might be as simple as the following pseudocode:

```
PacketGenerator::handleMessage(msg)
{
    create and send out a new packet;
    schedule msg again to trigger next call to handleMessage;
}

PacketSink::handleMessage(msg)
{
    delete msg;
}
```

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
{
  public:
    Generator() : cSimpleModule()
  protected:
    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 3: Bursty traffic generator

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

We add some data members to the class:

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

The code:

```
class BurstyGenerator : public cSimpleModule
{
  protected:
    int burstLength;
    int burstCounter;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(BurstyGenerator);

void BurstyGenerator::initialize()
{
    // init parameters and state variables
    burstLength = par("burstLength");
    burstCounter = burstLength;
    // 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 (--burstCounter == 0)
    {
        // schedule next burst
        burstCounter = burstLength;
        scheduleAt(simTime()+exponential(5.0), msg);
    }
    else
    {
        // schedule next sending within burst
        scheduleAt(simTime()+exponential(1.0), msg);
    }
}
```

Pros and Cons of using `handleMessage()`

Pros:

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

Cons:

- local variables cannot be used to store state information
- need to redefine `initialize()`

Usually, `handleMessage()` should be preferred to `activity()`.

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$^T M$ 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.

4.4.2 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()` – 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.

Application area

Generally you should prefer `handleMessage()` to `activity()`. The main problem with `activity()` is that it doesn't scale because every module needs a separate coroutine stack. It has also been observed that `activity()` does not encourage a good programming style.

There is one scenario where `activity()`'s process-style description is convenient: 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 have to handle several connections simultaneously, you may dynamically create them as instances of the simple module above. Dynamic module creation will be discussed later.

There are situations when you certainly *do not want* to use `activity()`. If your `activity()` function contains no `wait()` and it has only one `receive()` call at the top of an infinite loop, there's no point in using `activity()` and the code should be written with `handleMessage()`. The body of the infinite loop would then become the body to `handleMessage()`, state variables inside `activity()` would become data members in the module class, and you'd initialize them in `initialize()`.

Example:

```
void Sink::activity()
{
    while(true)
    {
        msg = receive();
        delete msg;
    }
}
```

should rather be programmed as:

```
void Sink::handleMessage(cMessage *msg)
{
    delete msg;
}
```

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 other coroutines. This implies that each coroutine must have its 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. It is important to understand, however, 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()` and `wait()`, so their implementations contain a `transferTo(main)` call somewhere.

Their pseudocode, as implemented in OMNeT++:

```
receive()
{
    transferTo(main)
    retrieve current event
    return the event // remember: events = messages
}

wait()
{
    create event e
    schedule it at (current sim. time + wait interval)
    transferTo(main)
    retrieve current event
    if (current event is not e) {
        error
    }
    delete e  // note: actual impl. reuses events
    return
}
```

Thus, the `receive()` and `wait()` calls are special points in the `activity()` function, because they are 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

The simulation programmer needs to define the processor stack size for coroutines. This cannot be automated.

16 or 32 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-in 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 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()`.

You do need `finish()`, however, if you want to write statistics at the end of the simulation. Because `finish()` cannot access the local variables of `activity()`, you have to put the variables and objects containing 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 in 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
}
```

Pros and Cons of using `activity()`

Pros:

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

Cons:

- limited scalability: coroutine stacks can 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
- does not enforce a good programming style: using `activity()` tends to lead to unreliable, spaghetti code

In most cases, cons outweigh pros and it is a better idea to use `handleMessage()` instead.

Other simulators

Coroutines are used by a number of other simulation packages:

- All simulation software which inherits from SIMULA (e.g. C++SIM) is 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 with "normal" preemptive threads). The philosophy is quite similar to OMNeT++. PARSEC, being "just" a programming language, it has a more elegant syntax but far fewer features than OMNeT++.
- Many Java-based simulation libraries are based on Java threads.

4.4.3 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's `initialize()` function runs *before* that of its submodules.

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 of the order of `initialize()`: first submodules, then the encompassing compound module. (The bottom line is that at the moment there is no "official" possibility to redefine `initialize()` and `finish()` for compound modules; the unofficial way is to write into the nedtool-generated C++ code. Future versions of OMNeT++ will support adding these functions to compound modules.)

This is summarized in the following pseudocode:

```
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. no errors
        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. This is because 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 operations which are supposed to run only on successful completion. Cleanup code should go into 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() const;
```

At the beginning of the simulation, `initialize(0)` is called for *all* modules, then `initialize(1)`, `initialize(2)`, etc. You can think of it like initialization takes place in several "waves". 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. [3]

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 several simulators by introducing a special *end-of-simulation* event. This is not a very good practice because the simulation programmer has to code the models (often represented as 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 can also be witnessed 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 has led to awkward programming in many cases, so for PARSEC end-of-simulation events were dropped in favour of `finish()` (called `finalize()` in PARSEC).

4.4.4 handleParameterChange()$^{New!}$

The `handleParameterChange()` method was added in OMNeT++ 3.2, and it gets called by the simulation kernel when a module parameter changes. The method signature is the following:

```
void handleParameterChange(const char *parname);
```

[3]Note `const` in the `numInitStages()` declaration. If you forget it, by C++ rules you create a *different* function instead of redefining the existing one in the base class, thus the existing one will remain in effect and return 1.

The user can redefine this method to let the module react to runtime parameter changes. A typical use is to re-read the changed parameter, and update the module state if needed. For example, if a timeout value changes, one can restart or modify running timers.

The primary motivation for this functionality was to facilitate the implementation of *scenario manager* modules which can be programmed to change parameters at certain simulation times. Such modules can be very convenient in studies involving transient behaviour.

The following example shows a queue module, which supports runtime change of its `serviceTime` parameter:

```
void Queue::handleParameterChange(const char *parname)
{
    if (strcmp(parname, "serviceTime")==0)
    {
        // queue service time parameter changed, re-read it
        serviceTime = par("serviceTime");

        // if there any job being serviced, modify its service time
        if (endServiceMsg->isScheduled())
        {
            cancelEvent(endServiceMsg);
            scheduleAt(simTime()+serviceTime, endServiceMsg);
        }
    }
}
```

4.4.5 Reusing module code via subclassing

It is often needed to have several variants of a simple module. A good design strategy is to create a simple module class with the common functionality, then subclass from it to create the specific simple module types.

An example:

```
class ModifiedTransportProtocol : public TransportProtocol
{
  protected:
    virtual void recalculateTimeout();
};

Define_Module(ModifiedTransportProtocol);

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

4.5 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 handle 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 define, 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 parentheses 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,newState);
```

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

Debugging FSMs

FSMs can log their state transitions `ev`, with the output looking like this:

```
...
FSM GenState: leaving state SLEEP
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SEND
FSM GenState: leaving state SEND
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SLEEP
...
```

To enable the above output, you have to `#define FSM_DEBUG` before including `omnetpp.h`.

```
#define FSM_DEBUG    // enables debug output from FSMs
#include <omnetpp.h>
```

The actual logging is done via the `FSM_Print()` macro. It is currently defined as follows, but you can change the output format by undefining `FSM_Print()` after including `omnetpp.ini` and providing a new definition instead.

```
#define FSM_Print(fsm,exiting)
  (ev << "FSM " << (fsm).name()
      << ((exiting) ? ": leaving state " : ": entering state ")
      << (fsm).stateName() << endl)
```

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 scary, but if you're dying to see it, it is 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 bursty generator. It will have 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
{
  protected:
    // 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("sleepTimeMean");
    burstTimeMean = par("burstTimeMean");
    sendIATime = par("sendIATime");
    msgLength = &par("msgLength");
    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;
    }
  }
}
```

4.6 Sending and receiving messages

On an abstract level, an OMNeT++ simulation model is a set of simple modules that communicate with each other via message passing. The essence of simple modules is that they create, send, receive, store, modify, schedule and destroy messages – everything else is supposed to facilitate this task, and collect statistics about what was going on.

Messages in OMNeT++ are instances of the `cMessage` class or one of its subclasses. Message objects are created using the C++ `new` operator and destroyed using the `delete` operator when they are no longer needed. During their lifetimes, messages travel between modules via gates and connections (or are sent directly, bypassing the connections), or they are scheduled by and delivered to modules, representing internal events of that module.

Messages are described in detail in chapter 5. At this point, all we need to know about them is that they are referred to as `cMessage *` pointers. Message objects can be given descriptive names (a `const char *` string) that often helps in debugging the simulation. The message name string can be specified in the constructor, so it should not surprise you if you see something like `new cMessage("token")` in the examples below.

4.6.1 Sending messages

Once created, a message object can be sent through an output gate using one of the following functions:

```
send(cMessage *msg, const char *gateName, int index=0);
send(cMessage *msg, int gateId);
send(cMessage *msg, cGate *gate);
```

In the first function, the argument `gateName` 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 and third functions use the gate Id and the pointer to the gate object. They are faster than the first one because they don't have to search through the gate array.

Examples:

```
send(msg, "outGate");
send(msg, "outGates", i); // send via outGates[i]
```

The following code example creates and sends messages every 5 simulated seconds:

```
int outGateId = findGate("outGate");
while(true)
{
  send(new cMessage("packet"), outGateId);
  wait(5);
}
```

Modeling packet transmissions

If you're sending messages over a link that has (nonzero) data rate, it is modeled as described earlier in this manual, in section 4.2.

If you want to have full control over the transmission process, you'll probably need the `isBusy()` and `transmissionFinishes()` member functions of `cGate`. They are described in section 4.8.3.

4.6.2 Broadcasts and retransmissions

When you implement broadcasts or retransmissions, two frequently occurring tasks in protocol simulation, you might feel tempted to use the same message in multiple `send()` operations. Do not do it – you cannot send the same message object multiple times. The solution in such cases is duplicating the message.

Broadcasting messages

In your model, you may need to broadcast a message to several destinations. Broadcast can be implemented in a simple module by sending out copies of the same message, for example on every gate of a gate vector. As described above, you cannot use the same message pointer for in all `send()` calls – what you have to do instead is create copies (duplicates) of the message object and send them.

Example:

```
for (int i=0; i<n; i++)
{
    cMessage *copy = (cMessage *) msg->dup();
    send(copy, "out", i);
}
delete msg;
```

You might have noticed that copying the message for the last gate is redundant (we could send out the original message), so it can be optimized out like this:

```
for (int i=0; i<n-1; i++)   // note n-1 instead of n
{
    cMessage *copy = (cMessage *) msg->dup();
    send(copy, "out", i);
}
send(msg, "out", n-1);  // send original on last gate
```

Retransmissions

Many communication protocols involve retransmissions of packets (frames). When implementing retransmissions, you cannot just hold a pointer to the same message object and send it again and again – you'd get the *not owner of message* error on the first resend.

Instead, whenever it comes to (re)transmission, you should create and send copies of the message, and retain the original. When you are sure there will not be any more retransmission, you can delete the original message.

Creating and sending a copy:

```
// (re)transmit packet:
cMessage *copy = (cMessage *) packet->dup();
send(copy, "out");
```

and finally (when no more retransmissions will occur):

```
delete packet;
```

Why?

A message is like any real world object – it cannot be at two places at the same time. Once you've sent it, the message object no longer belongs to the module: it is taken over by the simulation kernel, and will eventually be delivered to the destination module. The sender module should not even refer to its pointer any more. Once the message arrived in the destination module, that module will have full authority over it – it can send it on, destroy it immediately, or store it for further handling. The same applies to messages that have been scheduled – they belong to the simulation kernel until they are delivered back to the module.

To enforce the rules above, all message sending functions check that you actually own the message you are about to send. If the message is with another module, it is currently scheduled or in a queue etc., you'll get a runtime error: *not owner of message*. [4]

4.6.3 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( someDelay );
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 in Section 4.6.7, "Self-messages", would need to be employed.

There is also a more straightforward method than those mentioned above: delayed sending. Delayed sending can be achieved by using one of these functions:

```
sendDelayed(cMessage *msg, double delay, const char *gateName, int index);
sendDelayed(cMessage *msg, double delay, int gateId);
sendDelayed(cMessage *msg, double delay, cGate *gate);
```

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 non-negative.

Example:

```
sendDelayed(msg, 0.005, "outGate");
```

4.6.4 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:

```
sendDirect(cMessage *msg, double delay, cModule *mod, int gateId)
sendDirect(cMessage *msg, double delay, cModule *mod, const char *gateName, int index=-
sendDirect(cMessage *msg, double delay, cGate *gate)
```

In addition to the message and a delay, it also takes the destination module and gate. The gate should be an *input* gate and should not be connected. In other words, the module needs dedicated gates for receiving via `sendDirect()`. (Note: For leaving a gate unconnected in a compound module, you'll need to specify `connections nocheck:` instead of plain `connections:` in the NED file.)

An example:

```
cModule *destinationModule = parentModule()->submodule("node2");
double delay = truncnormal(0.005, 0.0001);
sendDirect(new cMessage("packet"), delay, destinationModule, "inputGate");
```

At the destination module, there is no difference between messages received directly and those received over connections.

[4]The feature does not increase runtime overhead significantly, because it uses the object ownership management (described in Section 6.12); it merely checks that the owner of the message is the module that wants to send it.

4.6.5 Receiving messages

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

Messages are received using the `receive()` function. `receive()` is a member of `cSimpleModule`.

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

The `receive()` function accepts an optional *timeout* parameter. (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. [5]

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

if (msg==NULL)
{
    ...   // handle timeout
}
else
{
    ...  // process message
}
```

4.6.6 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*).

```
wait( delay );
```

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 for a (potentially random amount of) time, specified
  // in the interArrivalTime module parameter
  wait( par("interArrivalTime") );

  // generate and send message
  ...
}
```

It is a runtime error if a message arrives during the wait interval. If you expect messages to arrive during the wait period, you can use the `waitAndEnqueue()` function. It takes a pointer to a queue object (of class `cQueue`, described in chapter 6) in addition to the wait interval. Messages that arrive during the wait interval will be accumulated in the queue, so you can process them after the `waitAndEnqueue()` call returned.

[5] Putaside-queue and the functions `receiveOn()`, `receiveNew()`, and `receiveNewOn()` were deprecated in OMNeT++ 2.3 and removed in OMNeT++ 3.0.

```
cQueue queue("queue");
...
waitAndEnqueue(waitTime, &queue);
if (!queue.empty())
{
  // process messages arrived during wait interval
  ...
}
```

4.6.7 Modeling events using self-messages

In most simulation models it is necessary to implement timers, or schedule events that occur at some point in the future. For example, when a packet is sent by a communications protocol model, it has to schedule an event that would occur when a timeout expires, because it will have to resent the packet then. As another example, suppose you want to write a model of a server which processes jobs from a queue. Whenever it begins processing a job, the server model will want to schedule an event to occur when the job finishes processing, so that it can begin processing the next job.

In OMNeT++ you solve such tasks by letting the simple module send a message to itself; the message would be delivered to the simple module at a later point of time. Messages used this way are called self-messages. Self-messages are used to model events which occur within the module.

Scheduling an event

The module can send a message to itself using the `scheduleAt()` function. `scheduleAt()` accepts an *absolute* simulation time, usually calculated as `simTime()`+*delta*:

```
scheduleAt(absoluteTime, msg);
scheduleAt(simtime()+delta, msg);
```

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.

As an example, here's how you could implement your own `wait()` function in an `activity()` simple module, if the simulation kernel didn't provide it already:

```
cMessage *msg = new cMessage();
scheduleAt(simtime()+waitTime, msg);
cMessage *recvd = receive();
if (recvd!=msg)
   // hmm, some other event occurred meanwhile: error!
...
```

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

```
if (msg->isScheduled())
  // currently scheduled
else
  // not scheduled
```

Re-scheduling an event

If you want to reschedule an event which is currently scheduled to a different simulation time, first you have to cancel it using `cancelEvent()`.

Cancelling an event

Scheduled self-messages 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.

Implementing timers

The following example shows how to implement timers:

```
cMessage *timeoutEvent = new cMessage("timeout");

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

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

4.6.8 Stopping the simulation

Normal termination

You can finish the simulation with the `endSimulation()` function:

```
endSimulation();
```

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 you want your simulation to stop if it detects an error condition, you can call 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 error text, as it will be added by OMNeT++.

4.7 Accessing module parameters

Module parameters can be accessed by calling the `par()` member function of `cModule`:

```
cPar& delayPar = 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 numTasks = par("numTasks");
double processingDelay = par("processingDelay");
```

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& waitTime = par("waitTime");
for(;;)
{
  //...
  wait( (simtime_t)waitTime );
}
```

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("waitTime") = 0.12;
```

Or:

```
cPar& waitTime = par("waitTime");
waitTime = 0.12;
```

The `cPar` class is discussed in more detail in section 6.6.

4.7.1 Emulating parameter arrays

As of version 3.2, OMNeT++ does not support parameter arrays, but in practice they can be emulated using string parameters. One can assign the parameter a string which contains all values in a textual form (for example, "`0 1.234 3.95 5.467`"), then parse this string in the simple module.

The `cStringTokenizer` class can be quite helpful for this purpose. The constructor accepts a string, which it regards as a sequence of tokens (words) separated by delimiter characters (by default, spaces). Then, calling the `nextToken()` method several times will return the tokens one by one. After the last token, it returns `NULL`.

For example, you can parse a string containing a sequence of integers into a vector using the following code:

```
const char *str = "34 42 13 46 72 41"; // input
std::vector<int> numbers;  // array to hold the result

cStringTokenizer tokenizer(str);
const char *token;
while ((token = tokenizer.nextToken())!=NULL)
    numbers.push_back(atoi(token));   // convert and store
```

The class also has a `hasMoreTokens()` method, so the above code can also be written as

```
...
cStringTokenizer tokenizer(str);
while (tokenizer.hasMoreTokens())
    numbers.push_back(atoi(tokenizer.nextToken()));
```

For converting `long`s and `double`s, replace `atoi()` with `atol()` and `atof()`, respectively.

For storing the tokens in a string vector, the `cStringTokenizer` class has a convenience function named `asVector()`, so conversion can be done in just one line of code:

```
const char *str = "34 42 13 46 72 41";
std::vector<std::string> strVec = cStringTokenizer(str).asVector();
```

4.8 Accessing gates and connections

4.8.1 Gate objects

Module gates are `cGate` objects. Gate objects know whether, and to which gate they are connected. They can also be queried on 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 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.

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)
```

Instead of `gate->size()`, you can also call the `gateSize()` method of `cModule`, which does the same:

```
int size2 = gateSize("out");
```

For non-vector gates, `size()` returns 1 and `index()` returns 0.

Zero-size gate vectors are represented with a placeholder gate whose `size()` method returns zero and cannot be connected.

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 `fromApp` and `in[3]` and output gates of `toApp` and `status`, the array may look like this:

ID	dir	name[index]
0	*input*	`fromApp`
1	*output*	`toApp`
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, thus 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 gate 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.

You can also obtain gate IDs with the `findGate()` member of `cModule`:

```
int id1 = findGate("out");
int id2 = findGate("outvect",5);
```

4.8.2 Connection parameters

Connection attributes (propagation delay, transmission data rate, bit error rate) are represented by the channel object, which is available via the source gate of the connection.

```
cChannel *chan = outgate->channel();
```

`cChannel` is a small base class. All interesting attributes are part of its subclass `cBasicChannel`, so you have to cast the pointer before getting to the delay, error and data rate values.

```
cBasicChannel *chan = check_and_cast<cBasicChannel *>(outgate->channel());
double d = chan->delay();
double e = chan->error();
double r = chan->datarate();
```

You can also change the channel attributes with the corresponding `setXXX()` functions. Note, however, that (as it was explained in section 4.2) changes will not affect messages already sent, even if they have not begun transmission yet.

4.8.3 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. (If the gate in not currently transmitting, `transmissionFinishes()` returns the simulation time when it finished its last transmission.)

The semantics have been described in section 4.2.

An example:

```
cMessage *packet = new cMessage("DATA");
packet->setByteLength(1024);  // 1K

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 directly connected to the simple module's output gate but is the second one in the route, you have to check the second gate's busy condition. You could 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.

4.8.4 Connectivity

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 `destination-Gate()`.

4.9 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";
```

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.

```
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 would crash with an access violation because of the NULL pointer dereference.)

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 iter(*parentModule()); !iter.end(); iter++)
{
  ev << iter()->fullName();
}
```

(`iter()` 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 iter(*parentModule()); !iter.end(); iter++)
{
  if (iter()->isName(name())) // if iter() is in the same
                              // vector as this module
  {
    int itsIndex = iter()->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()`.

4.10 Direct method calls between modules

In some simulation models, there might be modules which are too tightly coupled for message-based communication to be efficient. In such cases, the solution might be calling one simple module's public C++ methods from another module.

Simple modules are C++ classes, so normal C++ method calls will work. Two issues need to be mentioned, however:

- how to get a pointer to the object representing the module;
- how to let the simulation kernel know that a method call across modules is taking place.

Typically, the called module is in the same compound module as the caller, so the `parentModule()` and `submodule()` methods of `cModule` can be used to get a `cModule*` pointer to the called module. (Further ways to obtain the pointer are described in the section 4.9.) The `cModule*` pointer then has to be cast to the actual C++ class of the module, so that its methods become visible.

This makes the following code:

```
cModule *calleeModule = parentModule()->submodule("callee");
Callee *callee = check_and_cast<Callee *>(calleeModule);
callee->doSomething();
```

The `check_and_cast<>()` template function on the second line is part of OMNeT++. It does a standard C++ `dynamic_cast`, and checks the result: if it is NULL, `check_and_cast` raises an OMNeT++ error. Using `check_and_cast` saves you from writing error checking code: if `calleeModule` from the first line is NULL because the submodule named `"callee"` was not found, or if that module is actually not of type `Callee`, an error gets thrown from `check_and_cast`.

The second issue is how to let the simulation kernel know that a method call across modules is taking place. Why is this necessary in the first place? First, the simulation kernel always has to know which module's code is currently executing, in order to several internal mechanisms to work correctly. (One such mechanism is ownership handling.) Second, the Tkenv simulation GUI can animate method calls, but to be able to do that, it has to know about them.

The solution is to add the `Enter_Method()` or `Enter_Method_Silent()` macro at the top of the methods that may be invoked from other modules. These calls perform context switching, and, in case of `Enter_Method()`, notify the simulation GUI so that animation of the method call can take place. `Enter_Method_Silent()` does not animate the call. `Enter_Method()` expects a `printf()`-like argument list – the resulting string will be displayed during animation.

```
void Callee::doSomething()
{
    Enter_Method("doSomething()");
    ...
}
```

4.11 Dynamic module creation

4.11.1 When do you need dynamic module creation

In some situations you need to dynamically create and maybe destroy modules. For example, when simulating a mobile network, you may create a new module whenever a new user enters the simulated area, and dispose of them when they leave the area.

As another example, when implementing a server or a transport protocol, it might be convenient to dymically create modules to serve new connections, and dispose of them when the connection is closed. (You would write a manager module that receives connection requests and creates a module for each connection. The Dyna example simulation does something like this.)

Both simple and compound modules can be created dynamically. If you create a compound module, all its submodules will be created recursively.

It is often convenient to use direct message sending with dynamically created modules.

Once created and started, dynamic modules aren't any different from "static" modules; for example, one could also delete static modules during simulation (though it is rarely useful.)

4.11.2 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 factory 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 is possible to call its factory method and create an instance of the corresponding module class – without having to include the C++ header file containing module's class declaration into your source 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 first create the module itself, second, set parameter values and gate vector sizes, and then call the method that creates 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 starter messages, because the initializing code may insert new messages into the FES, and these messages should be processed *after* the starter message.

4.11.3 Creating modules

The first step, finding the factory object:

```
cModuleType *moduleType = findModuleType("WirelessNode");
```

Simplified form

`cModuleType` has a `createScheduleInit(const char *name, cModule *parentmod)` convenience function to get a module up and running in one step.

```
mod = modtype->createScheduleInit("node",this);
```

It does `create()` + `buildInside()` + `scheduleStart(now)` + `callInitialize()`.

This method can be used for both simple and compound modules. Its applicability is somewhat limited, however: because it does everything in one step, you do not have the chance to set parameters or gate sizes, and to connect gates before `initialize()` is called. (`initialize()` expects all parameters and gates to be in place and the network fully built when it is called.) Because of the above limitation, this function is mainly useful for creating basic simple modules.

Expanded form

If the previous simple form cannot be used. There are 5 steps:

1. find factory 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 factory object
cModuleType *moduleType = findModuleType("WirelessNode");

// create (possibly compound) module and build its submodules (if any)
cModule *module = moduleType->create("node", this);
module->buildInside();

// 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:

```
// create
cModuleType *moduleType = findModuleType("WirelessNode");
cModule *module = moduleType->create("node", this);

// set up parameters and gate sizes before we set up its submodules
module->par("address") = ++lastAddress;
module->setGateSize("in", 3);
module->setGateSize("out", 3);

// create internals, and schedule it
module->buildInside();
module->scheduleStart(simTime());
```

4.11.4 Deleting modules

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, the `deleteModule()` call does not return to the caller.

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.

4.11.5 Module deletion and finish()

When you delete a module *during simulation*, its `finish()` function is not called automatically (`deleteModule()` doesn't do it.) How the module was created doesn't play any role here: `finish()` gets called for *all* modules – at the end of the simulation. If a module doesn't live that long, `finish()` is not invoked, but you can still manually invoke it.

You can use the `callFinish()` function to arrange `finish()` to be called. It is usually not a good idea to invoke `finish()` directly. If you're deleting a compound module, `callFinish()` will recursively invoke `finish()` for all submodules, and if you're deleting a simple module from another module, `callFinish()` will do the context switch for the duration of the call. [6]

Example:

```
mod->callFinish();
mod->deleteModule();
```

4.11.6 Creating connections

Connections can be created using `cGate`'s `connectTo()` method. [7] `connectTo()` should be invoked on the source gate of the connection, and expects the destination gate pointer as an argument:

```
srcGate->connectTo(destGate);
```

The *source* and *destination* words correspond to the direction of the arrow in NED files.

As an example, we create two modules and connect them in both directions:

```
cModuleType *moduleType = findModuleType("TicToc");
cModule *a = modtype->createScheduleInit("a",this);
cModule *b = modtype->createScheduleInit("b",this);

a->gate("out")->connectTo(b->gate("in"));
b->gate("out")->connectTo(a->gate("in"));
```

`connectTo()` also accepts a channel object as an additional, optional argument. Channels are subclassed from `cChannel`. Almost always you'll want use an instance of `cBasicChannel` as channel – this is the one that supports delay, bit error rate and data rate. The channel object will be owned by the source gate of the connection, and you cannot reuse the same channel object with several connections.

`cBasicChannel` has `setDelay()`, `setError()` and `setDatarate()` methods to set up the channel attributes.

An example that sets up a channel with a delay:

```
cBasicChannel *channel = new cBasicChannel("channel");
channel->setDelay(0.001);

a->gate("out")->connectTo(b->gate("in"), channel); // a,b are modules
```

[6]The `finish()` function is even made `protected` in `cSimpleModule`, in order to discourage its invocation from other modules.

[7]The earlier `connect()` global functions that accepted two gates have been deprecated, and may be removed from further OMNeT++ releases.

4.11.7 Removing connections

The `disconnect()` method of `cGate` can be used to remove connections. This method has to be invoked on the *source* side of the connection. It also destroys the channel object associated with the connection, if one has been set.

```
srcGate->disconnect();
```

Chapter 5

Messages

5.1 Messages and packets

5.1.1 The cMessage class

`cMessage` is a central class in OMNeT++. 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 a string (`const char *`), which can be freely used by the simulation programmer. The message name appears in many places in Tkenv (for example, in animations), and it is generally very useful to choose a descriptive name. This attribute is inherited from `cObject` (see section 6.1.1).
- 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.
- 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 such as noting 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*, *control info* and *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

The `cMessage` constructor accepts several arguments. Most commonly, you would create a message using an *object name* (a `const char *` string) and a *message kind* (`int`):

```
cMessage *msg = new cMessage("MessageName", msgKind);
```

Both arguments are optional and initialize to the null string ("") and 0, so the following statements are also valid:

```
cMessage *msg = new cMessage();
cMessage *msg = new cMessage("MessageName");
```

It is a good idea to *always* use message names – they can be extremely useful when debugging or demonstrating your simulation.

Message kind is usually initialized with a symbolic constant (e.g. an *enum* value) which signals what the message object represents in the simulation (i.e. a data packet, a jam signal, a job, etc.) Please use *positive values or zero* only as message kind – negative values are reserved for use by the simulation kernel.

The `cMessage` constructor accepts further arguments too (*length*, *priority*, *bit error flag*), but for readability of the code it is best to set them explicitly via the `set...()` methods described below. Length and priority are integers, and the bit error flag is boolean.

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

```
msg->setKind( kind );
msg->setLength( length );
msg->setByteLength( lengthInBytes );
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. `setByteLength()` sets the same length field as `setLength()`, only the parameters gets internally multiplied by 8.

The values can be obtained by the following functions:

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

`byteLength()` also reads the length field as `length()`, but the result gets divided by 8 and rounded up.

Duplicating messages

It is often necessary to duplicate a message (for example, sending one and keeping a copy). This can be done in the same way as for any other OMNeT++ object:

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

or

```
cMessage *copy = 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.

5.1.2 Self-messages

Using a message as self-message

Messages are often used to represent events internal to a module, such as a periodically firing timer on expiry of a timeout. A message is termed *self-message* when it is used in such a scenario – otherwise self-messages are normal messages, of class `cMessage` or a class derived from it.

When a message is delivered to a module by the simulation kernel, you can call the `isSelfMessage()` method to determine if it is a self-message; it other words, if it was scheduled with `scheduleAt()` or was sent with one of the `send...()` methods. The `isScheduled()` method returns true if the message is currently scheduled. A scheduled message can also be cancelled (`cancelEvent()`).

```
bool isSelfMessage();
bool isScheduled();
```

The following methods return the time of creating and scheduling the message as well as its arrival time. While the message is scheduled, arrival time is the time it will be delivered to the module.

```
simtime_t creationTime()
simtime_t sendingTime();
simtime_t arrivalTime();
```

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.

5.1.3 Modelling packets

Arrival gate and time

The following methods can tell where the message came from and where it arrived (or will arrive if it is currently scheduled or under way.)

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

The following methods are just convenience functions which build on the ones above.

```
cModule *senderModule();
cGate *senderGate();
cGate *arrivalGate();
```

And there are further convenience functions to tell whether the message arrived on a specific gate given with id or name+index.

```
bool arrivedOn(int id);
bool arrivedOn(const char *gname, int gindex=0);
```

The following methods return message creation time and the last sending and arrival times.

```
simtime_t creationTime()
simtime_t sendingTime();
simtime_t arrivalTime();
```

Control info

One of the main application areas of OMNeT++ is the simulation of telecommunication networks. Here, protocol layers are usually implemented as modules which exchange packets. Packets themselves are represented by messages subclassed from `cMessage`.

However, communication between protocol layers requires sending additional information to be attached to packets. For example, a TCP implementation sending down a TCP packet to IP will want to specify the destination IP address and possibly other parameters. When IP passes up a packet to TCP after decapsulation from the IP header, it'll want to let TCP know at least the source IP address.

This additional information is represented by *control info* objects in OMNeT++. Control info objects have to be subclassed from `cPolymorphic` (a small footprint base class with no data members), and attached to the messages representing packets. `cMessage` has the following methods for this purpose:

```
void setControlInfo(cPolymorphic *controlInfo);
cPolymorphic *controlInfo();
cPolymorphic *removeControlInfo();
```

When a "command" is associated with the message sending (such as TCP OPEN, SEND, CLOSE, etc), the message kind field (`kind()`, `setKind()` methods of `cMessage`) should carry the command code. When the command doesn't involve a data packet (e.g. TCP CLOSE command), a dummy packet (empty `cMessage`) can be sent.

Identifying the protocol

In OMNeT++ protocol models, the protocol type is usually represented in the message subclass. For example, instances of class `IPv6Datagram` represent IPv6 datagrams and `EthernetFrame` represents Ethernet frames) and/or in the message kind value. The PDU type is usually represented as a field inside the message class.

The C++ `dynamic_cast` operator can be used to determine if a message object is of a specific protocol.

```
cMessage *msg = receive();
if (dynamic_cast<IPv6Datagram *>(msg) != NULL)
{
    IPv6Datagram *datagram = (IPv6Datagram *)msg;
    ...
}
```

5.1.4 Encapsulation

Encapsulating packets

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->setByteLength(2048);  // 2K
cMessage *tcpseg = new cMessage("tcp");
tcpseg->setByteLength(24);
tcpseg->encapsulate(userdata);
ev << tcpseg->byteLength() << endl; // --> 2048+24 = 2072
```

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.

Reference counting$^{New!}$

Since the 3.2 release, OMNeT++ implements reference counting of encapsulated messages, meaning that if you `dup()` a message that contains an encapsulated message, then the encapsulated message will not be duplicated, only a reference count incremented. Duplication of the encapsulated message is deferred until `decapsulate()` actually gets called. If the outer message gets deleted without its `decapsulate()` method ever being called, then the reference count of the encapsulated message simply gets decremented. The encapsulated message is deleted when its reference count reaches zero.

Reference counting can significantly improve performance, especially in LAN and wireless scenarios. For example, in the simulation of a broadcast LAN or WLAN, the IP, TCP and higher layer packets won't get duplicated (and then discarded without being used) if the MAC address doesn't match in the first place.

The reference counting mechanism works transparently. However, there is one implication: **one must not change anything in a message that is encapsulated into another!** That is, `encapsulat-edMsg()` should be viewed as if it returned a pointer to a read-only object (it returns a `const` pointer indeed), for quite obvious reasons: the encapsulated message may be shared between several messages, and any change would affect those other messages as well.

Encapsulating several messages

The `cMessage` class doesn't directly support adding more than one messages to a message object, but you can subclass `cMessage` and add the necessary functionality. (It is recommended that you use the message definition syntax 5.2 and customized messages 5.2.6 to be described later on in this chapter – it can spare you some work.)

You can store the messages in a fixed-size or a dynamically allocated array, or you can use STL classes like `std::vector` or `std::list`. There is one additional "trick" that you might not expect: your message class has to **take ownership** of the inserted messages, and **release** them when they are removed from the message. These are done via the `take()` and `drop()` methods. Let us see an example which assumes you have added to the class an `std::list` member called `messages` that stores message pointers:

```
void MessageBundleMessage::insertMessage(cMessage *msg)
{
    take(msg);  // take ownership
    messages.push_back(msg);  // store pointer
}

void MessageBundleMessage::removeMessage(cMessage *msg)
{
    messages.remove(msg);  // remove pointer
    drop(msg);  // release ownership
}
```

You will also have to provide an `operator=()` method to make sure your message objects can be copied and duplicated properly – this is something often needed in simulations (think of broadcasts and retransmissions!). Section 6.11 contains more info about the things you need to take care of when deriving new classes.

5.1.5 Attaching parameters and objects

If you want to add parameters or objects to a message, the preferred way to do that is via message definitions, described in chapter 5.2.

Attaching objects

The `cMessage` class has an internal `cArray` object which can carry objects. Only objects that are derived from `cObject` (most OMNeT++ classes are so) can be attached. The `addObject()`, `getObject()`, `hasObject()`, `removeObject()` methods use the object name as the key to the array. An example:

```
cLongHistogram *pklenDistr = new cLongHistogram("pklenDistr");
msg->addObject( pklenDistr );
...
if (msg->hasObject("pklenDistr"))
{
   cLongHistogram *pklenDistr =
       (cLongHistogram *) msg->getObject("pklenDistr");
   ...
}
```

You should take care that names of the attached objects do not clash with each other or with `cPar` parameter names (see next section). If you do not attach anything to the message and do not call the `parList()` function, the internal `cArray` object will not be created. This saves 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));
```

Attaching parameters

The preferred way of extending messages with new data fields is to use message definitions (see section 5.2).

The old, deprecated way of adding new fields to messages is via attaching `cPar` objects. There are several downsides of this approach, the worst being large memory and execution time overhead. `cPar`'s are heavyweight and fairly complex objects themselves. It has been reported that using `cPar` message parameters might account for a large part of execution time, sometimes as much as 80%. Using `cPar`s is also error-prone because `cPar` objects have to be added dynamically and individually to each message object. In contrast, subclassing benefits from static type checking: if you mistype the name of a field in the C++ code, already the compiler can detect the mistake.

However, if you still need to use cPars, here's a short summary how you can do it. You add a new parameter to the message with the `addPar()` member function, and get back a reference to the parameter object with the `par()` member function. `hasPar()` tells you if the message has a given parameter or not. 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.

Example:

```
msg->addPar("destAddr");
msg->par("destAddr") = 168;
...
long destAddr = msg->par("destAddr");
```

5.2 Message definitions

5.2.1 Introduction

In practice, you'll need to add various fields to `cMessage` to make it useful. For example, if you're modelling packets in communication networks, you need to have a way to store protocol header fields in message objects. Since the simulation library is written in C++, the natural way of extending `cMessage` is via subclassing it. However, because for each field you need to write at least three things (a private data member, a getter and a setter method), and the resulting class has to integrate with the simulation framework, writing the necessary C++ code can be a tedious and time-consuming task.

OMNeT++ offers a more convenient way called *message definitions*. Message definitions provide a very compact syntax to describe message contents. C++ code is automatically generated from message definitions, saving you a lot of typing.

A common source of complaint about code generators in general is lost flexibility: if you have a different idea how the generated code should look like, there's little you can do about it. In OMNeT++, however, there's nothing to worry about: you can customize the generated class to any extent you like. Even if you decide to heavily customize the generated class, message definitions still save you a great deal of manual work.

The message subclassing feature in OMNeT++ is still somewhat experimental, meaning that:

- The message description syntax and features may slightly change in the future, based on feedback from the community;
- The compiler that translates message descriptions into C++ is a perl script `opp_msgc`. This is a temporary solution until the C++-based `nedtool` is finished.

The subclassing approach for adding message parameters was originally suggested by Nimrod Mesika.

The first message class

Let us begin with a simple example. Suppose that you need message objects to carry source and destination addresses as well as a hop count. You could write a `mypacket.msg` file with the following contents:

```
message MyPacket
{
    fields:
       int srcAddress;
       int destAddress;
       int hops = 32;
};
```

The task of the *message subclassing compiler* is to generate C++ classes you can use from your models as well as "reflection" classes that allow Tkenv to inspect these data stuctures.

If you process `mypacket.msg` with the message subclassing compiler, it will create the following files for you: `mypacket_m.h` and `mypacket_m.cc`. `mypacket_m.h` contains the declaration of the `MyPacket` C++ class, and it should be included into your C++ sources where you need to handle `MyPacket` objects.

The generated `mypacket_m.h` will contain the following class declaration:

```
class MyPacket : public cMessage {
    ...
    virtual int getSrcAddress() const;
    virtual void setSrcAddress(int srcAddress);
    ...
};
```

So in your C++ file, you could use the `MyPacket` class like this:

```
#include "mypacket_m.h"

...
MyPacket *pkt = new MyPacket("pkt");
pkt->setSrcAddress( localAddr );
...
```

The `mypacket_m.cc` file contains implementation of the generated `MyPacket` class, as well as "reflection" code that allows you to inspect these data stuctures in the Tkenv GUI. The `mypacket_m.cc` file should be compiled and linked into your simulation. (If you use the `opp_makemake` tool to generate your makefiles, the latter will be automatically taken care of.)

What is message subclassing *not?*

There might be some confusion around the purpose and concept of message definitions, so it seems to be a good idea to deal with them right here.

It is ***not:***

- ... *an attempt to reproduce the functionality of C++ with another syntax.* Do not look for complex C++ types, templates, conditional compilation, etc. Also, it defines *data* only (or rather: an interface to access data) – not any kind of active behaviour.
- ... *a generic class generator.* This is meant for defining message contents, and data structure you put in messages. Defining methods is not supported on purpose. Also, while you can probably (ab)use the syntax to generate classes and structs used internally in simple modules, this is probably not a good idea.

The goal is to define the *interface* (getter/setter methods) of messages rather than their implementations in C++. A simple and straightforward implementation of fields is provided – if you'd like a different internal representation for some field, you can have it by customizing the class.

There are questions you might ask:

- *Why doesn't it support `std::vector` and other STL classes?* Well, it does. Message definitions focus on the interface (getter/setter methods) of the classes, optionally leaving the implementation to you – so you can implement fields (dynamic array fields) using `std::vector`. (This aligns with the idea behind STL – it was designed to be *nuts and bolts* for C++ programs).
- *Why does it support C++ data types and not octets, bytes, bits, etc..?* That would restrict the scope of message definitions to networking, and OMNeT++ wants to support other application areas as well. Furthermore, the set of necessary concepts to be supported is probably not bounded, there would always be new data types to be adopted.
- *Why no embedded classes?* Good question. As it does not conflict with the above principles, it might be added someday.

The following sections describe the message syntax and features in detail.

5.2.2 Declaring enums

An `enum {..}` generates a normal C++ enum, plus creates an object which stores text representations of the constants. The latter makes it possible to display symbolic names in Tkenv. An example:

```
enum ProtocolTypes
{
   IP = 1;
   TCP = 2;
};
```

Enum values need to be unique.

5.2.3 Message declarations

Basic use

You can describe messages with the following syntax:

```
message FooPacket
{
    fields:
        int sourceAddress;
        int destAddress;
        bool hasPayload;
};
```

Processing this description with the message compiler will produce a C++ header file with a generated class, `FooPacket`. `FooPacket` will be a subclass of `cMessage`.

For each field in the above description, the generated class will have a protected data member, a getter and a setter method. The names of the methods will begin with `get` and `set`, followed by the field name with its first letter converted to uppercase. Thus, `FooPacket` will contain the following methods:

```
virtual int getSourceAddress() const;
virtual void setSourceAddress(int sourceAddress);

virtual int getDestAddress() const;
virtual void setDestAddress(int destAddress);

virtual bool getHasPayload() const;
virtual void setHasPayload(bool hasPayload);
```

Note that the methods are all declared `virtual` to give you the possibility of overriding them in subclasses.

Two constructors will be generated: one that optionally accepts object name and (for `cMessage` subclasses) message kind, and a copy constructor:

```
FooPacket(const char *name=NULL, int kind=0);
FooPacket(const FooPacket& other);
```

Appropriate assignment operator (`operator=()`) and `dup()` methods will also be generated.

Data types for fields are not limited to `int` and `bool`. You can use the following primitive types (i.e. primitive types as defined in the C++ language):

- `bool`
- `char`, `unsigned char`
- `short`, `unsigned short`
- `int`, `unsigned int`
- `long`, `unsigned long`
- `double`

Field values are initialized to zero.

Initial values

You can initialize field values with the following syntax:

```
message FooPacket
{
   fields:
        int sourceAddress = 0;
        int destAddress = 0;
        bool hasPayload = false;
};
```

Initialization code will be placed in the constructor of the generated class.

Enum declarations

You can declare that an `int` (or other integral type) field takes values from an enum. The message compiler can than generate code that allows Tkenv display the symbolic value of the field.

Example:

```
message FooPacket
{
  fields:
      int payloadType enum(PayloadTypes);
};
```

The enum has to be declared separately in the message file.

Fixed-size arrays

You can specify fixed size arrays:

```
message FooPacket
{
    fields:
        long route[4];
};
```

The generated getter and setter methods will have an extra `k` argument, the array index:

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, long route);
```

If you call the methods with an index that is out of bounds, an exception will be thrown.

Dynamic arrays

If the array size is not known in advance, you can declare the field to be a dynamic array:

```
message FooPacket
{
   fields:
       long route[];
};
```

In this case, the generated class will have two extra methods in addition to the getter and setter methods: one for setting the array size, and another one for returning the current array size.

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, long route);
virtual unsigned getRouteArraySize() const;
virtual void setRouteArraySize(unsigned n);
```

The `set...ArraySize()` method internally allocates a new array. Existing values in the array will be preserved (copied over to the new array.)

The default array size is zero. This means that you need to call the `set...ArraySize()` before you can start filling array elements.

String members

You can declare string-valued fields with the following syntax:

```
message FooPacket
{
   fields:
       string hostName;
};
```

The generated getter and setter methods will return and accept `const char*` pointers:

```
virtual const char *getHostName() const;
virtual void setHostName(const char *hostName);
```

The generated object will have its own copy of the string.

NOTE: a string member is different from a character array, which is treated as an array of any other type. For example,

```
message FooPacket
{
   fields:
       char chars[10];
};
```

will generate the following methods:

```
virtual char getChars(unsigned k);
virtual void setChars(unsigned k, char a);
```

5.2.4 Inheritance, composition

So far we have discussed how to add fields of primitive types (`int`, `double`, `char`, ...) to `cMessage`. This might be sufficient for simple models, but if you have more complex models, you'll probably need to:

- set up a hierarchy of message (packet) classes, that is, not only subclass from `cMessage` but also from your own message classes;

- use not only primitive types as fields, but also structs, classes or typedefs. Sometimes you'll want to use a C++ type present in an already existing header file, another time you'll want a struct or class to be generated by the message compiler so that you can benefit from Tkenv inspectors.

The following section describes how to do this.

Inheritance among message classes

By default, messages are subclassed from `cMessage`. However, you can explicitly specify the base class using the `extends` keyword:

```
message FooPacket extends FooBase
{
    fields:
        ...
};
```

For the example above, the generated C++ code will look like:

```
class FooPacket : public FooBase { ... };
```

Inheritance also works for structs and classes (see next sections for details).

Defining classes

Until now we have used the `message` keyword to define classes, which implies that the base class is `cMessage`, either directly or indirectly.

But as part of complex messages, you'll need structs and other classes (rooted or not rooted in `cObject`) as building blocks. Classes can be created with the `class` class keyword; structs we'll cover in the next section.

The syntax for defining classes is almost the same as defining messages, only the `class` keyword is used instead of `message`.

Slightly different code is generated for classes that are rooted in `cObject` than for those which are not. If there is no `extends`, the generated class will not be derived from `cObject`, thus it will not have `name()`, `className()`, etc. methods. To create a class with those methods, you have to explicitly write `extends cObject`.

```
class MyClass extends cObject
{
    fields:
        ...
};
```

Defining plain C structs

You can define C-style structs to be used as fields in message classes, "C-style" meaning "containing only data and no methods". (Actually, in the C++ a struct can have methods, and in general it can do anything a class can.)

The syntax is similar to that of defining messages:

```
struct MyStruct
{
    fields:
        char array[10];
        short version;
};
```

However, the generated code is different. The generated struct has no getter or setter methods, instead the fields are represented by public data members. For the definition above, the following code is generated:

```
// generated C++
struct MyStruct
{
    char array[10];
    short version;
};
```

A struct can have primitive types or other structs as fields. It cannot have string or class as field.

Inheritance is supported for structs:

```
struct Base
{
    ...
};

struct MyStruct extends Base
{
    ...
};
```

But because a struct has no member functions, there are limitations:

- dynamic arrays are not supported (no place for the array allocation code)
- "generation gap" or abstract fields (see later) cannot be used, because they would build upon virtual functions.

Using structs and classes as fields

In addition to primitive types, you can also use other structs or objects as a field. For example, if you have a struct named `IPAddress`, you can write the following:

```
message FooPacket
{
   fields:
       IPAddress src;
};
```

The `IPAddress` structure must be known in advance to the message compiler; that is, it must either be a struct or class defined earlier in the message description file, or it must be a C++ type with its header file included via `cplusplus {{...}}` and its type announced (see Announcing C++ types).

The generated class will contain an `IPAddress` data member (that is, **not** a pointer to an `IPAddress`). The following getter and setter methods will be generated:

```
virtual const IPAddress& getSrc() const;
virtual void setSrc(const IPAddress& src);
```

Pointers

Not supported yet.

5.2.5 Using existing C++ types

Announcing C++ types

If you want to use one of your own types (a class, struct or typedef, declared in a C++ header) in a message definition, you have to announce those types to the message compiler. You also have to make sure that your header file gets included into the generated `_m.h` file so that the C++ compiler can compile it.

Suppose you have an `IPAddress` structure, defined in an `ipaddress.h` file:

```
// ipaddress.h
struct IPAddress {
    int byte0, byte1, byte2, byte3;
};
```

To be able to use `IPAddress` in a message definition, the message file (say `foopacket.msg`) should contain the following lines:

```
cplusplus {{
#include "ipaddress.h"
}};

struct IPAddress;
```

The effect of the first three lines is simply that the `#include` statement will be copied into the generated `foopacket_m.h` file to let the C++ compiler know about the `IPAddress` class. The message compiler itself will not try to make sense of the text in the body of the `cplusplus {{ ... }}` directive.

The next line, `struct IPAddress`, tells the message compiler that `IPAddress` is a C++ struct. This information will (among others) affect the generated code.

Classes can be announced using the `class` keyword:

```
class cSubQueue;
```

The above syntax assumes that the class is derived from `cObject` either directly or indirectly. If it is not, the `noncobject` keyword should be used:

```
class noncobject IPAddress;
```

The distinction between classes derived and not derived from `cObject` is important because the generated code differs at places. The generated code is set up so that if you incidentally forget the `noncobject` keyword (and thereby mislead the message compiler into thinking that your class is rooted in `cObject` when in fact it is not), you'll get a C++ compiler error in the generated header file.

5.2.6 Customizing the generated class

The Generation Gap pattern

Sometimes you need the generated code to do something more or do something differently than the version generated by the message compiler. For example, when setting a integer field named `payloadLength`, you might also need to adjust the packet length. That is, the following default (generated) version of the `setPayloadLength()` method is not suitable:

```
void FooPacket::setPayloadLength(int payloadLength)
{
    this->payloadLength = payloadLength;
}
```

Instead, it should look something like this:

```
void FooPacket::setPayloadLength(int payloadLength)
{
    int diff = payloadLength - this->payloadLength;
    this->payloadLength = payloadLength;
    setLength(length() + diff);
}
```

According to common belief, the largest drawback of generated code is that it is difficult or impossible to fulfill such wishes. Hand-editing of the generated files is worthless, because they will be overwritten and changes will be lost in the code generation cycle.

However, object oriented programming offers a solution. A generated class can simply be customized by subclassing from it and redefining whichever methods need to be different from their generated versions. This practice is known as the *Generation Gap* design pattern. It is enabled with the following syntax:

```
message FooPacket
{
   properties:
       customize = true;
   fields:
       int payloadLength;
};
```

The `properties` section within the message declaration contains meta-info that affects how generated code will look like. The customize property enables the use of the Generation Gap pattern.

If you process the above code with the message compiler, the generated code will contain a `FooPacket_Base` class instead of `FooPacket`. The idea is that you have to subclass from `FooPacket_Base` to produce `FooPacket`, while doing your customizations by redefining the necessary methods.

```
class FooPacket_Base : public cMessage
{
  protected:
    int src;
    // make constructors protected to avoid instantiation
    FooPacket_Base(const char *name=NULL);
    FooPacket_Base(const FooPacket_Base& other);
  public:
    ...
    virtual int getSrc() const;
    virtual void setSrc(int src);
};
```

There is a minimum amount of code you have to write for `FooPacket`, because not everything can be pre-generated as part of `FooPacket_Base`, e.g. constructors cannot be inherited. This minimum code is the following (you'll find it the generated C++ header too, as a comment):

```
class FooPacket : public FooPacket_Base
{
  public:
    FooPacket(const char *name=NULL) : FooPacket_Base(name) {}
    FooPacket(const FooPacket& other) : FooPacket_Base(other) {}
    FooPacket& operator=(const FooPacket& other)
        {FooPacket_Base::operator=(other); return *this;}
    virtual cPolymorphic *dup() {return new FooPacket(*this);}
};

Register_Class(FooPacket);
```

Note that it is important that you redefine `dup()` and provide an assignment operator (`operator=()`).

So, returning to our original example about payload length affecting packet length, the code you'd write is the following:

```
class FooPacket : public FooPacket_Base
{
    // here come the mandatory methods: constructor,
    // copy contructor, operator=(), dup()
    // ...

    virtual void setPayloadLength(int newlength);
}

void FooPacket::setPayloadLength(int newlength)
{
    // adjust message length
    setLength(length()-getPayloadLength()+newlength);

    // set the new length
    FooPacket_Base::setPayloadLength(newlength);
}
```

Abstract fields

The purpose of abstract fields is to let you to override the way the value is stored inside the class, and still benefit from inspectability in Tkenv.

For example, this is the situation when you want to store a bitfield in a single `int` or `short`, and still you want to present bits as individual packet fields. It is also useful for implementing computed fields.

You can declare any field to be abstract with the following syntax:

```
message FooPacket
{
   properties:
       customize = true;
   fields:
       abstract bool urgentBit;
};
```

For an `abstract` field, the message compiler generates no data member, and generated getter/setter methods will be pure virtual:

```
virtual bool getUrgentBit() const = 0;
virtual void setUrgentBit(bool urgentBit) = 0;
```

Usually you'll want to use abstract fields together with the Generation Gap pattern, so that you can immediately redefine the abstract (pure virtual) methods and supply your implementation.

5.2.7 Using STL in message classes

You may want to use STL `vector` or `stack` classes in your message classes. This is possible using abstract fields. After all, `vector` and `stack` are representations of a *sequence* – same abstraction as dynamic-size vectors. That is, you can declare the field as `abstract T fld[]`, and provide an underlying implementation using `vector<T>`. You can also add methods to the message class that invoke `push_back()`, `push()`, `pop()`, etc. on the underlying STL object.

See the following message declaration:

```
struct Item
{
    fields:
        int a;
        double b;
}

message STLMessage
{
   properties:
       customize=true;
   fields:
       abstract Item foo[]; // will use vector<Item>
       abstract Item bar[]; // will use stack<Item>
}
```

If you compile the above, in the generated code you'll only find a couple of abstract methods for `foo` and `bar`, no data members or anything concrete. You can implement everything as you like. You can write the following C++ file then to implement `foo` and `bar` with `std::vector` and `std::stack`:

```
#include <vector>
#include <stack>
#include "stlmessage_m.h"


class STLMessage : public STLMessage_Base
{
  protected:
    std::vector<Item> foo;
    std::stack<Item> bar;

  public:
    STLMessage(const char *name=NULL, int kind=0) : STLMessage_Base(name,kind) {}
    STLMessage(const STLMessage& other) : STLMessage_Base(other.name()) {operator=(othe
    STLMessage& operator=(const STLMessage& other) {
        if (&other==this) return *this;
        STLMessage_Base::operator=(other);
        foo = other.foo;
        bar = other.bar;
        return *this;
    }
    virtual cPolymorphic *dup() {return new STLMessage(*this);}

    // foo methods
    virtual void setFooArraySize(unsigned int size) {}
    virtual unsigned int getFooArraySize() const {return foo.size();}
    virtual Item& getFoo(unsigned int k) {return foo[k];}
    virtual void setFoo(unsigned int k, const Item& afoo) {foo[k]=afoo;}
    virtual void addToFoo(const Item& afoo) {foo.push_back(afoo);}

    // bar methods
    virtual void setBarArraySize(unsigned int size) {}
    virtual unsigned int getBarArraySize() const {return bar.size();}
    virtual Item& getBar(unsigned int k) {throw new cRuntimeException("sorry");}
    virtual void setBar(unsigned int k, const Item& bar) {throw new cRuntimeException("
    virtual void barPush(const Item& abar) {bar.push(abar);}
    virtual void barPop() {bar.pop();}
    virtual Item& barTop() {return bar.top();}
};

Register_Class(STLMessage);
```

Some additional notes:

1. `setFooArraySize()`, `setBarArraySize()` are redundant.
2. `getBar(int k)` cannot be implemented in a straightforward way (`std::stack` does not support accessing elements by index). It could still be implemented in a less efficient way using STL iterators, and efficiency does not seem to be major problem because only Tkenv is going to invoke this function.
3. `setBar(int k, const Item&)` could not be implemented, but this is not particularly a problem. The exception will materialize in a Tkenv error dialog when you try to change the field value.

You may regret that the STL `vector`/`stack` are not directly exposed. Well you could expose them (by adding a `vector<Item>& getFoo() {return foo;}` method to the class) but this is probably not a good idea. STL itself was purposefully designed with a low-level approach, to provide "nuts and bolts" for C++ programming, and STL is better used in other classes for internal representation of data.

5.2.8 Summary

This section attempts to summarize the possibilities.

You can generate:

- classes rooted in `cObject`
- messages (default base class is `cMessage`)
- classes not rooted in `cObject`
- plain C structs

The following data types are supported for fields:

- primitive types: `bool`, `char`, `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `double`
- `string`, a dynamically allocated string, presented as `const char *`
- fixed-size arrays of the above types
- structs, classes (both rooted and not rooted in `cObject`), declared with the message syntax or externally in C++ code
- variable-sized arrays of the above types (stored as a dynamically allocated array plus an integer for the array size)

Further features:

- fields initialize to zero (except struct members)
- fields initializers can be specified (except struct members)
- assigning `enum`s to variables of integral types.
- inheritance
- customizing the generated class via subclassing (*Generation Gap* pattern)
- abstract fields (for nonstandard storage and calculated fields)

Generated code (all generated methods are `virtual`, although this is not written out in the following table):

Field declaration	**Generated code**

primitive types
 	
`double field;`
	
	`double getField();`
	`void setField(double d);`

string type
 	
`string field;`
	
	`const char *getField();`
	`void setField(const char *);`

fixed-size arrays
 	
`double field[4];`
	
	`double getField(unsigned k);`
	`void setField(unsigned k, double d);`
	`unsigned getFieldArraySize();`

dynamic arrays
`double field[];`
	`void setFieldArraySize(unsigned n);`
	`unsigned getFieldArraySize();`
	`double getField(unsigned k);`
	`void setField(unsigned k, double d);`

customized class
`class Foo {`
`  properties:`
`    customize=true;`
	`class Foo_Base { ... };`
	and you have to write:
	`class Foo : public Foo_Base {`
	`   ...`
	`};`

abstract fields
`abstract double field;`
	`double getField() = 0;`
	`void setField(double d) = 0;`

Example simulations

Several of the example simulations (Token Ring, Dyna, Hypercube) use message definitions. For example, in Dyna you'll find this:

- `dynapacket.msg` defines `DynaPacket` and `DynaDataPacket`;
- `dynapacket_m.h` and `dynapacket_m.cc` are produced by the message subclassing compiler from it, and they contain the generated `DynaPacket` and `DynaDataPacket` C++ classes (plus code for Tkenv inspectors);
- other model files (`client.cc`, `server.cc`, ...) use the generated message classes

5.2.9 What else is there in the generated code?

In addition to the message class and its implementation, the message compiler also generates reflection code which makes it possible to inspect message contents in Tkenv. To illustrate why this is necessary, suppose you manually subclass `cMessage` to get a new message class. You could write the following: [1]

```
class RadioMsg : public cMessage
{
  public:
    int freq;
    double power;
    ...
};
```

Now it is possible to use `RadioMsg` in your simple modules:

[1]Note that the code is only for illustration. In real code, `freq` and `power` should be private members, and getter/setter methods should exist to access them. Also, the above class definition misses several member functions (constructor, assignment operator, etc.) that should be written.

```
RadioMsg *msg = new RadioMsg();
msg->freq = 1;
msg->power = 10.0;
...
```

You'd notice one drawback of this solution when you try to use Tkenv for debugging. While `cPar`-based message parameters can be viewed in message inspector windows, fields added via subclassing do not appear there. The reason is that Tkenv, being just another C++ library in your simulation program, doesn't know about your C++ instance variables. The problem cannot be solved entirely within Tkenv, because C++ does not support "reflection" (extracting class information at runtime) like for example Java does.

There is a solution however: one can supply Tkenv with missing "reflection" information about the new class. Reflection info might take the form of a separate C++ class whose methods return information about the `RadioMsg` fields. This descriptor class might look like this:

```
class RadioMsgDescriptor : public Descriptor
{
  public:
    virtual int getFieldCount() {return 2;}

    virtual const char *getFieldName(int k) {
        const char *fieldname[] = {"freq", "power";}
        if (k<0 || k>=2) return NULL;
        return fieldname[k];
    }

    virtual double getFieldAsDouble(RadioMsg *msg, int k) {
        if (k==0) return msg->freq;
        if (k==1) return msg->power;
        return 0.0; // not found
    }
    //...
};
```

Then you have to inform Tkenv that a `RadioMsgDescriptor` exists and that it should be used whenever Tkenv finds messages of type `RadioMsg` (as it is currently implemented, whenever the object's `className()` method returns `"RadioMsg"`). So when you inspect a `RadioMsg` in your simulation, Tkenv can use `RadioMsgDescriptor` to extract and display the values of the `freq` and `power` variables.

The actual implementation is somewhat more complicated than this, but not much.

Chapter 6

The Simulation Library

OMNeT++ has an extensive C++ class library which you can use when implementing simple modules. Parts of the class library have already been covered in the previous chapters:

- the message class `cMessage` (chapter 5)
- sending and receiving messages, scheduling and canceling events, terminating the module or the simulation (section 4.6)
- access to module gates and parameters via `cModule` member functions (sections 4.7 and 4.8)
- accessing other modules in the network (section 4.9)
- dynamic module creation (section 4.11)

This chapter discusses the rest of the simulation library:

- random number generation: `normal()`, `exponential()`, etc.
- module parameters: `cPar` class
- storing data in containers: the `cArray` and `cQueue` classes
- routing support and discovery of network topology: `cTopology` class
- recording statistics into files: `cOutVector` class
- collecting simple statistics: `cStdDev` and `cWeightedStddev` classes
- distribution estimation: `cLongHistogram`, `cDoubleHistogram`, `cVarHistogram`, `cPSquare`, `cK-Split` classes
- making variables inspectable in the graphical user interface (Tkenv): the `WATCH()` macros
- 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

6.1.1 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)

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

6.1.2 Setting and getting attributes

Member functions that set and query object attributes follow consistent naming. The setter member function has the form `setFoo(...)` and its getter counterpart is named `foo()`. (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();
```

6.1.3 className()

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

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

6.1.4 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). An example:

```
cMessage *timeoutMsg = new cMessage("timeout");
```

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

```
timeoutMsg->setName("timeout");
```

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

```
const char *name = timeoutMsg->name(); // --> "timeout"
```

For convenience and efficiency reasons, the empty string `""` and `NULL` are treated as equivalent by library objects. That is, `""` is stored as `NULL` but returned as `""`. If you create a message object with either `NULL` or `""` as name string, it will be stored as `NULL` and `name()` will return a pointer to a static `""`.

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

6.1.5 fullName() and fullPath()

Objects have two more member functions which return strings based on object names: `fullName()` and `fullPath()`. For gates and modules which are part of gate or module vectors, `fullName()` returns the name with the index in brackets. That is, for a module `node[3]` in the submodule vector `node[10]` `name()` returns `"node"`, and `fullName()` returns `"node[3]"`. For other objects, `fullName()` is the same as `name()`.

`fullPath()` returns `fullName()`, prepended with the parent or owner object's `fullPath()` and separated by a dot. That is, if the `node[3]` module above is in the compound module `"net.subnet1"`, its `fullPath()` method will return `"net.subnet1.node[3]"`.

```
ev << this->name();     // --> "node"
ev << this->fullName(); // --> "node[3]"
ev << this->fullPath(); // --> "net.subnet1.node[3]"
```

`className()`, `fullName()` and `fullPath()` are extensively used on the graphical runtime environment Tkenv, and also appear in error messages.

`name()` and `fullName()` return `const char *` pointers, and `fullPath()` returns `std::string`. This makes no difference with `ev«`, but when `fullPath()` is used as a `"%s"` argument to `sprintf()` you have to write `fullPath().c_str()`.

```
char buf[100];
sprintf("msg is '%80s'", msg->fullPath().c_str()); // note c_str()
```

6.1.6 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 by 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. This is a deep copy: object contained in the object will also be duplicated if necessary. `operator=()` does not copy the name string – this task is done by the copy constructor.

6.1.7 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 (cQueue::Iterator queueIter(queue); !queueIter.end(); queueIter++)
{
    cObject *containedObject = queueIter();
}
```

6.1.8 Error handling

When library objects detect an error condition, they throw a C++ exception. This exception is then caught by the simulation environment which pops up an error dialog or displays the error message.

At times it can be useful to be able stop the simulation at the place of the error (just before the exception is thrown) and use a C++ debugger to look at the stack trace and examine variables. Enabling the `debug-on-errors` ini file entry lets you do that – check it in section 8.2.6 .

If you detect an error condition in your code, you can stop the simulation with an error message using the `opp_error()` function. `opp_error()`'s argument list works like `printf()`: the first argument is a format string which can contain `"%s"`, `"%d"` etc, filled in using subsequent arguments.

An example:

```
if (msg->controlInfo()==NULL)
    opp_error("message (%s)%s has no control info attached",
              msg->className(), msg->name());
```

6.2 Logging from modules

The logging feature will be used extensively in the code examples, we introduce it here.

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 " << seqNum << endl;
ev << "queue full, discarding packet\n";
```

An alternative solution is `ev.printf()`:

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

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 `omnetpp.ini` so that it doesn't slow down simulation when it is not needed.) In Tkenv, the runtime GUI, you can open a text output window for every module. It is not recommended that you use `printf()` or `cout` to print messages – `ev` output can be controlled much better from `omnetpp.ini` and it is more convenient to view, using Tkenv.

One can save CPU cycles by making logging statements conditional on whether the output actually gets displayed or recorded anywhere. The `ev.disabled()` call returns true when `ev`« output is disabled, such as in Tkenv or Cmdenv "express" mode. Thus, one can write code like this:

```
if (!ev.disabled())
    ev << "Packet " << msg->name() << " received\n";
```

A more sophisticated implementation of the same idea is to define an `EV` macro which can be used in logging statements instead of `ev`. The definition:

```
#define EV  ev.disabled()?std::cout:ev
```

And after that, one would simply write `EV`« instead of `ev`«.

```
EV << "Packet " << msg->name() << " received\n";
```

The slightly tricky definition of `EV` makes use of the fact that the « operator binds looser than `?:`.

6.3 Simulation time conversion

Simulation time is represented by the type `simtime_t` which is a typedef to `double`. OMNeT++ provides utility functions, which convert `simtime_t` to a printable string (`"3s 130ms 230us"`) and vica versa.

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

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

The `simtimeToStrShort()` is similar to `simtimeToStr()`, but its output is more concise.

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 something extra";

simtime_t t = strToSimtime0(s); // now s points to "and something extra"
```

6.4 Generating random numbers

Random numbers in simulation are never random. Rather, they are produced using deteministic algorithms. Algorithms take a *seed* value and perform some deterministic calculations on them to produce a "random" number and the next seed. Such algorithms and their implementations are called *random number generators* or RNGs, or sometimes pseudo random number generators or PRNGs to highlight their deterministic nature. [1]

Starting from the same seed, RNGs always produce the same sequence of random numbers. This is a useful property and of great importance, because it makes simulation runs repeatable.

RNGs produce uniformly distributed integers in some range, usually between 0 or 1 and 2^{32} or so. Mathematical transformations are used to produce random variates from them that correspond to specific distributions.

6.4.1 Random number generators

Mersenne Twister

By default, OMNeT++ uses the Mersenne Twister RNG (MT) by M. Matsumoto and T. Nishimura [MN98]. MT has a period of $2^{19937} - 1$, and 623-dimensional equidistribution property is assured. MT is also very fast: as fast or faster than ANSI C's `rand()`.

[1]There are real random numbers as well, see e.g. http://www.random.org/, http://www.comscire.com, or the Linux */dev/random* device. For non-random numbers, try www.noentropy.net.

The "minimal standard" RNG

OMNeT++ releases prior to 3.0 used a linear congruential generator (LCG) with a cycle length of $2^{31} - 2$, described in [Jai91], pp. 441-444,455. This RNG is still available and can be selected from `omnetpp.ini` (Chapter 8). This RNG is only suitable for small-scale simulation studies. As shown by Karl Entacher et al. in [EHW02], the cycle length of about 2^{31} is too small (on todays fast computers it is easy to exhaust all random numbers), and the structure of the generated "random" points is too regular. The [Hel98] paper provides a broader overview of issues associated with RNGs used for simulation, and it is well worth reading. It also contains useful links and references on the topic.

The Akaroa RNG

When you execute simulations under Akaroa control (see section 8.10), you can also select Akaroa's RNG as the RNG underlying for the OMNeT++ random number functions. The Akaroa RNG also has to be selected from `omnetpp.ini` (section 8.6).

Other RNGs

OMNeT++ allows plugging in your own RNGs as well. This mechanism, based on the `cRNG` interface, is described in section 13.5.3. For example, one candidate to include could be L'Ecuyer's CMRG [LSCK02] which has a period of about 2^{191} and can provide a large number of *guaranteed* independent streams.

6.4.2 Random number streams, RNG mapping

Simulation programs may consume random numbers from several streams, that is, from several independent RNG instances. For example, if a network simulation uses random numbers for generating packets and for simulating bit errors in the transmission, it might be a good idea to use different random streams for both. Since the seeds for each stream can be configured independently, this arrangement would allow you to perform several simulation runs with the same traffic but with bit errors occurring in different places. A simulation technique called *variance reduction* is also related to the use of different random number streams.

It is also important that different streams and also different simulation runs use non-overlapping series of random numbers. Overlap in the generated random number sequences can introduce unwanted correlation in your results.

The number of random number streams as well as seeds for the individual streams can be configured in `omnetpp.ini` (section 8.6). For the "minimal standard RNG", the `seedtool` program can be used for selecting good seeds (section 8.6.6).

In OMNeT++, streams are identified with RNG numbers. The RNG numbers used in simple modules may be *arbitrarily mapped* to the actual random number streams (actual RNG instances) from `omnetpp.ini` (section 8.6). The mapping allows for great flexibility in RNG usage and random number streams configuration – even for simulation models which were not written with RNG awareness.

6.4.3 Accessing the RNGs

The `intrand(n)` function generates random integers in the range $[0, n-1]$, and `dblrand()` generates a random double on $[0, 1)$. These functions simply wrap the underlying RNG objects. Examples:

```
int dice = 1 + intrand(6); // result of intrand(6) is in the range 0..5
double p = dblrand();      // dblrand() produces numbers in [0,1)
```

They also have a counterparts that use generator k:

```
int dice = 1 + genk_intrand(k,6); // uses generator k
double prob = genk_dblrand(k);    // ""
```

The underlying RNG objects are subclassed from `cRNG`, and they can be accessed via `cModule`'s `rng()` method. The argument to `rng()` is a local RNG number which will undergo RNG mapping.

```
cRNG *rng1 = rng(1);
```

`cRNG` contains the methods implementing the above `intrand()` and `dblrand()` functions. The `cRNG` interface also allows you to access the "raw" 32-bit random numbers generated by the RNG and to learn their ranges (`intRand()`, `intRandMax()`) as well as to query the number of random numbers generated (`numbersDrawn()`).

6.4.4 Random variates

The following functions are based on `dblrand()` and return random variables of different distributions:

Random variate functions use one of the random number generators (RNGs) provided by OMNeT++. By default this is generator 0, but you can specify which one to be used.

OMNeT++ has the following predefined distributions:

Function	**Description**
Continuous distributions	
`uniform(a, b, `*`rng=0`*`)`	uniform distribution in the range [a,b)
`exponential(mean, `*`rng=0`*`)`	exponential distribution with the given mean
`normal(mean, stddev, `*`rng=0`*`)`	normal distribution with the given mean and standard deviation
`truncnormal(mean, stddev, `*`rng=0`*`)`	normal distribution truncated to nonnegative values
`gamma_d(alpha, beta, `*`rng=0`*`)`	gamma distribution with parameters alpha>0, beta>0
`beta(alpha1, alpha2, `*`rng=0`*`)`	beta distribution with parameters alpha1>0, alpha2>0
`erlang_k(k, mean, `*`rng=0`*`)`	Erlang distribution with k>0 phases and the given mean
`chi_square(k, `*`rng=0`*`)`	chi-square distribution with k>0 degrees of freedom
`student_t(i, `*`rng=0`*`)`	student-t distribution with i>0 degrees of freedom
`cauchy(a, b, `*`rng=0`*`)`	Cauchy distribution with parameters a,b where b>0
`triang(a, b, c, `*`rng=0`*`)`	triangular distribution with parameters a<=b<=c, a!=c
`lognormal(m, s, rng=0)`	lognormal distribution with mean m and variance s>0
`weibull(a, b, `*`rng=0`*`)`	Weibull distribution with parameters a>0, b>0
`pareto_shifted(a, b, c, `*`rng=0`*`)`	generalized Pareto distribution with parameters a, b and shift c
Discrete distributions	
`intuniform(a, b, `*`rng=0`*`)`	uniform integer from a..b
`bernoulli(p, `*`rng=0`*`)`	result of a Bernoulli trial with probability 0<=p<=1 (1 with probability p and 0 with probability (1-p))

`binomial(n, p, rng=0)`	binomial distribution with parameters n>=0 and 0<=p<=1
`geometric(p, rng=0)`	geometric distribution with parameter 0<=p<=1
`negbinomial(n, p, rng=0)`	binomial distribution with parameters n>0 and 0<=p<=1
`poisson(lambda, rng=0)`	Poisson distribution with parameter lambda

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.

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.

6.4.5 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.

6.5 Container classes

6.5.1 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.

A queue object has a head and a tail. Normally, new elements are inserted at its head and elements are removed at its tail.

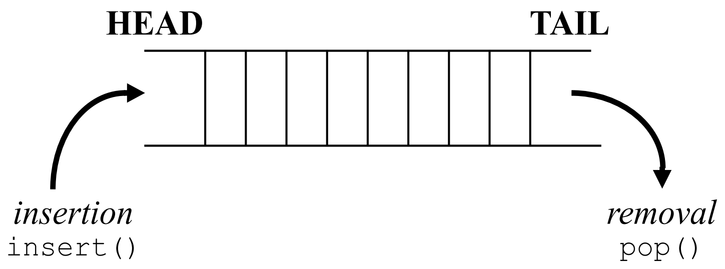


Figure 6.1: cQueue: insertion and removal

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, `cQueue::Iterator`, you can examine each object in the queue.

The `cQueue::Iterator` 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( cQueue::Iterator iter(queue,1); !iter.end(), iter++)
{
  cMessage *msg = (cMessage *) iter();
  //...
}
```

6.5.2 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 it grows automatically when it gets full. Internally, cArray is implemented with an array of pointers; when the array fills up, 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 is 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;
  }
}
```

6.6 The parameter class: cPar

Module parameters (as discussed in section 4.7) are represented as cPar objects. The module parameter name is the cPar object's name, and the object can store any parameter type supported by the NED language, that is, numeric (long or double), bool, string and XML config file reference. [2]

Module parameters are accessed via cModule's par() method:

```
cPar& par(const char *parameterName);
```

6.6.1 Reading the value

cPar has a number of methods for getting the parameter's value:

```
bool boolValue();
long longValue();
const char *stringValue();
double doubleValue();
cXMLElement *xmlValue();
```

There are also overloaded type cast operators for C/C++ primitive types including bool, int, long, double, const char *, and also for cXMLElement *. [3]

Thus, any of the following ways would work to store a parameter's value in a variable:

```
double foo = par("foo").doubleValue();
double foo = (double) par("foo");
double foo = par("foo");
```

If you use the par("foo") parameter in expressions (such as 4*par("foo")+2), the C++ compiler may be unable to decide between overloaded operators and report ambiguity. In that case you have to clarify by adding either an explicit cast ((double)par("foo") or (long)par("foo")) or use the doubleValue() or longValue() methods.

The isConstant() method can be used to determine whether a cPar stores a constant, or an expression that may produce a different value every time the object is read, such as 1+exponential(0.5).

[2] cPar objects used to be employed also for adding parameters (extra fields) to cMessage. While technically this is still feasible, message definitions (section 5.2) are a far superior solution in every respect.

[3] cPar also supports the void * and cObject * types, but these types were used primarily for message parameters before message definitions (section 5.2) got supported, and you cannot create such module parameters from NED.

6.6.2 Changing the value

There are many ways to set a `cPar`'s value. One is the `set...Value()` member functions:

```
cPar& foo = par("foo");
foo.setLongValue(12);
foo.setDoubleValue(2.7371);
foo.setStringValue("one two three");
```

There are also overloaded assignment operators for C++ primitive types, `const char *`, and `cXMLElement *`.

```
cPar pp("pp");
pp = 12;
pp = 2.7371;
pp = "one two three";
```

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).

`cPar` can also store other types which yield numeric results such as 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 offered to enter the value interactively.

Examples:

```
cPar foo("foo");
foo.setPrompt("Enter foo value:");
foo.setInput(true);   // make it an input parameter

double d = (double)foo; // the user will be prompted HERE
```

Further `set..()` functions to assign other storage types, e.g. double function with constant args (MathFuncNoArgs, MathFunc1Args, etc), reverse Polish expression, compiled expressions based on `cDouble-Expression`, random distribution based on a `cStatistic`'s `random()` method, pointer to `cObject`, etc. are listed in the next section; however, they are rarely useful for programming simulation models.

6.6.3 cPar storage types

`cPar` supports the basic data types (long, double, bool, string, XML) via several *storage types*. Storage types are internally identified by type characters. The type character is returned by the `type()` method.

Example:

```
cPar par = 10L;
char typechar = par.type(); // returns storage type 'L'
```

The all `cPar` data types and are summarized in the table below. The `isNumeric()` function tells whether the object stores a data types which allows the `doubleValue()` method to be called.

Type char	Storage type	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 a random variable of a certain distribution.

X	expr.	`setDoubleValue(`
		`  cPar::ExprElem*,int);`
		`double doubleValue();`
		`op double();`
			Runtime-evaluated Reverse Polish expression. Expression can contain constants, `cPar` objects, refer to other `cPars` (e.g. module parameters), can use math operators (+-*/^% etc), function calls (function must take 0,1,2 or 3 doubles and return a double). The expression must be given in an array of `cPar::ExprElem` structs.

C	compiled expr.	`setDoubleValue(`
		`  cDoubleEx-`
		`pression *expr);`
		`double doubleValue();`
		`op double();`
			Runtime-evaluated compiled expression. The expression should be supplied in a method of an object subclassed from `cDoubleExpression`.

T	distrib.	`setDoubleValue(`
		`  cStatistic*);`
		`double doubleValue();`
		`op double();`
			random variable generated from a distribution collected by a statistical data collection object (derived from `cStatistic`).

M	XML	`setXMLValue(`
		`  cXMLElement *node);`
		`cXMLElement *xmlValue();`
		op `cXMLElement*();`
			Reference to an XML element, found in an XML config file.

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 `configPointer()` 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.

6.7 Routing support: cTopology

6.7.1 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.

6.7.2 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 Host. (Router and Host can be either simple or compound module types.)

```
cTopology topo;
topo.extractByModuleType("Router", "Host", NULL);
```

Any number of module types can be supplied; the list must be terminated by NULL.

A dynamically assembled list of module types can be passed as a NULL-terminated array of const char* pointers, or in an STL string vector std::vector<std::string>. An example for the former:

```
cTopology topo;
const char *typeNames[3];
typeNames[0] = "Router";
typeNames[1] = "Host";
typeNames[2] = NULL;
topo.extractByModuleType(typeNames);
```

Second, you can extract all modules which have a certain parameter:

```
topo.extractByParameter( "ipAddress" );
```

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( "includeInTopo", &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 selectFunction(cModule *mod, void *)
{
  return mod->parentModule() == simulation.systemModule();
}

topo.extractFromNetwork( selectFunction, NULL );
```

A cTopology object uses two types: cTopology::Node for nodes and cTopology::Link for edges. (sTopoLinkIn and cTopology::LinkOut are 'aliases' for cTopology::Link; we'll talk 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++)
{
  cTopology::Node *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++)
  {
    cTopology::Node *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 `cTopology::Node` structure.

The correspondence between a graph node and a module can be obtained by:

```
cTopology::Node *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.

`cTopology::Node`'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 `cTopology::Link` is returned either as `cTopology::LinkIn` or as `cTopology::LinkOut` so that "remote" and "local" can be correctly interpreted for edges of both directions.)

6.7.3 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 =...;
cTopology::Node *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 `cTopology::Node` methods. Naturally, each call to `unweightedSingleShortestPathsTo()` overwrites the results of the previous call.

Walking along the path from our module to the target node:

```
cTopology::Node *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";
    cTopology::LinkOut *path = node->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 `cTopology::LinkOut`. 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:

```
cTopology::Node *thisnode = topo.nodeFor( this );
thisnode->disable();
topo.unweightedSingleShortestPathsTo( targetnode );
thisnode->enable();

for (int j=0; j<thisnode->outLinks(); j++)
{
  cTopology::LinkOut *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(cTopology::Node *target);
weightedSingleShortestPathsTo(cTopology::Node *target);
weightedMultiShortestPathsTo(cTopology::Node *target);
```

6.8 Statistics and distribution estimation

6.8.1 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 [JC85]. 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.

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 numSamples = stat.samples();
double smallest = stat.min(),
       largest = stat.max();
double mean = stat.mean(),
       standardDeviation = stat.stddev(),
       variance = stat.variance();
```

6.8.2 Distribution estimation

Initialization and usage

The distribution estimation classes (cLongHistogram, cDoubleHistogram, cVarHistogram, 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 for setting up the range and to specify how many observations should be used for automatically determining the range.

```
setRange(lower,upper);
setRangeAuto(numFirstvals, rangeExtFactor);
setRangeAutoLower(upper, numFirstvals, rangeExtFactor);
setRangeAutoUpper(lower, numFirstvals, rangeExtFactor);


setNumFirstVals(numFirstvals);
```

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.

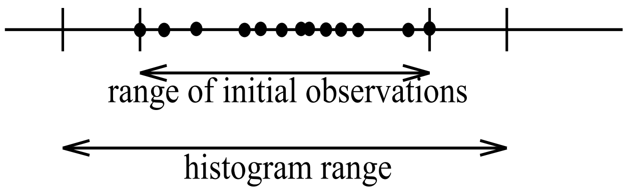


Figure 6.2: Setting up a histogram's range

After the cells have been set up, collection 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.

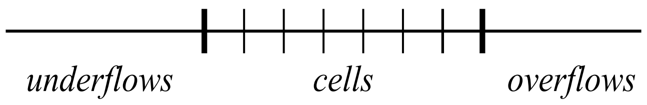


Figure 6.3: Histogram structure after setting up the cells

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`.

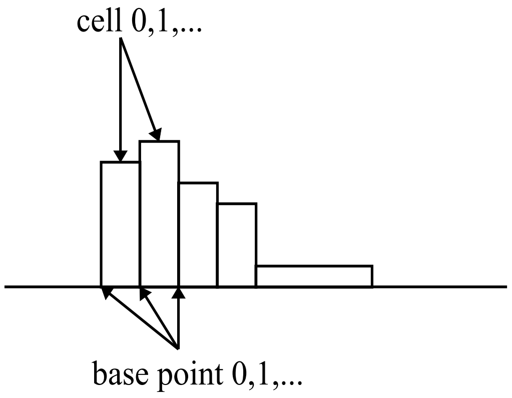


Figure 6.4: base points and cells

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 rndPar("rndPar");
rndPar.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)rndPar; // 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);

cDoubleHistogram hist2("Hist-from-file");
FILE *f2 = fopen("histogram.dat","r");
hist2.loadFromFile(f2); // load stored distribution
fclose(f2);
```

Histogram with custom cells

The `cVarHistogram` class can be used to create histograms with arbitrary (non-equidistant) cells. It can operate in two modes:

- *manual*, where you specify cell boundaries explicitly before starting collecting
- *automatic*, where `transform()` will set up the cells after collecting a certain number of initial observations. The cells will be set up so that as far as possible, an equal number of observations fall into each cell (equi-probable cells).

Modes are selected with a *transform-type* parameter:

- `HIST_TR_NO_TRANSFORM`: no transformation; uses bin boundaries previously defined by `addBin-Bound()`
- `HIST_TR_AUTO_EPC_DBL`: automatically creates equiprobable cells
- `HIST_TR_AUTO_EPC_INT`: like the above, but for integers

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 `numFirstVals` initial observations. One cannot add cell boundaries when the histogram has already been transformed.

6.8.3 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. Experience shows that $k = 2$ worked best.

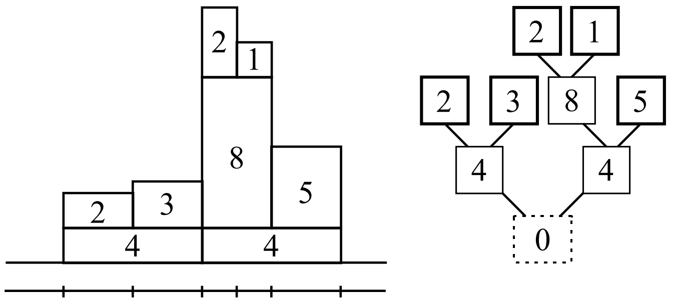


Figure 6.5: Illustration of the k-split algorithm, $k = 2$. The numbers in boxes represent the observation count values

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:

$m_{...,i,j} = (1 - \lambda)\tilde{n}_{...,i}/k + \lambda \tilde{n}_{...,i} s_{...,i,j}/s_{...,i}$ where $\lambda \in [0, 1]$

Note that while $n_{...,i}$ are integers, $m_{...,i}$ and thus $\tilde{n}_{...,i}$ 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.

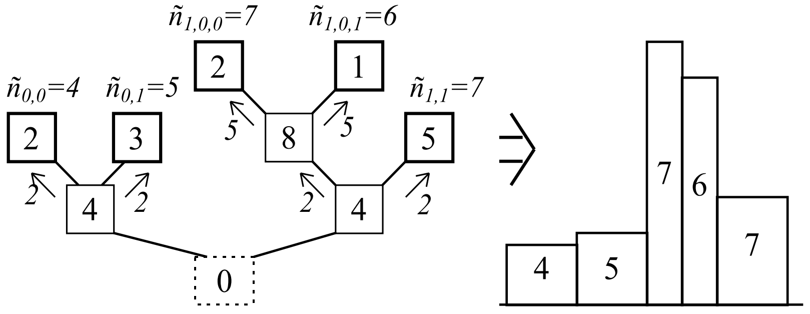


Figure 6.6: Density estimation from the k-split cell tree. We assume $\lambda = 0$, i.e. we distribute mother observations evenly.

Strictly speaking, the *k*-split algorithm is semi-online, because its needs some observations to set up the initial histogram range. 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 sufficient 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

The `cKSplit` class is an implementation of the ***k-split*** method. Member functions:

```
void setCritFunc(KSplitCritFunc _critfunc, double *_critdata);
void setDivFunc(KSplitDivFunc \_divfunc, double *\_divdata);
void rangeExtension( bool enabled );

int treeDepth();
int treeDepth(sGrid& grid);

double realCellValue(sGrid& grid, int cell);
void printGrids();

sGrid& grid(int k);
sGrid& rootGrid();

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
};
```

6.8.4 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 has 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

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 accuracy 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);
```

6.9 Recording simulation results

6.9.1 Output vectors: cOutVector

Objects of type `cOutVector` are responsible for writing time series data (referred to as *output vectors*) to a file. The `record()` method is used to output a value (or a value pair) with a timestamp. The object name will serve as the name of the output vector.

The vector name can be passed in the constructor,

```
cOutVector responseTimeVec("response time");
```

but in the usual arrangement you'd make the `cOutVector` a member of the module class and set the name in `initialize()`. You'd record values from `handleMessage()` or from a function called from `handleMessage()`.

The following example is a `Sink` module which records the lifetime of every message that arrives to it.

```
class Sink : public cSimpleModule
{
  protected:
    cOutVector endToEndDelayVec;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(Sink);

void Sink::initialize()
{
    endToEndDelayVec.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->creationTime();
    endToEndDelayVec.record(eed);
    delete msg;
}
```

There is also a `recordWithTimestamp()` method$^{New!}$, to make it possible to record values into output vectors with a timestamp other than `simTime()`. Increasing timestamp order is still enforced though.

All `cOutVector` objects write to a single *output vector file* named `omnetpp.vec` by default. You can configure output vectors from `omnetpp.ini`: you can disable writing to the file, or limit it to a certain simulation time interval for recording (section 8.5).

The format and processing of output vector files is described in section 10.1.

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.

6.9.2 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 output 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()` method of `cSimpleModule`, and you'll usually want to insert this code into the `finish()` function. An example:

```
void Transmitter::finish()
{
    double avgThroughput = totalBits / simTime();
    recordScalar("Average throughput", avgThroughput);
}
```

You can record whole statistics objects by calling their `recordScalar()` methods, declared as part of `cStatistic`. In the following example we create a `Sink` module which calculates the mean, standard deviation, minimum and maximum values of a variable, and records them at the end of the simulation.

```
class Sink : public cSimpleModule
{
  protected:
    cStdDev eedStats;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};

Define_Module(Sink);

void Sink::initialize()
{
    eedStats.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->creationTime();
    eedStats.collect(eed);
    delete msg;
}

void Sink::finish()
{
    recordScalar("Simulation duration", simTime());
    eedStats.recordScalar();
}
```

The above calls write into the *output scalar file* which is named `omnetpp.sca` by default. The output scalar file is preserved across simulation runs (unlike the output vector file which gets deleted at the beginning of every simulation run). Data are always appended at the end of the file, and output from different simulation runs are separated by special lines. The format and processing of output vector files is described in section 10.2.

6.9.3 Precision$^{New!}$

Output scalar and output vector files are text files, and floating point values (`double`s) are recorded into it using `fprintf()`'s `"%g"` format. The number of significant digits can be configured using the `output-scalar-precision=` and `output-vector-precision=` configuration entries (see 8.2.6). The default precision is 12 digits. The following has to be considered when changing the default value:

IEEE-754 doubles are 64-bit numbers. The mantissa is 52 bits, which is roughly equivalent to 16 decimal places (52*log(2)/log(10)). However, due to rounding errors, usually only 12..14 digits are correct, and the rest is pretty much random garbage which should be ignored. However, when you convert the decimal representation back into an IEEE-754 double (as in Plove and Scalars), an additional small error will occurs because 0.1, 0.01, etc cannot be accurately represented in binary. This conversion error is usually smaller than the one that the `double` variable already had before recording into the file, however if it is important you can eliminate it by setting >16 digits precision for the file (but again, be aware that the last digits are garbage). The practical upper limit is 17 digits, setting it higher doesn't make any difference in `fprintf()`'s output.

Errors coming from converting to/from decimal representation can be eliminated by choosing an output vector/output scalar manager class which stores `double`s in their native binary form. The appropriate configuration entries are `outputvectormanager-class=` and `outputvectormanager-class=`; see 8.2.6. For example, `cMySQLOutputScalarManager` and `cMySQLOutputScalarManager` provided in `samples/database` fulfill this requirement.

However, before worrying too much about rounding and conversion errors, it is worth considering what is the *real* accuracy of your results. Some things to consider:

- in real life, it is very hard to measure quantities (weight, distance, even time) with more than a few digits of precision. What precision are your input data? For example, if you approximate inter-arrival time as *exponential(0.153)* when the mean is really *0.152601...* and the distribution is not even exactly exponential, you are already starting out with a bigger error than rounding can cause.
- the simulation model is itself an approximation of real life. How much error do the (known and unknown) simplifications cause in the results?

6.10 Watches and snapshots

6.10.1 Basic watches

It would be nice, but variables of type `int`, `long`, `double` do not show up by default in Tkenv; neither do STL classes (`std::string`, `std::vector`, etc.) or your own structs and classes. This is because the simulation kernel, being a library, knows nothing about types and variables in your source code.

OMNeT++ provides `WATCH()` and set of other macros to come to your rescue, and make variable to be inspectable in Tkenv and to be output into the snapshot file. `WATCH()` macros are usually placed into `initialize()` (to watch instance variables) or to the top of the `activity()` function (to watch its local variables), the point being that they should only be executed once.

```
long packetsSent;
double idleTime;
```

```
WATCH(packetsSent);
WATCH(idleTime);
```

Of course, members of classes and structs can also be watched:

```
WATCH(config.maxRetries);
```

When you open an inspector for the simple module in Tkenv and click the Objects/Watches tab in it, you'll see your watched variables and their values there. Tkenv also lets you change the value of a watched variable.

The `WATCH()` macro can be used with any type that has a stream output operator (`operator«`) defined$^{New!}$. By default, this includes all primitive types and `std::string`, but since you can write `operator«` for your classes/structs and basically any type, `WATCH()` can be used with anything. The only limitation is that since the output should more or less fit on single line, the amount of information that can be conveniently displayed is limited.

An example stream output operator:

```
std::ostream& operator<<(std::ostream& os, const ClientInfo& cli)
{
    os << "addr=" << cli.clientAddr << "  port=" << cli.clientPort; // no endl!
    return os;
}
```

And the `WATCH()` line:

```
WATCH(currentClientInfo);
```

6.10.2 Read-write watches$^{New!}$

Watches for primitive types and `std::string` allow for changing the value from the GUI as well, but for other types you need to explicitly add support for that. What you need to do is define a stream input operator (`operator»`) and use the `WATCH_RW()` macro instead of `WATCH()`.

The stream input operator:

```
std::ostream& operator>>(std::istream& is, ClientInfo& cli)
{
    // read a line from "is" and parse its contents into "cli"
    return is;
}
```

And the `WATCH_RW()` line:

```
WATCH_RW(currentClientInfo);
```

6.10.3 Structured watches$^{New!}$

`WATCH()` and `WATCH_RW()` are basic watches: they allow one line of (unstructured) text to be displayed. However, if you have a data structure generated from message definitions (see Chapter 5), then one can do better. The message compiler automatically generates meta-information describing individual fields of the class or struct, which makes it possible to display the contents on field level.

The WATCH macros to be used for this purpose are WATCH_OBJ() and WATCH_PTR(). Both expect the object to be subclassed from cPolymorphic; WATCH_OBJ() expects a reference to such class, and WATCH_PTR() expects a pointer variable.

```
ExtensionHeader hdr;
ExtensionHeader *hdrPtr;
...
WATCH_OBJ(hdr);
WATCH_PTR(hdrPtr);
```

CAUTION: With WATCH_PTR(), the pointer variable must point to a valid object or be NULL at all times, otherwise the GUI may crash while trying to display the object. This practically means that the pointer should be initialized to NULL even if not used, and should be set to NULL when the object to which it points gets deleted.

```
delete watchedPtr;
watchedPtr = NULL;  // set to NULL when object gets deleted
```

6.10.4 STL watches$^{New!}$

The standard C++ container classes (vector, map, set, etc) also have structured watches, available via the following macros:

WATCH_VECTOR(), WATCH_PTRVECTOR(), WATCH_LIST(), WATCH_PTRLIST(), WATCH_SET(), WATCH_PTRSET(), WATCH_MAP(), WATCH_PTRMAP().

The PTR-less versions expect the data items ("T") to have stream output operators (operator «), because that's how they will display them. The PTR versions assume that data items are pointers to some type which has operator «. WATCH_PTRMAP() assumes that only the value type ("second") is a pointer, the key type ("first") is not. (If you happen to use pointers as key, then define operator « for the pointer type itself.)

Examples:

```
std::vector<int> intvec;
WATCH_VECTOR(intvec);

std::map<std::string,Command*> commandMap;
WATCH_PTRMAP(commandMap);
```

6.10.5 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(&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:

```
[...]

(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

(TokenRing) `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

[...]

(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

[...]
```

It is possible that the format of the snapshot file will change to XML in future OMNeT++ releases.

6.10.6 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.

6.10.7 Getting coroutine stack usage

It 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. [4]

`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.

6.11 Deriving new classes

6.11.1 cObject or not?

If you plan to implement a completely new class (as opposed to subclassing something already present in OMNeT++), you have to ask yourself whether you want the new class to be based on `cObject` or not. Note that we are *not* saying you should always subclass from `cObject`. Both solutions have advantages and disadvantages, which you have to consider individually for each class.

`cObject` already carries (or provides a framework for) significant functionality that is either relevant to your particular purpose or not. Subclassing `cObject` generally means you have more code to write (as you *have to* redefine certain virtual functions and adhere to conventions) and your class will be a bit more heavy-weight. However, if you need to store your objects in OMNeT++ objects like `cQueue`, or you'll want to store OMNeT++ classes in your object, then you *must* subclass from `cObject`. [5]

The most significant features `cObject` has is the name string (which has to be stored somewhere, so it has its overhead) and ownership management (see section 6.12) which also has the advantages but also some costs.

As a general rule, small `struct`-like classes like `IPAddress`, `MACAddress`, `RoutingTableEntry`, `TCP-ConnectionDescriptor`, etc. are better *not* sublassed from `cObject`. If your class has at least one virtual member function, consider subclassing from `cPolymorphic`, which does not impose any extra cost because it doesn't have data members at all, only virtual functions.

6.11.2 cObject virtual methods

Most 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 is presented here. You do not need to worry about the length of the list: most functions are not absolutely necessary to implement. For example, you do not need to redefine `forEachChild()` unless your class is a container class.

The following methods **must** be implemented:

- *Constructor*. At least two constructors should be provided: one that takes the object name string

[4]The actual value is platform-dependent.

[5]For simplicity, in the these sections "OMNeT++ object" should be understood as "object of a class subclassed from `cObject`"

as `const char *` (recommended by convention), and another one with no arguments (must be present). The two are usually implemented as a single method, with `NULL` as default name string.

- *Copy constructor*, which must have the following signature for a class `X`: `X(const X&)`. The copy constructor is used whenever an object is duplicated. The usual implementation of the copy constructor is to initialize the base class with the name (`name()`) of the other object it receives, then call the assignment operator (see below).
- *Destructor*.
- *Duplication function,* `cPolymorphic *dup() const`. It should create and return an exact duplicate of the object. It is usually a one-line function, implemented with the help of the `new` operator and the copy constructor.
- *Assigment operator*, that is, `X& operator=(const X&)` for a class `X`. It should copy the contents of the other object into this one, except the name string. See later what to do if the object contains pointers to other objects.

If your class contains other objects subclassed from `cObject`, either via pointers or as data member, the following function **should** be implemented:

- *Iteration function,* `void forEachChild(cVisitor * v)`. The implementation should call the function passed for each object it contains via pointer or as data member; see the API Reference on `cObject` on how to implement `forEachChild()`. `forEachChild()` makes it possible for Tkenv to display the object tree to you, to perform searches on it, etc. It is also used by `snapshot()` and some other library functions.

The following methods are **recommended** to implement:

- *Object info,* `std::string info()`. The `info()` function should return a one-line string describing the object's contents or state. `info()` is displayed at several places in Tkenv.
- *Detailed object info,* `std::string detailedInfo()`. This method may potentially be implemented in addition to `info()`; it can return a multi-line description. `detailedInfo()` is also displayed by Tkenv in the object's inspector.
- *Serialization*, `netPack()` and `netUnpack()` methods. These methods are needed for parallel simulation, if you want objects of this type to be transmitted across partitions.

6.11.3 Class registration

You should also use the `Register_Class()` macro to register the new class. It is used by the `createOne()` factory function, which can create any object given the class name as a string. `createOne()` is used by the Envir library to implement `omnetpp.ini` options such as `rng-class="..."` or `scheduler-class="..."`. (see Chapter 13)

For example, an omnetpp.ini entry such as

```
rng-class="cMersenneTwister"
```

would result in something like the following code to be executed for creating the RNG objects:

```
cRNG *rng = check_and_cast<cRNG*>(createOne("cMersenneTwister"));
```

But for that to work, we needed to have the following line somewhere in the code:

```
Register_Class(cMersenneTwister);
```

`createOne()` is also needed by the parallel distributed simulation feature (Chapter 12) to create blank objects to unmarshal into on the receiving side.

6.11.4 Details

We'll go through the details using an example. We create a new class `NewClass`, redefine all above mentioned `cObject` member functions, and explain the conventions, rules and tips associated with them. To demonstrate as much as possible, the class will contain an `int` data member, dynamically allocated non-`cObject` data (an array of `double`s), an OMNeT++ object as data member (a `cQueue`), and a dynamically allocated OMNeT++ object (a `cMessage`).

The class declaration is the following. It contains the declarations of all methods discussed in the previous section.

```
//
// file: NewClass.h
//
#include <omnetpp.h>

class NewClass : public cObject
{
  protected:
    int data;
    double *array;
    cQueue queue;
    cMessage *msg;
    ...
  public:
    NewClass(const char *name=NULL, int d=0);
    NewClass(const NewClass& other);
    virtual ~NewClass();
    virtual cPolymorphic *dup() const;
    NewClass& operator=(const NewClass& other);

    virtual void forEachChild(cVisitor *v);
    virtual std::string info();
};
```

We'll discuss the implementation method by method. Here's the top of the `.cc` file:

```
//
// file: NewClass.cc
//
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include "newclass.h"

Register_Class( NewClass );


NewClass::NewClass(const char *name, int d) : cObject(name)
```

```
{
    data = d;
    array = new double[10];
    take(&queue);
    msg = NULL;
}
```

The constructor (above) calls the base class constructor with the name of the object, then initializes its own data members. You need to call `take()` for `cObject`-based data members.

```
NewClass::NewClass(const NewClass& other) : cObject(other.name())
{
    array = new double[10];
    msg = NULL;
    take(&queue);
    operator=(other);
}
```

The copy constructor relies on the assignment operator. Because by convention the assignment operator does not copy the name member, it is passed here to the base class constructor. (Alternatively, we could have written `setName(other.name())` into the function body.)

Note that pointer members have to be initialized (to `NULL` or to an allocated object/memory) before calling the assignment operator, to avoid crashes.

You need to call `take()` for `cObject`-based data members.

```
NewClass::~NewClass()
{
    delete [] array;
    if (msg->owner()==this)
        delete msg;
}
```

The destructor should delete all data structures the object allocated. `cObject`-based objects should *only* be deleted if they are owned by the object – details will be covered in section 6.12.

```
cPolymorphic *NewClass::dup() const
{
    return new NewClass(*this);
}
```

The `dup()` functions is usually just one line, like the one above.

```
NewClass& NewClass::operator=(const NewClass& other)
{
    if (&other==this)
        return *this;
    cObject::operator=(other);

    data = other.data;

    for (int i=0; i<10; i++)
        array[i] = other.array[i];
    queue = other.queue;
    queue.setName(other.queue.name());

    if (msg && msg->owner()==this)
        delete msg;
    if (other.msg && other.msg->owner()==const_cast<cMessage*>(&other))
        take(msg = (cMessage *)other.msg->dup());
    else
        msg = other.msg;
    return *this;
}
```

Complexity associated with copying and duplicating the object is concentrated in the assignment operator, so it is usually the one that requires the most work from you of all methods required by `cObject`.

If you do not want to implement object copying and duplication, you should implement the assigment operator to call `copyNotSupported()` – it'll throw an exception that stops the simulation with an error message if this function is called.

The assignment operator copies contents of the `other` object to this one, except the name string. It should always return `*this`.

First, we should make sure we're not trying to copy the object to itself, because it might be disastrous. If so (that is, `&other==this`), we return immediately without doing anything.

The base class part is copied via invoking the assignment operator of the base class.

New data members are copied in the normal C++ way. If the class contains pointers, you'll most probably want to make a deep copy of the data where they point, and not just copy the pointer values.

If the class contains pointers to OMNeT++ objects, you need to take ownership into account. If the contained object is *not owned* then we assume it is a pointer to an "external" object, consequently we only copy the pointer. If it is *owned*, we duplicate it and become the owner of the new object. Details of ownership management will be covered in section 6.12.

```
void NewClass::forEachChild(cVisitor *v)
{
    v->visit(queue);
    if (msg)
        v->visit(msg);
}
```

The `forEachChild()` function should call `v->visit(obj)` for each `obj` member of the class. See the API Reference for more information of `forEachChild()`.

```
std::string NewClass::info()
{
    std::stringstream out;
    out << "data=" << data << ", array[0]=" << array[0];
    return out.str();

}
```

The `info()` method should produce a concise, one-line string about the object. You should try not to exceed 40-80 characters, since the string will be shown in tooltips and listboxes.

See the virtual functions of `cPolymorphic` and `cObject` in the class library reference for more information. The sources of the Sim library (`include/`, `src/sim/`) can serve as further examples.

6.12 Object ownership management

6.12.1 The ownership tree

OMNeT++ has a built-in ownership management mechanism which is used for sanity checks, and as part of the infrastructure supporting Tkenv inspectors.

Container classes like `cQueue` own the objects inserted into them. But this is not limited to objects inserted into a container: *every `cObject`-based object has an owner all the time*. From the user's point of view, ownership is managed transparently. For example, when you create a new `cMessage`, it will be owned by the simple module. When you send it, it will first be handed over to (i.e. change owership to) the FES, and, upon arrival, to the destination simple module. When you encapsulate the message in another one, the encapsulating message will become the owner. When you decapsulate it again, the currently active simple module becomes the owner.

The `owner()` method, defined in `cObject`, returns the owner of the object:

```
cObject *o = msg->owner();
ev << "Owner of " << msg->name() << " is: " <<
   << "(" << o->className() << ") " << o->fullPath() << endl;
```

The other direction, enumerating the objects owned can be implemented with the `forEachChild()` method by it looping through all contained objects and checking the owner of each object.

Why do we need this?

The traditional concept of object ownership is associated with the "right to delete" objects. In addition to that, keeping track of the owner and the list of objects owned also serves other purposes in OMNeT++:

- enables methods like `fullPath()` to be implemented.
- prevents certain types of programming errors, namely, those associated with wrong ownership handling.
- enables Tkenv to display the list of simulation objects present within a simple module. This is extremely useful for finding memory leaks caused by forgetting to delete messages that are no longer needed.

Some examples of programming errors that can be caught by the ownership facility:

- attempts to send a message while it is still in a queue, encapsulated in another message, etc.
- attempts to send/schedule a message while it is still owned by the simulation kernel (i.e. scheduled as a future event)
- attempts to send the very same message object to multiple destinations at the same time (ie. to all connected modules)

For example, the `send()` and `scheduleAt()` functions check that the message being sent/scheduled *must* is owned by the module. If it is not, then it signals a programming error: the message is probably owned by another module (already sent earlier?), or currently scheduled, or inside a queue, a message or some other object – in either case, the module does not have any authority over it. When you get the error message (`"not owner of object"`), you need to carefully examine the error message: which object has the ownership of the message, why's that, and then probably you'll need to fix the logic somewhere in your program.

The above errors are easy to make in the code, and if not detected automatically, they could cause random crashes which are usually very difficult to track down. Of course, some errors of the same kind still cannot be detected automatically, like calling member functions of a message object which has been sent to (and so currently kept by) another module.

6.12.2 Managing ownership

Ownership is managed transparently for the user, but this mechanism has to be supported by the participating classes themselves. It will be useful to look inside `cQueue` and `cArray`, because they might give you a hint what behavior you need to implement when you want to use non-OMNeT++ container classes to store messages or other `cObject`-based objects.

Insertion

`cArray` and `cQueue` have internal data structures (array and linked list) to store the objects which are inserted into them. However, they do *not* necessarily own all of these objects. (Whether they own an object or not can be determined from that object's `owner()` pointer.)

The default behaviour of `cQueue` and `cArray` is to take ownership of the objects inserted. This behavior can be changed via the *takeOwnership* flag.

Here's what the *insert* operation of `cQueue` (or `cArray`) does:

- insert the object into the internal array/list data structure
- if the *takeOwnership* flag is true, take ownership of the object, otherwise just leave it with its original owner

The corresponding source code:

```
void cQueue::insert(cObject *obj)
{
    // insert into queue data structure
    ...

    // take ownership if needed
    if (takeOwnership())
        take(obj);

}
```

Removal

Here's what the *remove* family of operations in `cQueue` (or `cArray`) does:

- remove the object from the internal array/list data structure
- if the object is actually owned by this `cQueue/cArray`, release ownership of the object, otherwise just leave it with its current owner

After the object was removed from a `cQueue/cArray`, you may further use it, or if it is not needed any more, you can delete it.

The *release ownership* phrase requires further explanation. When you remove an object from a queue or array, the ownership is expected to be transferred to the simple module's local objects list. This is acomplished by the `drop()` function, which transfers the ownership to the object's default owner. `defaultOwner()` is a virtual method returning `cObject*` defined in `cObject`, and its implementation returns the currently executing simple module's local object list.

As an example, the `remove()` method of `cQueue` is implemented like this: [6]

```
cObject *cQueue::remove(cObject *obj)
{
    // remove object from queue data structure
    ...

    // release ownership if needed
    if (obj->owner()==this)
        drop(obj);

    return obj;
}
```

Destructor

The concept of `ownership` is that *the owner has the exclusive right and duty to delete the objects it owns*. For example, if you delete a `cQueue` containing `cMessage`s, all messages it contains *and* owns will also be deleted.

The destructor should delete all data structures the object allocated. From the contained objects, only the owned ones are deleted – that is, where `obj->owner()==this`.

Object copying

The ownership mechanism also has to be taken into consideration when a `cArray` or `cQueue` object is duplicated. The duplicate is supposed to have the same content as the original, however the question is whether the contained objects should also be duplicated or only their pointers taken over to the duplicate `cArray` or `cQueue`.

The convention followed by `cArray`/`cQueue` is that only owned objects are copied, and the contained but not owned ones will have their pointers taken over and their original owners left unchanged.

In fact, the same question arises in three places: the assignment operator `operator=()`, the copy constructor and the `dup()` method. In OMNeT++, the convention is that copying is implemented in the assignment operator, and the other two just rely on it. (The copy constructor just constructs an empty object and invokes assigment, while `dup()` is implemented as `new cArray(*this)`).

[6]Actual code in `src/sim` is structured somewhat differently, but the meaning is the same.

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.
- Message definitions, in files with `.msg` suffix.
- Simple modules implementations and other C++ code, in `.cc` files (or `.cpp`, on Windows)

To build an executable simulation program, you first need to translate the NED files and the message files into C++, using the NED compiler (`nedtool`) and the message compiler (`opp_msgc`). After this step, the process is the same as building any C/C++ program from source: all C++ sources need to be compiled into object files (`.o` files on Unix/Linux, and `.obj` on Windows), and all object files need to be linked with the necessary libraries to get an executable.

File names for libraries differ for Unix/Linux and for Windows, and also different for static and shared libraries. Let us suppose you have a library called Tkenv. On a Unix/Linux system, the file name for the static library would be something like `libtkenv.a` (or `libtkenv.a.`<*version*>), and the shared library would be called `libtkenv.so` (or `libtkenv.so.`<*version*>). The Windows version of the static library would be `tkenv.lib`, and the DLL (which is the Windows equivalent of shared libraries) would be a file named `tkenv.dll`.

You'll need to link with the following libraries:

- The simulation kernel and class library, called *sim_std* (file `libsim_std.a`, `sim_std.lib`, etc).
- User interfaces. The common part of all user interfaces is the *envir* library (file `libenvir.a`, etc), and the specific user interfaces are *tkenv* and *cmdenv* (`libtkenv.a`, `libcmdenv.a`, etc). You have to link with *envir*, plus either *tkenv* or *cmdenv*.

Luckily, you do not have to worry about the above details, because automatic tools like `opp_makemake` will take care of the hard part for you.

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 (also Windows with Cygwin or MinGW)
- MSVC 6.0 on Windows

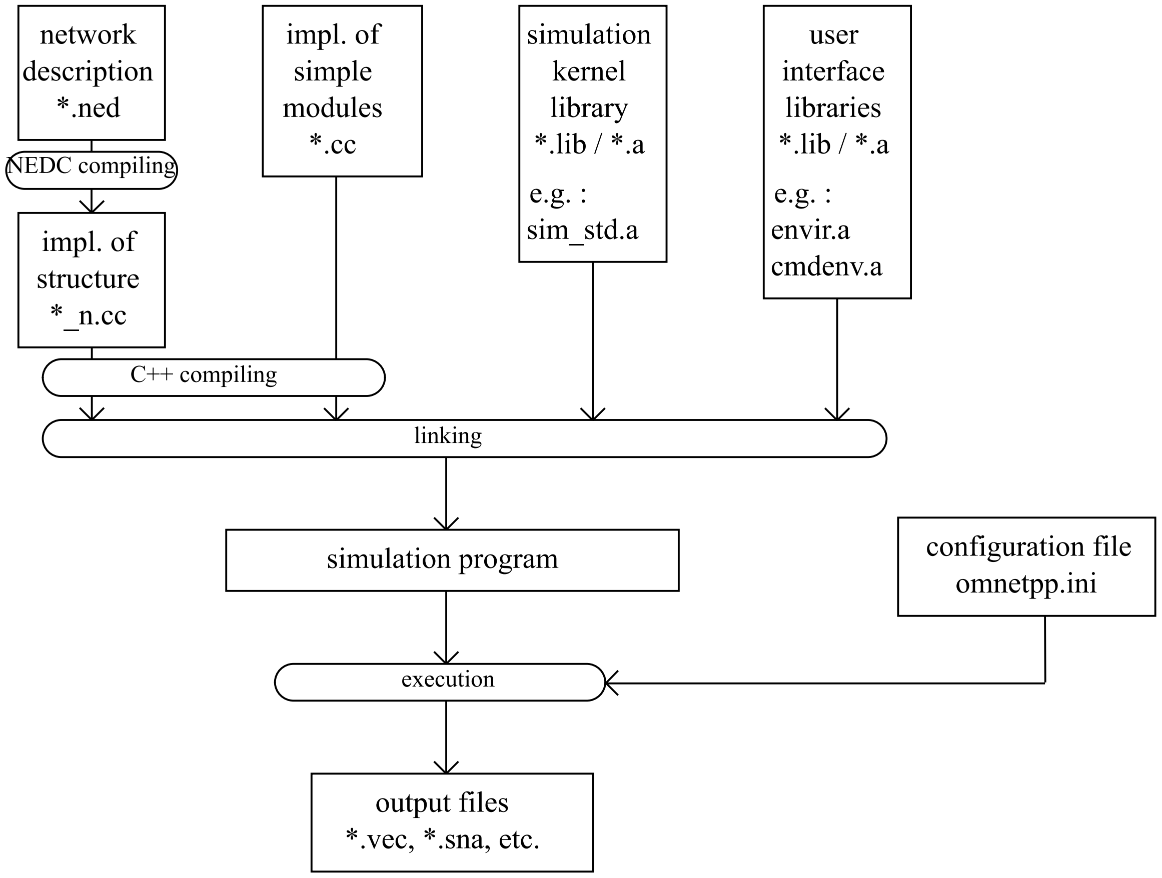

network description *.ned
NEDC compiling
impl. of structure *_n.cc
impl. of simple modules *.cc
simulation kernel library *.lib / *.a
e.g. : sim_std.a
user interface libraries *.lib / *.a
e.g. : envir.a cmdenv.a
C++ compiling
linking
simulation program
configuration file omnetpp.ini
execution
output files *.vec, *.sna, etc.

Figure 7.1: Building and running simulation

7.2 Using Unix and gcc

This section applies to using OMNeT++ on Linux, Solaris, FreeBSD and other Unix derivatives, and also more or less to Cygwin and MinGW on Windows.

Here in the manual we can give you a rough overview only. The `doc/` directory of your OMNeT++ installation contains `Readme.`*<platform>* files that provide up-to-date, more detailed and more precise instructions.

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, so it is better to refer to the readme files. If you compile from source, you can expect the usual GNU procedure: `./configure` followed by `make`.

7.2.2 Building simulation models

The `opp_makemake` script can automatically generate the `Makefile` for your simulation program, based on the source files in the current directory. (It can also handle large models which are spread across several directories; this is covered later in this section.)

`opp_makemake` has several options, with the `-h` option it displays a summary.

```
% opp_makemake -h
```

Once you have the source files (`*.ned`, `*.msg`, `*.cc`, `*.h`) in a directory, `cd` there 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`. The warnings during the dependency generation process can be safely ignored.

In addition to the simulation executable, the `Makefile` contains other targets, too. Here is a list of important ones:

Target	**Action**
	The default target is to build the simulation executable
depend	Adds (or refreshes) dependencies in the `Makefile`
clean	Deletes all files that were produced by the make process
makefiles	Regenerates the `Makefile` using `opp_makemake` (this is useful if e.g. after upgrading OMNeT++, if `opp_makemake` has changed)
makefile-ins	Similar to `make makefiles`, but it regenerates the `Makefile.in` instead

If you already had a `Makefile` in that directory, `opp_makemake` will refuse to overwrite 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 for the changes to take effect.

You can specify the user interface (Cmdenv/Tkenv) with the -u option (with no -u, Tkenv is the default):

```
% opp_makemake -u Tkenv
```

Or:

```
% opp_makemake -u Cmdenv
```

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 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 (shared) libraries. Here we discuss static linking.

In every subdirectory which contains source files, (say `app/` and `routing/`), run

```
opp_makemake -n
```

The -n option means no linking is necessary, only compiling has to be done.

In non-leaf directories, run

```
opp_makemake -r -n
```

The -r option enables recursive make: when you build the simulation, make will descend into the subdirectories and runs make in them too. By default, -r decends into all subdirectories; the -X directory option can be used to make it ignore certain subdirectories.

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../routing
```

in the `app/` directory, and vice versa.

To build an executable, the -w option can be used; it causes a simulation executable to be built using all object files from the include (-I) directories:

```
opp_makemake -w -I../routing -I../app
```

You can affect build order by adding dependencies among subdirectories into the `makefrag` (`makefrag.vc`) file.

For a complex example of using opp_makemake, check the Makefiles of the INET Framework, or rather, the makemake script (and makemake.bat file) which contain the commands to generate the makefiles.

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 Windows and Microsoft Visual C++

This is only a rough overview. Up-to-date, more detailed and more precise instructions can be found in the `doc/` directory of your OMNeT++ installation, in the file `Readme.MSVC`.

7.3.1 Installation

It is easiest to start with the binary, installer version. It contains all necessary software except MSVC, and you can get a working system up and running very fast.

Later you'll probably want to download and build the source distribution too. Reasons for that might be to compile the libraries with different flags, to debug into them, or to recompile with support for additional packages (e.g. Akaroa, MPI). Compilation should be painless (it takes a single `nmake -f Makefile.vc` command) after you get the different component directories right in `configuser.vc`. Additional software needed for the compilation is also described in `doc/`.

7.3.2 Building simulation models on the command line

OMNeT++ has an automatic MSVC makefile creator named `opp_nmakemake` which is probably the easier way to go. Its usage is very similar to the similarly named tool for Unix.

If you run `opp_nmakemake` in a directory of model sources, it collects all the names of all source files in the directory, and creates a makefile from them. The resulting makefile is called `Makefile.vc`.

To use `opp_nmakemake`, open a command window (*Start menu* -> *Run...* –> type `cmd`), then `cd` to the directory of your model and type:

```
opp_nmakemake
```

`opp_nmakemake` has several command-line options, mostly the same as the Unix version.

Then you can build the program by typing:

```
nmake -f Makefile.vc
```

The most common problem is that `nmake` (which is is part of MSVC) cannot be found because it is not in the path. You can fix this by running `vcvars32.bat`, which can be found in the MSVC `bin` directory (usually `C:\Program Files\Microsoft Visual Studio\VC98\Bin`).

7.3.3 Building simulation models from the MSVC IDE

You can also use the MSVC IDE for development. It is best to start by copying one of the sample simulations.

If you want to use compiled NED files (as opposed to dynamic NED loading, described in section 8.3), you need to add NED files to the project, with Custom Build Step commands to invoke the NED compiler (`nedtool`) on them. You also need to add the `_n.cc` files generated by `nedtool` to the project. There is an `AddNEDFileToProject` macro which performs exactly this task: adding a NED file and the corresponding `_n.cc` file, and configuring the Custom Build Step.

Some caveats (please read `doc/Readme.MSVC` for more!):

- **how to get the graphical environment**. By default, the sample simulations link with Cmdenv if you rebuild them from the IDE. To change to Tkenv, choose Build | Set active configuration from the menu, select "Debug-Tkenv" or "Release-Tkenv", then re-link the executable.

- **can't find a usable init.tcl**. If you get this message, Tcl/Tk is missing the `TCL_LIBRARY` environment variable which is normally set by the installer. If you see this message, you need to set this variable yourself to the Tcl `lib/` directory.
- **changed compiler settings**. Changes since OMNeT++ 2.2: You'll need exception handling and RTTI turned ON, and stack size set to as low as 64K. See the readme file for rationale and more hints.
- **adding NED files**. After you added a `.ned` file to the project, you also have to add a `_n.cpp` file, and set a *Custom Build Step* for them:

  ```
  Description: NED Compiling $(InputPath)
  Command: nedtool -s _n.cpp $(InputPath)
  Outputs: $(InputName)_n.cpp
  ```

  For msg files you need an analogous procedure.
- **file name extension**: MSVC 6.0 doesn't recognize `.cc` files as C++ sources. Your options are to switch to the `.cpp` extension, to convince MSVC by changing by the corresponding registry entries. Do a web search to find out what exactly you need to change.

Chapter 8

Configuring and Running Simulations

8.1 User interfaces

OMNeT++ simulations can be run under different user interfaces. Currenly, two user interfaces are supported:

- Tkenv: Tcl/Tk-based graphical, windowing user interface
- Cmdenv: command-line user interface for batch execution

You would typically test and debug your simulation under Tkenv, then run actual simulation experiments from the command line or shell script, using Cmdenv. Tkenv is also better suited for educational or demonstration purposes.

Both Tkenv and Cmdenv are provided in the form of a library, and you choose between them by linking one or the other into your simulation executable. (Creating the executable was described in chapter 7). Both user interfaces are supported on Unix and Windows platforms.

Common functionality in Tkenv and Cmdenv has been collected and placed into the Envir library, which can be thought of as the "common base class" for the two user interfaces.

The user interface is separated from the simulation kernel, and the two parts interact through a well-defined interface. This also means that, if needed, you can write your own user interface or embed an OMNeT++ simulation into your application without any change to models or the simulation library.

Configuration and input data for the simulation are described in a configuration file usually called `omnetpp.ini`. Some entries in this file apply to Tkenv or Cmdenv only, other settings are in effect regardless of the user interface. Both user interfaces accept command-line arguments, too.

The following sections explain `omnetpp.ini` and the common part of the user interfaces, describe Cmdenv and Tkenv in detail, then go on to specific problems.

8.2 The configuration file: omnetpp.ini

8.2.1 An example

For a start, let us see a simple `omnetpp.ini` file which can be used to run the Fifo1 sample simulation under Cmdenv.

```
[General]
network = fifonet1
sim-time-limit = 500000s
output-vector-file = fifo1.vec

[Cmdenv]
express-mode = yes

[Parameters]
# generate a large number of jobs of length 5..10 according to Poisson
fifonet1.gen.num_messages = 10000000
fifonet1.gen.ia_time = exponential(1)
fifonet1.gen.msg_length = intuniform(5,10)
# processing speeed of queue server
fifonet1.fifo.bits_per_sec = 10
```

The file is grouped into *sections* named `[General]`, `[Cmdenv]` and `[Parameters]`, each one containing several *entries*. The `[General]` section applies to both Tkenv and Cmdenv, and the entries in this case specify that the network named `fifonet1` should be simulated and run for 500,000 simulated seconds, and vector results should be written into the `fifo1.vec` file. The entry in the `[Cmdenv]` section tells Cmdenv to run the simulation at full speed and print periodic updates about the progress of the simulation. The `[Parameters]` section assigns values to parameters that did not get a value (or got `input` value) inside the NED files.

Lines that start with “#” or “;” are comments.

When you build the Fifo1 sample with Cmdenv and you run it by typing `fifo1` (or on Unix, `./fifo1`) on the command prompt, you should see something like this.

```
OMNeT++ Discrete Event Simulation  (C) 1992-2003 Andras Varga
See the license for distribution terms and warranty disclaimer
Setting up Cmdenv (command-line user interface)...

Preparing for Run #1...
Setting up network 'fifonet1'...
Running simulation...
** Event #0        T=0.0000000  ( 0.00s)   Elapsed: 0m  0s   ev/sec=0
** Event #100000   T=25321.99 ( 7h  2m)    Elapsed: 0m  1s   ev/sec=0
** Event #200000   T=50275.694 (13h 57m)   Elapsed: 0m  3s   ev/sec=60168.5
** Event #300000   T=75217.597 (20h 53m)   Elapsed: 0m  5s   ev/sec=59808.6
** Event #400000   T=100125.76 ( 1d  3h)   Elapsed: 0m  6s   ev/sec=59772.9
** Event #500000   T=125239.67 ( 1d 10h)   Elapsed: 0m  8s   ev/sec=60168.5
...
** Event #1700000  T=424529.21 ( 4d 21h)   Elapsed: 0m 28s   ev/sec=58754.4
** Event #1800000  T=449573.47 ( 5d  4h)   Elapsed: 0m 30s   ev/sec=59066.7
** Event #1900000  T=474429.06 ( 5d 11h)   Elapsed: 0m 32s   ev/sec=59453
** Event #2000000  T=499417.66 ( 5d 18h)   Elapsed: 0m 34s   ev/sec=58719.9
<!> Simulation time limit reached -- simulation stopped.

Calling finish() at end of Run #1...
*** Module: fifonet1.sink***
Total jobs processed: 9818
Avg queueing time:    1.8523
Max queueing time:    10.5473
Standard deviation:   1.3826
```

```
End run of OMNeT++
```

As Cmdenv runs the simulation, periodically it prints the sequence number of the current event, the simulation time, the elapsed (real) time, and the performance of the simulation (how many events are processed per second; the first two values are 0 because there wasn't enough data for it to calculate yet). At the end of the simulation, the `finish()` methods of the simple modules are run, and the output from them are displayed. On my machine this run took 34 seconds. This Cmdenv output can be customized via `omnetpp.ini` entries. The output file `fifo1.vec` contains vector data recorded during simulation (here, queueing times), and it can be processed using Plove or other tools.

8.2.2 The concept of simulation runs

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.

8.2.3 File syntax

The ini file is a text file consisting of entries grouped into different sections. The order of the sections doesn't matter. Also, if you have two sections with the same name (e.g. `[General]` occurs twice in the file), they will be merged.

Lines that start with "#" or ";" are comments, and will be ignored during processing.

Long lines can be broken up using the backslash notation: if the last character of a line is "\", it will be merged with the next line.

The size of the ini file (the number of sections and entries) is not limited. Currently there is a 1024-character limit on the line length, which *cannot* be increased by breaking up the line using backslashes. This limit might be lifted in future releases.

Example:

```
[General]
# this is a comment
foo="this is a single value \
for the foo parameter"

[General]  # duplicate sections are merged
bar="belongs to the same section as foo"
```

8.2.4 File inclusion

OMNeT++ supports including an ini file in another, via the `include` keyword. This feature allows you to partition large ini files into logical units, fixed and varying part etc.

An example:

```
# omnetpp.ini
...
include parameters.ini
include per-run-pars.ini
...
```

You can also include files from other directories. If the included ini file further includes others, their path names will be understood as relative to the location of the file which contains the reference, rather than relative to the current working directory of the simulation. This rule also applies to other file names occurring in ini files (such as the `preload-ned-files=`, `load-libs=`, `bitmap-path=`, `output-vector-file=`, `output-scalar-file=` etc entries, and `xmldoc()` module parameter values.)

8.2.5 Sections

The following sections can exist:

Section	Description
`[General]`	Contains general settings that apply to all simulation runs and all user interfaces. For details, see section 8.2.6.
`[Run 1]`, `[Run 2]`, ...	Contains per-run settings. These sections may contain any entries that are accepted in other sections.
`[Cmdenv]`	Contains Cmdenv-specific settings. For details, see section 8.7.2
`[Tkenv]`	Contains Tkenv-specific settings. For details, see section 8.8.2
`[Parameters]`	Contains values for module parameters that did not get a value (or got `input` value) inside the NED files. For details, see section 8.4
`[OutVectors]`	Configures recording of output vectors. You can specify filtering by vector names and by simulation time (start/stop recording). For details, see section 8.5

8.2.6 The [General] section

The most important options of the `[General]` section are the following.

- 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 output file names can be set with the following options: `output-vector-file`, `output-scalar-file` and `snapshot-file`.

The full list of supported options follows. Almost every one these options can also be put into the `[Run` *n*`]` sections. Per-run settings have priority over globally set ones.

Name and default value	Description
[General]	
`ini-warnings` = yes	When enabled, OMNeT++ lists the names of ini file entries for which the default values were used. This can at times be useful for debugging ini files.
`preload-ned-files` =	List of NED files to be loaded dynamically (see 8.3).
`network` =	The name of the network to be simulated.
`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.

`sim-time-limit` =	Duration of the simulation in simulation time.
`cpu-time-limit` =	Duration of the simulation in real time.
`num-rngs` = 1	Number of random number generators.
`rng-class` = `"cMersenneTwister"`	The RNG class to be used. It can be `"cMersenneTwister"`, `"cLCG32"`, or `"cAkaroaRNG"`, or you can use your own RNG class (it must be subclassed from `cRNG`).
`seed-N-mt` =, `seed-N-lcg32` =	Specifies seeds for the cMersenneTwister and the cLCG32 RNGs (substitute N with the RNG number: 0, 1, 2...); default is auto seed selection. This obsoletes the *random-seed*= and *gen0-seed*=, *gen1-seed*=, etc. entries which are no longer in use.
`total-stack-kb` =	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.
`debug-on-errors` = false	When set to `true`, runtime errors will cause the simulation program to break into the C++ debugger (if the simulation is running under one, or just-in-time debugging is activated). Once in the debugger, you can view the stack trace or examine variables.
`load-libs` =	List of shared libraries (separated by spaces) to load in the initialization phase. OMNeT++ appends a platform-specific extension to the library name: `.dll` on Windows and `.so` on Unix systems. This feature can be used to dynamically load Envir extensions (RNGs, output vector managers, etc.) or simple modules. Example: `load-libs = "../lib/rng2 ../lib/ospfrouting"`
`perform-gc` = false	If `true`, the simulation kernel will `delete` on network cleanup the simulation objects not deleted by simple module destructors. Not recommended because it may cause crashes under certain scenarios. See 4.3.5. *New!*
`print-undisposed` = true	When perform-gc is `false` (default setting), it selects whether simulation objects not deleted by simple module destructors should be reported by the simulation kernel. *New!*
`output-scalar-precision` = `12`	Adjusts the number of significant digits recorded into the output scalar file. See 6.9.3 for a discussion. *New!*
`output-vector-precision` = `12`	Adjusts the number of significant digits recorded into the output vector file. See 6.9.3 for a discussion. *New!*
`fname-append-host` = false	Turning it on will cause the host name and process Id to be appended to the names of output files (e.g. `omnetpp.vec`, `omnetpp.sca`). This is especially useful for parallel distributed simulation (chapter 12).

`parallel-simulation` = false	Enables parallel distributed simulation (see chapter 12).
`scheduler-class` = `cSequentialScheduler`	Part of the Envir plugin mechanism: selects the scheduler class. This plugin interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation. The class has to implement the `cScheduler` interface defined in `envirext.h`. More details in section 13.5.3.
`configuration-class` = `cInifile`	Part of the Envir plugin mechanism: selects the class from which all configuration will be obtained. In other words, this option lets you replace `omnetpp.ini` with some other implementation, e.g. database input. The simulation program still has to bootstrap from an `omnetpp.ini` though (which contains the `configuration-class` setting). The class has to implement the `cConfiguration` interface defined in `envirext.h`. More details in section 13.5.3.
`outputvectormanager-class` = `cFileOutputVectorManager`	Part of the Envir plugin mechanism: selects 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`. More details in section 13.5.3.
`outputscalarmanager-class` = `cFileOutputScalarManager`	Part of the Envir plugin mechanism: selects 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`. More details in section 13.5.3.
`snapshotmanager-class` = `cFileSnapshotManager`	Part of the Envir plugin mechanism: selects the class to handle streams to which `snapshot()` writes its output. The class has to implement the `cSnapshotManager` interface defined in `envirext.h`. More details in section 13.5.3.

8.3 Dynamic NED loading

Prior to OMNeT++ 3.0, NED files had to be translated into C++ by the NED compiler, compiled and linked into the simulation program. From OMNeT++ 3.0 up, one can use dynamic NED loading, which means that a simulation program can load NED files at runtime when it starts – compiling NED files into the simulation program is no longer necessary. This results in more flexibility, and can also save model development time.

The key is the `preload-ned-files=` configuration option in the `[General]` section of `omnetpp.ini`. This option should list the names of the NED files to be loaded when the simulation program starts.

Example:

```
[General]
```

```
preload-ned-files = host.ned router.ned networks/testnetwork1.ned
```

Wildcards can also be used:

```
[General]
preload-ned-files = *.ned networks/*.ned
```

It is also possible to use list files, with the `@` notation:

```
[General]
preload-ned-files = *.ned @../nedfiles.lst
```

where the `nedfiles.lst` file contains the list of NED files, one per line, like this:

```
transport/tcp/tcp.ned
transport/udp/udp.ned
network/ip/ip.ned
```

The Unix `find` command is often a very convenient way to create list files (try `find . -name '*.ned' > listfile.lst`).

Moreover, the list file can also contain wildcards, and references to other list files:

```
transport/tcp/*.ned
transport/udp/*.ned
@moreprotocols.lst
```

Files given with relative paths are relative to the location of the list file (and not to the current working directory). That is, the `transport` directory and `moreprotocols.lst` in the example above are expected to be in the same directory as `nedfiles.lst`, whatever the current working directory is.

It is important to note, that the loaded NED files may contain any number of modules, channel and *any number of networks* as well. It does not matter whether you use all or just some of them in the simulations. You will be able to select `any` of the networks that occur in the loaded NED files using the `network=` `omnetpp.ini` entry, and as long as every module, channel etc for it has been loaded, network setup will be successful.

8.4 Setting module parameters in omnetpp.ini

Simulations get input via module parameters, which can be assigned a value in NED files or in `omnetpp.ini` – in this order. Since parameters assigned in NED files cannot be overridden in omnetpp.ini, one can think about them as being "hardcoded". In contrast, it is easier and more flexible to maintain module parameter settings in omnetpp.ini.

In omnetpp.ini, module parameters are referred to by their full paths or hiearchical names. This name consists of the dot-separated list of the module names (from the top-level module down to the module containg the parameter), plus the parameter name (see section 6.1.5).

An example `omnetpp.ini` which sets the `numHosts` parameter of the toplevel module and the `transactionsPerS` parameter of the `server` module:

```
[Parameters]
net.numHosts = 15
net.server.transactionsPerSecond = 100
```

8.4.1 Run-specific and general sections

Values for module parameters can be placed 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.

Though runs are identified by numbers, you can assign them short descriptive labels, which will get displayed e.g. in the Tkenv run selection dialog. Just place a `description="some text"` line under the `[Run x]` heading.

An example `omnetpp.ini` (everything after # is a comment):

```
[Parameters]
net.numHosts = 15
net.server.transactionsPerSecond = 100

[Run 1]
description="general settings"
# uses settings from the [Parameters] section

[Run 2]
description="higher transaction rate"
net.server.transactionsPerSecond = 150  # overrides the value in [Parameters]
# net.numHosts comes from the [Parameters] section

[Run 3]
description="more hosts and higher transaction rate"
# override both setting in [Parameters]
net.numHosts = 20
net.server.transactionsPerSecond = 150
```

8.4.2 Using wildcard patterns

Models can have a large number of parameters to be configured, and it would be tedious to set them one-by-one in `omnetpp.ini`. OMNeT++ supports *wildcards patterns* which allow for setting several model parameters at once.

The notation is a variation on the usual glob-style patterns. The most apperent differences to the usual rules are the distinction between `*` and `**`, and that character ranges should be written with curly braces instead of square brackets (that is, *any-letter* is `{a-zA-Z}` not `[a-zA-Z]`, because square brackets are already reserved for the notation of module vector indices).

Pattern syntax:

- `?` : matches any character except dot (.)
- `*` : matches zero or more characters except dot (.)
- `**` : matches zero or more character (any character)
- `{a-f}` : *set*: matches a character in the range a-f
- `{^a-f}`: *negated set*: matches a character NOT in the range a-f
- `{38..150}` : *numeric range*: any number (i.e. sequence of digits) in the range 38..150 (e.g. `99`)
- `[38..150]` : *index range*: any number in square brackets in the range 38..150 (e.g. `[99]`)
- backslash (\) : takes away the special meaning of the subsequent character

Precedence

If you use wildcards, the order of entries is important: if a parameter name matches several wildcards-patterns, the *first* matching occurrence is used. This means that you need to list specific settings first, and more general ones later. Catch-all settings should come last.

An example ini file:

```
[Parameters]
*.host[0].waitTime = 5ms   # specifics come first
*.host[3].waitTime = 6ms
*.host[*].waitTime = 10ms  # catch-all comes last
```

Asterisk vs double asterisk

The `*` wildcard is for matching a single module or parameter name in the path name, while `**` can be used to match several components in the path. For example, `**.queue*.bufSize` matches the `bufSize` parameter of any module whose name begins with `queue` in the model, while `*.queue*.bufSize` or `net.queue*.bufSize` selects only queues immediately on network level. Also note that `**.queue**.bufSize` would match `net.queue1.foo.bar.bufSize` as well!

Sets, negated sets

Sets and negated sets can contain several character ranges and also enumeration of characters. For example, `{_a-zA-Z0-9}` matches any letter or digit, plus the underscore; `{xyzc-f}` matches any of the characters x, y, z, c, d, e, f. To include '-' in the set, put it at a position where it cannot be interpreted as character range, for example: `{a-z-}` or `{-a-z}`. If you want to include '}' in the set, it must be the first character: `{}a-z}`, or as a negated set: `{^}a-z}`. A backslash is always taken as literal backslash (and NOT as escape character) within set definitions.

Numeric ranges and index ranges

Only nonnegative integers can be matched. The start or the end of the range (or both) can be omitted: `{10..}`, `{..99}` or `{..}` are valid numeric ranges (the last one matches any number). The specification must use exactly two dots. Caveat: `*{17..19}` will match `a17`, `117` and `963217` as well, because the `*` can also match digits!

An example for numeric ranges:

```
[Parameters]
*.*.queue[3..5].bufSize = 10
*.*.queue[12..].bufSize = 18
*.*.queue[*].bufSize = 6  # this will only affect queues 0,1,2 and 6..11
```

Compatibility

In OMNeT++ versions prior to 3.0, the `**` wildcard did not exist, and `*` matched dot as well. This means that ini files written for earlier OMNeT++ versions may have a different meaning when used in OMNeT++ 3.0 – so ini files have to be updated. In practice, every line which begins with `*.` should be changed to begin with `**.` – that'll do most of the time, further tweaking is rarely necessary.

If you still want to run the old `omnetpp.ini` (for example, to check the new one against it), you can add the line

```
#% old-wildcards
```

at the top of (each) old ini file. This will switch back to the old behaviour. Since `#% old-wildcards` is only provided to ease transition from OMNeT++ 2.3 to 3.0, it will be removed in some future version.

8.4.3 Applying the defaults

It is also possible to utilize the default values specified with `input(`*default-value*`)` in the NED files. The <*parameter-name*>`.use-default=yes` setting assigns the default value to the parameter, or 0, false or empty string if there was no default value in the NED file.

The following example sets `ttl` (time-to-live) of `hostA`'s `ip` module to 5, while all other nodes in the network will get the default specified with `input()` in the NED files.

```
[Parameters]
**.hostA.ip.ttl = 5
**.ip.ttl.use-default = yes
```

To make use of *all* defaults in NED files, you'd add the following to `omnetpp.ini`:

```
[Parameters]
**.use-default = yes
```

8.5 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.

Output vectors can be configured with the following syntax:

```
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");
```

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.

8.6 Configuring the random number generators

The random number architecture of OMNeT++ was already outlined in section 6.4. Here we'll cover the configuration of RNGs in `omnetpp.ini`.

8.6.1 Number of RNGs

The `num-rngs=` configuration entry sets the number of random number generator instances (i.e. random number streams) available for the simulation model (see 6.4). Referencing an RNG number greater or equal to this number (from a simple module or NED file) will cause a runtime error.

8.6.2 RNG choice

The `rng-class=` configuration entry sets the random number generator class to be used. It defaults to `"cMersenneTwister"`, the Mersenne Twister RNG. Other available classes are `"cLCG32"` (the "legacy" RNG of OMNeT++ 2.3 and earlier versions, with a cycle length of $2^{31} - 2$), and `"cAkaroaRNG"` (Akaroa's random number generator, see section 8.10).

8.6.3 RNG mapping

The RNG numbers used in simple modules may be arbitrarily mapped to the actual random number streams (actual RNG instances) from `omnetpp.ini`. The mapping allows for great flexibility in RNG usage and random number streams configuration – even for simulation models which were not written with RNG awareness.

RNG mapping may be specified in `omnetpp.ini`. The syntax of configuration entries is the following.

```
[General]
<modulepath>.rng-N=M  (where N,M are numeric, M<num-rngs)
```

This maps module-local RNG N to physical RNG M. The following example maps all `gen` module's default (N=0) RNG to physical RNG 1, and all `noisychannel` module's default (N=0) RNG to physical RNG 2.

```
[General]
num-rngs = 3
**.gen[*].rng-0 = 1
**.noisychannel[*].rng-0 = 2
```

This mapping allows variance reduction techniques to be applied to OMNeT++ models, without any model change or recompilation.

8.6.4 Automatic seed selection

Automatic seed selection gets used for an RNG if you don't explicitly specify seeds in omnetpp.ini. Automatic and manual seed selection can co-exist: for a particular simulation, some RNGs can be configured manually, and some automatically.

The automatic seed selection mechanism uses two inputs: the *run number* (i.e. the number in the `[Run 1]`, `[Run 2]`, etc. section names), and the *RNG number*. For the same the run number and RNG number, OMNeT++ always selects the same seed value for any simulation model. If the run number or the RNG number is different, OMNeT++ does its best to choose different seeds which are also sufficiently apart in the RNG's sequence so that the generated sequences don't overlap.

The run number can be specified either in in omnetpp.ini (e.g. via the `[Cmdenv]/runs-to-execute=` entry) or on the command line:

```
./mysim -r 1
./mysim -r 2
./mysim -r 3
```

For the `cMersenneTwister` random number generator, selecting seeds so that the generated sequences don't overlap is easy, due to the extremely long sequence of the RNG. The RNG is initialized from the 32-bit seed value $seed = runNumber * numRngs + rngNumber$. (This implies that simulation runs participating in the study should have the same number of RNGs set). [1]

For the `cLCG32` random number generator, the situation is more difficult, because the range of this RNG is rather short ($2^{31} - 1$, about 2 billion). For this RNG, OMNeT++ uses a table of 256 pre-generated seeds, equally spaced in the RNG's sequence. Index into the table is calculated with the $runNumber * numRngs + rngNumber$ formula. Care should be taken that one doesn't exceed 256 with the index, or it will wrap and the same seeds will be used again. It is best not to use the `cLCG32` at all – `cMersenneTwister` is superior in every respect.

8.6.5 Manual seed configuration

In some cases you may want manually configure seed values. Reasons for doing that may be that you want to use variance reduction techniques, or you may want to use the same seeds for several simulation runs.

For the cLCG32 RNG, OMNeT++ provides a standalone program to generate seed values (`seedtool` is discussed in section 8.6.6), and you can specify those 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:

```
[General]
rng-class=cLCG32  # needed because the default is cMersenneTwister
num-rngs = 2

[Run 1]
seed-0-lcg32 = 1768507984
seed-1-lcg32 = 33648008

[Run 2]
seed-0-lcg32 = 1082809519
seed-1-lcg32 = 703931312
...
```

[1]While (to our knowledge) no one has proven that the seeds 0,1,2,... are well apart in the sequence, this is probably true, due to the extremely long sequence of MT. The author would however be interested in papers published about seed selection for MT.

To manually set seeds for the Mersenne Twister RNG (which should seldom, if ever, be necessary), use the `seed-0-mt=`, `seed-1-mt=`, etc settings:

```
[General]
num-rngs = 2

[Run 1]
seed-0-mt = 1317366363
seed-1-mt = 1453732904

[Run 2]
...
```

To set a seed value for all runs, place the necessary seed entries into the `[General]` section.

8.6.6 Choosing good seed values: the seedtool utility

The `seedtool` program can be used for selecting seeds for the cLCG32 RNG. When started without command-line arguments, the program prints out the following help:

```
seedtool - part of OMNeT++/OMNEST, (C) 1992-2004 Andras Varga
See the license for distribution terms and warranty disclaimer.

Generates seeds for the LCG32 random number generator. This RNG has a
period length of 2^31-2, which makes about 2,147 million random numbers.
Note that Mersenne Twister is also available in OMNeT++, which has a
practically infinite period length (2^19937).

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 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.

8.7 Cmdenv: the command-line interface

The command line user interface is a small, portable and fast user interface that compiles and runs on all platforms. Cmdenv is designed primarily for batch execution.

Cmdenv uses simply executes some or all simulation runs that are described in the configuration file. If one run stops with an error message, subsequent ones will still be executed. The runs to be executed can be passed via command-line argument or in the ini file.

8.7.1 Command-line switches

A simulation program built with Cmdenv accepts the following command line switches:

`-h`	The program prints a short help message and the networks contained in the executable, then 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*>	It specifies which runs should be executed (e.g. `-r 2,4,6-8`). 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:

```
% ./fddi -h

OMNeT++/OMNEST Discrete Event Simulation  (C) 1992-2005 Andras Varga
See the license for distribution terms and warranty disclaimer
Setting up Tkenv...

Command line options:
  -h            Print this help and exit.
  -f <inifile>  Use the given ini file instead of omnetpp.ini. Multiple
                -f options are accepted to load several ini files.
  -u <ui>       Selects the user interface. Standard choices are Cmdenv
                and Tkenv. To make a user interface available, you need
                to link the simulation executable with the cmdenv/tkenv
                library, or load it as shared library via the -l option.
  -l <library>  Load the specified shared library (.so or .dll) on startup.
                The file name should be given without the .so or .dll suffix
                (it will be appended automatically.) The loaded module may
                contain simple modules, plugins, etc. Multiple -l options
                can be present.

Tkenv-specific options:
  -r <run>      Set up the given run, specified in a [Run n] section of
                the ini file.

The following components are available:
  module types:
    FDDI_Monitor
    FDDI_Generator4Sniffer
    FDDI_Generator4Ring
    ...

End run of OMNeT++
```

8.7.2 Cmdenv ini file options

Cmdenv can be executed in two modes, selected by the `express-mode` ini file entry:

- **Normal** (non-express) mode is for debugging: detailed information will be written to the standard output (event banners, module output, etc).
- **Express** mode can be used for long simulation runs: only periodical status update is displayed about the progress of the simulation.

The full list of ini file options recognized by Cmdenv:

Entry and default value	**Description**
[Cmdenv]	
`runs-to-execute` =	Specifies which simulation runs should be executed. It accepts a comma-separated list of run numbers or run number ranges, e.g. `1,3-4,7-9`. 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.
`express-mode`=yes/no (default: no)	Selects "normal" (debug/trace) or "express" mode.
`module-messages`=yes/no (default: yes)	In normal mode only: printing module ev« output on/off
`event-banners`=yes/no (default: yes)	In normal mode only: printing event banners on/off
`message-trace`=yes/no (default: no)	In normal mode only: print a line about each message sending (by `send()`,`scheduleAt()`, etc) and delivery on the standard output
`autoflush`=yes/no (default: no)	Call `fflush(stdout)` after each event banner or status update; affects both express and normal mode. Turning on autoflush can be useful with printf-style debugging for tracking down program crashes.

`status-frequency`=<integer> (default: 50000)	In express mode only: print status update every n events (on today's computers, and for a typical model, this will produce an update every few seconds, perhaps a few times per second)
`performance-display`=yes/no (default: yes)	In express mode only: print detailed performance information. Turning it on results in a 3-line entry printed on each update, containing ev/sec, simsec/sec, ev/simsec, number of messages created/still present/currently scheduled in FES.
`extra-stack-kb` = 8	Specifies the extra amount of stack (in kilobytes) that is reserved for each `activity()` simple module when the simulation is run under Cmdenv.

8.7.3 Interpreting Cmdenv output

When the simulation is running in "express" mode with detailed performance display enabled, Cmdenv periodically outputs a three-line status info about the progress of the simulation. The output looks like this:

```
...
** Event #250000   T=123.74354 ( 2m  3s)    Elapsed: 0m 12s
     Speed:     ev/sec=19731.6   simsec/sec=9.80713   ev/simsec=2011.97
     Messages:  created: 55532   present: 6553   in FES: 8
** Event #300000   T=148.55496 ( 2m 28s)    Elapsed: 0m 15s
     Speed:     ev/sec=19584.8   simsec/sec=9.64698   ev/simsec=2030.15
     Messages:  created: 66605   present: 7815   in FES: 7
...
```

The first line of the status display (beginning with `**`) contains:

- how many events have been processed so far
- the current simulation time (T), and
- the elapsed time (wall clock time) since the beginning of the simulation run.

The second line displays info about simulation performance:

- `ev/sec` indicates *performance*: how many events are processed in one real-time second. On one hand it depends on your hardware (faster CPUs process more events per second), and on the other hand it depends on the complexity (amount of calculations) associated with processing one event. For example, protocol simulations tend to require more processing per event than e.g. queueing networks, thus the latter produce higher ev/sec values. In any case, this value is independent of the size (number of modules) in your model.

- `simsec/sec` shows *relative speed* of the simulation, that is, how fast the simulation is progressing compared to real time, how many simulated seconds can be done in one real second. This value virtuall depends on everything: on the hardware, on the size of the simulation model, on the complexity of events, and the average simulation time between events as well.

- `ev/simsec` is the *event density*: how many events are there per simulated second. Event density only depends on the simulation model, regardless of the hardware used to simulate it: in a cell-level ATM simulation you'll have very hight values (10^9), while in a bank teller simulation this value is probably well under 1. It also depends on the size of your model: if you double the number of modules in your model, you can expect the event density double, too.

The third line displays the number of messages, and it is important because it may indicate the 'health' of your simulation.

- `Created`: total number of message objects created since the beginning of the simulation run. This does not mean that this many message object actually exist, because some (many) of them may have been deleted since then. It also does not mean that *you* created all those messages – the simulation kernel also creates messages for its own use (e.g. to implement `wait()` in an `activity()` simple module).
- `Present`: the number of message objects currently present in the simulation model, that is, the number of messages created (see above) minus the number of messages already deleted. This number includes the messages in the FES.
- `In FES`: the number of messages currently scheduled in the Future Event Set.

The second value, the number of messages present is more useful than perhaps one would initially think. It can indicator of the 'health' of the simulation: if it is growing steadily, then either you have a memory leak and losing messages (which indicates a programming error), or the network you simulate is overloaded and queues are steadily filling up (which might indicate wrong input parameters).

Of course, if the number of messages does not increase, it does not mean that you do *not* have a memory leak (other memory leaks are also possible). Nevertheless the value is still useful, because by far the most common way of leaking memory in a simulation is by not deleting messages.

8.8 Tkenv: the graphical user interface

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.

Tkenv is built with Tcl/Tk, and it work on all platforms where Tcl/Tk has been ported to: Unix/X, Windows, Macintosh. You can get more information about Tcl/Tk on the Web pages listed in the Reference.

8.8.1 Command-line switches

A simulation program built with Tkenv accepts the following command line switches:

`-h`	The program prints a short help message and the networks contained in the executable, then 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` *<run-number>*	It has the same effect as (but takes priority over) the `[Tkenv]/default-run=` ini file entry.

8.8.2 Tkenv ini file settings

Tkenv accepts the following settings in the `[Tkenv]` section of the ini file.

Entry and default value	Description
[Tkenv]	
`extra-stack-kb` = 48	Specifies the extra amount of stack (in kilobytes) that is reserved for each *`activity()`* simple module when the simulation is run under Tkenv. This value is significantly higher than the similar one for Cmdenv – handling GUI events requires a large amount of stack space.
`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.

The following configuration entries are of marginal usefulness, because corresponding settings are also accessible in the Simulation options dialog in the Tkenv GUI, and the GUI settings take precedence. Tkenv stores the settings in the `.tkenvrc` file in the current directory – the ini file settings are only used if there is no `.tkenvrc` file.

Entry and default value	Description

[Tkenv]	
`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.
`methodcalls-delay` =	Sets delay after method call animation.
`show-layouting` = true	Show layouting process of network graphics.

8.8.3 Using the graphical environment

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.

Tkenv has a status bar which is regularly updated while the simulation is running. The gauges displayed are similar to those in Cmdenv, described in section 8.7.3.

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.

Tkenv inspectors also display the object pointer, and can also copy the pointer value to the clipboard. This can be invaluable for debugging: when the simulation is running under a debugger like gdb or the MSVC IDE, you can paste the object pointer into the debugger and have closer look at the data structures.

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`. (See section **??** as well).

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.

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`. This possibility is built into the makefiles and can be optionally enabled.

8.8.4 In Memoriam...

There used to be other windowing 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...

8.9 Repeating or iterating simulation runs

Once your model works reliably, you'll usually want to run several simulations. You may want to run the model with various parameter settings, or you may want *(should want?)* to run the same model with the same parameter settings but with different random number generator seeds, to achieve statistically more reliable results.

Running a simulation several times by hand can easily become tedious, and then a good solution is to write a control script that takes care of the task automatically. Unix shell is a natural language choice to write the control script in, but other languages like Perl, Matlab/Octave, Tcl, Ruby might also have justification for this purpose.

The next sections are only for Unix users. We'll use the Unix 'Bourne' shell (`sh`, `bash`) to write the control script. If you'd prefer Matlab/Octave, the `contrib/octave/` directory contains example scripts (contributed by Richard Lyon).

8.9.1 Executing several runs

In simple cases, you may define all simulation runs needed in the `[Run 1]`, `[Run 2]`, etc. sections of `omnetpp.ini`, and invoke your simulation with the -r flag each time. The -f flag lets you use a file name different from `omnetpp.ini`.

The following script executes a simulation named `wireless` several times, with parameters for the different runs given in the `runs.ini` file.

```
#! /bin/sh
./wireless -f runs.ini -r 1
./wireless -f runs.ini -r 2
./wireless -f runs.ini -r 3
./wireless -f runs.ini -r 4
...
./wireless -f runs.ini -r 10
```

To run the above script, type it in a text file called e.g. `run`, give it `x` (executable) permission using `chmod`, then you can execute it by typing `./run`:

```
% chmod +x run
% ./run
```

You can simplify the above script by using a *for* loop. In the example below, the variable `i` iterates through the values of list given after the `in` keyword. It is very practical, since you can leave out or add runs, or change the order of runs by simply editing the list – to demonstrate this, we skip run 6, and include run 15 instead.

```
#! /bin/sh
for i in 3 2 1 4 5 7 15 8 9 10; do
   ./wireless -f runs.ini -r $i
done
```

If you have many runs, you can use a C-style loop:

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

8.9.2 Variations over parameter values

It may not be practical to hand-write descriptions of all runs in an ini file, especially if there are many parameter settings to try, or you want to try all possible combinations of two or more parameters. The solution might be to generate only a small fraction of the ini file with the variable parameters, and use it via ini file inclusion. For example, you might write your `omnetpp.ini` like this:

```
[General]
network = Wireless

[Parameters]
Wireless.n = 10
...   # other fixed parameters
include params.ini  # include variable part
```

And have the following as control script. It uses two nested loops to explore all possible combinations of the *alpha* and *beta* parameters. Note that `params.ini` is created by redirecting the `echo` output into file, using the > and » operators.

```
#! /bin/sh
for alpha in 1 2 5 10 20 50; do
   for beta in 0.1 0.2 0.3 0.4 0.5; do
       echo "Wireless.alpha=$alpha" > params.ini
       echo "Wireless.beta=$beta" >> params.ini
       ./wireless
   done
done
```

As a heavy-weight example, here's the "runall" script of Joel Sherrill's *File System Simulator*. It also demonstrates that loops can iterate over string values too, not just numbers. (`omnetpp.ini` includes the generated `algorithms.ini`.)

Note that instead of redirecting every `echo` command to file, they are grouped using parentheses, and redirected together. The net effect is the same, but you can spare some typing this way.

```
#! /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
```

8.9.3 Variations over seed value (multiple independent runs)

The same kind of control script can be used if you want to execute several runs with different random seeds. The following code does 500 runs with independent seeds. (`omnetpp.ini` should include `parameters.ini`.)

The seeds are 10 million numbers apart in the sequence (`seedtool` parameter), so one run should not use more random numbers than this, otherwise there will be overlaps in the sequences and the runs will not be independent.

```
#! /bin/sh
seedtool g 1 10000000 500 > seeds.txt
for seed in `cat seeds.txt`; do
   (
     echo "[General]"
     echo "random-seed = ${seed}"
     echo "output-vector-file = xcube-${seed}.vec"
   ) > parameters.ini
   ./xcube
done
```

8.10 Akaroa support: Multiple Replications in Parallel

8.10.1 Introduction

Typical simulations are Monte-Carlo simulations: they use (pseudo-)random numbers to drive the simulation model. For the simulation to produce statistically reliable results, one has to carefully consider the following:

- When is the initial transient over, when can we start collecting data? We usually do not want to include the initial transient when the simulation is still "warming up."
- When can we stop the simulation? We want to wait long enough so that the statistics we are collecting can "stabilize", can reach the required sample size to be statistically trustable.

Neither questions are trivial to answer. One might just suggest to wait "very long" or "long enough". However, this is neither simple (how do you know what is "long enough"?) nor practical (even with today's high speed processors simulations of modest complexity can take hours, and one may not afford multiplying runtimes by, say, 10, "just to be safe.") If you need further convincing, please read [PJL02] and be horrified.

A possible solution is to look at the statistics while the simulation is running, and decide at runtime when enough data have been collected for the results to have reached the required accuracy. One possible criterion is given by the confidence level, more precisely, by its width relative to the mean. But ex ante it is unknown how many observations have to be collected to achieve this level – it must be determined runtime.

8.10.2 What is Akaroa

Akaroa [EPM99] addresses the above problem. According to its authors, Akaroa (Akaroa2) is a "fully automated simulation tool designed for running distributed stochastic simulations in MRIP scenario" in a cluster computing environment.

MRIP stands for *Multiple Replications in Parallel*. In MRIP, the computers of the cluster run independent replications of the whole simulation process (i.e. with the same parameters but different seed for the

RNGs (random number generators)), generating statistically equivalent streams of simulation output data. These data streams are fed to a global data analyser responsible for analysis of the final results and for stopping the simulation when the results reach a satisfactory accuracy.

The independent simulation processes run independently of one another and continuously send their observations to the central analyser and control process. This process *combines* the independent data streams, and calculates from these observations an overall estimate of the mean value of each parameter. Akaroa2 decides by a given confidence level and precision whether it has enough observations or not. When it judges that is has enough observations it halts the simulation.

If n processors are used, the needed simulation execution time is usually n times smaller compared to a one-processor simulation (the required number of observations are produced sooner). Thus, the simulation would be sped up approximately in proportion to the number of processors used and sometimes even more.

Akaroa was designed at the University of Canterbury in Christchurch, New Zealand and can be used free of charge for teaching and non-profit research activities.

8.10.3 Using Akaroa with OMNeT++

Akaroa

Before the simulation can be run in parallel under Akaroa, you have to start up the system:

- Start `akmaster` running in the background on some host.
- On each host where you want to run a simulation engine, start `akslave` in the background.

Each `akslave` establishes a connection with the `akmaster`.

Then you use `akrun` to start a simulation. `akrun` waits for the simulation to complete, and writes a report of the results to the standard output. The basic usage of the `akrun` command is:

```
akrun -n num_hosts command [argument..]
```

where *command* is the name of the simulation you want to start. Parameters for Akaroa are read from the file named `Akaroa` in the working directory. Collected data from the processes are sent to the `akmaster` process, and when the required precision has been reached, `akmaster` tells the simulation processes to terminate. The results are written to the standard output.

The above description is not detailed enough help you set up and successfully use Akaroa – for that you need to read the Akaroa manual.

Configuring OMNeT++ for Akaroa

First of all, you have to compile OMNeT++ with Akaroa support enabled.

The OMNeT++ simulation must be configured in `omnetpp.ini` so that it passes the observations to Akaroa. The simulation model itself does not need to be changed – it continues to write the observations into output vectors (`cOutVector` objects, see chapter 6). You can place some of the output vectors under Akaroa control.

You need to add the following to `omnetpp.ini`:

```
[General]
rng-class="cAkaroaRNG"
outputvectormanager-class="cAkOutputVectorManager"
```

These lines cause the simulation to obtain random numbers from Akaroa, and allows data written to selected output vectors to be passed to Akaroa's global data analyser. [2]

Akaroa's RNG is a Combined Multiple Recursive pseudorandom number generator (CMRG) with a period of approximately 2^{191} random numbers, and provides a unique stream of random numbers for every simulation engine. It is vital to obtain random numbers from Akaroa: otherwise, all simulation processes would run with the same RNG seeds, and produce exactly the same results!

Then you need to specify which output vectors you want to be under Akaroa control. By default, all output vectors are under Akaroa control; the

```
<modulename>.<vectorname>.akaroa=false
```

setting can be used to make Akaroa ignore specific vectors. You can use the `*`, `**` wildcards here (see section 8.4.2). For example, if you only want a few vectors be placed under Akaroa, you can use the following trick:

```
<modulename>.<vectorname1>.akaroa=true
<modulename>.<vectorname2>.akaroa=true
...
**.*.akaroa=false  # catches everything not matched above
```

Using shared file systems

It is usually practical to have the same physical disk mounted (e.g. via NFS or Samba) on all computers in the cluster. However, because all OMNeT++ simulation processes run with the same settings, they would overwrite each other's output files (e.g. `omnetpp.vec`, `omnetpp.sca`). Your can prevent this from happening using the `fname-append-host` ini file entry:

```
[General]
fname-append-host=yes
```

When turned on, it appends the host name to the names of the output files (output vector, output scalar, snapshot files).

8.11 Typical issues

8.11.1 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. The solution is to increase the stack for that module type. 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*"

The resolution depends on whether you are using OMNeT++ on Unix or on Windows.

Unix. 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`:

[2]For more details on the plugin mechanism these settings make use of, see section 13.5.3.

```
[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).

Windows. You need to set a *low* (!) "reserved stack size" in the linker options, for example 64K (/stack:65536 linker flag) will do. The "reserved stack size" is an attribute in the Windows exe files' internal header. It can be set from the linker, or with the `editbin` Microsoft utility. You can use the `opp_stacktool` program (which relies on another Microsoft utility called `dumpbin`) to display reserved stack size for executables.

You need a low reserved stack size because the Win32 Fiber API which is the mechanism underlying `activity()` uses this number as coroutine stack size, and with 1MB being the default, it is easy to run out of the 2GB possible address space (2GB/1MB=2048).

A more detailed explanation follows. Each fiber has its own stack, by default 1MB (this is the "reserved" stack space – i.e. reserved in the address space, but not the full 1MB is actually "committed", i.e. has physical memory assigned to it). This means that a 2GB address space will run out after 2048 fibers, which is way too few. (In practice, you won't even be able to create this many fibers, because physical memory is also a limiting factor). Therefore, the 1MB reserved stack size (RSS) must be set to a smaller value: the coroutine stack size requested for the module, plus the `extra-stack-kb` amount for Cmdenv/Tkenv – which makes about 16K with Cmdenv, and about 48K when using Tkenv. Unfortunately, the CreateFiber() Win32 API doesn't allow the RSS to be specified. The more advanced CreateFiberEx() API which accepts RSS as parameter is unfortunately only available from Windows XP.

The alternative is the stacksize parameter stored in the EXE header, which can be set via the STACKSIZE .def file parameter, via the /stack linker option, or on an existing executable using the editbin /stack utility. This parameter specifies a common RSS for the main program stack, fiber and thread stacks. 64K should be enough. This is the way simulation executable should be created: linked with the /stack:65536 option, or the /stack:65536 parameter applied using editbin later. For example, after applying the editbin /stacksize:65536 command to dyna.exe, I was able to successfully run the Dyna sample with 8000 Client modules on my Win2K PC with 256M RAM (that means about 12000 modules at runtime, including about 4000 dynamically created modules.)

"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 2.4.x, the default stack limit is 8192K (that is, 8MB). The `ulimit` shell command can be used to modify the resource limits, and you can raise the allowed maximum stack size up to 64M.

```
$ ulimit -s 65500
$ ulimit -s
65500
```

Further increase is only possible if you're root. Resource limits are inherited by child processes. The following sequence can be used under Linux to get a shell with 256M stack limit:

```
$ su root
Password:
# ulimit -s 262144
# su andras
$ ulimit -s
262144
```

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.

Eventually...

Once you get to the point where you have to adjust the total stack size to get your program running, you should probably consider transforming (some of) your `activity()` simple modules to `handleMessage()`. `activity()` does not scale well for large simulations.

8.11.2 Memory leaks and crashes

The most common problems in C++ are associated with memory allocation (usage of `new` and `delete`):

- *memory leaks,* that is, forgetting to delete objects or memory blocks no longer used;
- *crashes,* usually due to referring to an already deleted object or memory block, or trying to delete one for a second time;
- *heap corruption* (enventually leading to crash) due to overrunning allocated blocks, i.e. writing past the end of an allocated array.

By far the most common ways leaking memory in simulation programs is by not deleting messages (`cMessage` objects or subclasses). Both Tkenv and Cmdenv are able to display the number of messages currently in the simulation, see e.g. section 8.7.3. If you find that the number of messages is steadily increasing, you need to find where the message objects are. You can do so by selecting *Inspect|From list of all objects...* from the Tkenv menu, and reviewing the list in the dialog that pops up. (If the model is large, it may take a while for the dialog to appear.)

If the number of messages is stable, it is still possible you're leaking other `cObject`-based objects. You can also find them using Tkenv's *Inspect|From list of all objects...* function.

If you're leaking non-`cObject`-based objects or just memory blocks (`struct`s, `int`/`double`/`struct` arrays, etc, allocated by `new`), you cannot find them via Tkenv. You'll probably need a specialized memory debugging tool like the ones described below.

Memory debugging tools

If you suspect that you may have memory allocation problems (crashes associated with double-deletion or accessing already deleted block, or memory leaks), you can use specialized tools to track them down.

By far the most efficient, most robust and most versatile tool is *Valgrind*, originally developed for debugging KDE.

Other memory debuggers are *NJAMD*, *MemProf*, *MPatrol*, *dmalloc* and *ElectricFence*. Most of the above tools support tracking down memory leaks as well as detecting double deletion, writing past the end of an allocated block, etc.

A proven commercial tool *Rational Purify*. It has a good reputation and proved its usefulness many times.

8.11.3 Simulation executes slowly

What can you do if the simulation executes much slower than you expect? The best advice that can be given here is that you should **use a good profiler** to find out how much time is spent in each part of the program. Do not make the mistake of omitting this step, thinking that you know "which part is slow"! Even for experienced programmers, profiling session is all too often full of surprises. It often turns out that lots of CPU time is spent in completely innocent-looking statements, while the big and complex algorithm doesn't take nearly as much time as expected. *Don't assume anything – profile before you optimize!* [3]

A really impressive profiler on Linux is the *Valgrind*-based *callgrind*, and its visualizer *KCachegrind*. Unfortunately it won't be ported to Windows anytime soon. On Windows, you're out of luck – commercial products may help, or, port your simulation to Linux. The latter goes usually much smoother than one would expect.

[3]And before blaming the simulation kernel for poor performance...

Chapter 9

Network Graphics And Animation

9.1 Display strings

9.1.1 Display string syntax

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:

- *submodules* – display string may contain position, arrangement (for module vectors), icon, icon color, auxiliary icon, status text, communication range (as circle or filled circle), etc.
- *connections* – display string can specify positioning, arrow color, arrow thickness
- *compound modules* – display string can specify background color, border color, border thickness
- *messages* – display string can specify icon, icon color, etc.

The following NED sample shows where to place display strings in the code.

```
module ClientServer
    submodules:
        pc: Host;
            display: "p=66,55;i=comp"; // position and icon
        server: Server;
            display: "p=135,73;i=server1";
    connections:
        pc.out --> server.in
            display "m=m,61,40,41,28"; // note missing ":"
        server.out --> pc.in
            display "m=m,15,57,35,69";
    display: "o=#ffffff"; // affects background
endmodule
```

9.1.2 Submodule display strings

The following table lists the 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 the ellipse's bounding box at (*xpos*,*ypos*), with the *width* and *height*. Defaults: *width,height* are chosen so that submodules do not overlap. **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. For color notation, see section 9.2. Defaults: *fillcolor*=#8080ff (a lightblue), *outline-color*=black, *borderwidth*=2
i=*iconname*,*color*,*percentage*	Use the named icon. It can be colorized, and percentage specifies the amount of colorization. Defaults: *iconname*: no default – if no icon name is present, *box* is used; *color*: no coloring; *percent-age*: 30%
is=*size*	Specifies the size of the icon. *size* can be one of `l`, `vl`, `s` and `vs` (for large, very large, small, very small). If this option is present, size cannot be included in the icon name (`"i="` tag) with the `"i=`*`<iconname>_<size>`*`"` notation.
i2=*iconname*,*color*,*percentage*	Displays a small "modifier" icon at the top right corner of the primary icon. Suggested icons are `status/busy`, `status/down`, `status/up`, `status/asleep`, etc. The arguments are analoguous with those of `"i="`.
r=*radius*,*fillcolor*,*color*,*width*	Draws a circle (or a filled circle) around the submodule with the given radius. It can be used to visualize transmission range of wireless nodes. Defaults: *radius*=100, *fillcolor*=none, *color*=black, *width*=1 (unfilled black circle)
q=*queue-object-name*	Displays the queue length next to submodule icon. It expects a `cQueue` object's name (as set by the `setName()` method, see section 6.1.4). Tkenv will do a depth-first search to find the object, and it will find the queue object within submodules as well.
t=*text*,*pos*,*color*	Displays a short text above or next to the icon. The text is meant to convey status information (*"up"*, *"down"*, *"5Kb in buffer"*) or statistics (*"4 pks received"*). *pos* can be `"l"`, `"r"` or `"t"` for left, right and top. Defaults: *pos*="t", *color*=blue
tt=*tooltip-text*	Displays the given text in a tooltip when the user moves the mouse over the icon. This complements the `t=` tag, and lets you display more information that otherwise would not fit on the screen.

Examples:

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

9.1.3 Background display strings

Compound module display strings specify the background. They can contain the following tags:

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. For color notation, see section 9.2. Defaults: *fillcolor*=#8080ff (a lightblue), *outlinecolor*=black, *borderwidth*=2
tt=*tooltip-text*	Displays the given text in a tooltip when the user moves the mouse over the module name in the top-left corner.

9.1.4 Connection display strings

Tags that can be 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. For color notation, see section 9.2. Defaults: *color*=black, *width*=2

t=*text*,*color*	Displays a short text near the connection arrow. The text may convey status information or connection properties (*"down"*, *"100Mb"*) or statistics. Defaults: *color*=#005030
tt=*tooltip-text*	Displays the given text in a tooltip when the user moves the mouse over the connection arrow. This complements the `t=` tag, and lets you display more information that otherwise would not fit on the screen.

Examples:

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

9.1.5 Message display strings

Message objects do not store a display string by default, but you can redefine the `cMessage`'s `displayString()` method and make it return one.

```
const char *CustomPacket::displayString() const
{
    return "i=msg/packet_vs";
}
```

This display string affects how messages are shown during animation. By default, they are displayed as a small filled circle, in one of 8 basic colors (the color is determined as *message kind modulo 8*), and with the message class and/or name displayed under it The latter is configurable in the Tkenv Options dialog, and message kind dependent coloring can also be turned off there.

The following tags can be used in message display strings:

Tag	Meaning
b=*width*,*height*,**oval**	Ellipse with the given *height* and *width*. Defaults: *width*=10, *height*=10
b=*width*,*height*,**rect**	Rectangle with the given *height* and *width*. Defaults: *width*=10, *height*=10
o=*fillcolor*,*outlinecolor*,*borderwidth*	Specifies options for the rectangle or oval. For color notation, see section 9.2. Defaults: *fillcolor*=red, *outlinecolor*=black, *borderwidth*=1
i=*iconname*,*color*,*percentage*	Use the named icon. It can be colorized, and percentage specifies the amount of colorization. If color name is "`kind`", a message kind dependent colors is used (like default behaviour). Defaults: *iconname*: no default – if no icon name is present, a small red solid circle will be used; *color*: no coloring; *percentage*: 30%
tt=*tooltip-text*	Displays the given text in a tooltip when the user moves the mouse over the message icon.

Examples:

```
"i=penguin"

"b=15,15,rect;o=white,kind,5"
```

9.2 Colors

9.2.1 Color names

Any valid Tk color specification is accepted: English color names (blue, lightgray, wheat) or *#rgb*, *#rrggbb* format (where *r*,*g*,*b* are hex digits).

It is also possible to specify colors in HSB (hue-saturation-brightness) as *@hhssbb* (with *h*, *s*, *b* being hex digits). HSB makes it easier to scale colors e.g. from white to bright red.

You can produce a transparent background by specifying a hyphen ("-") as color.

9.2.2 Icon colorization

The `"i="` display string tag allows for colorization of icons. It accepts a target color and a percentage as the degree of colorization. Percentage has no effect if the target color is missing. Brightness of icon is also affected – to keep the original brightness, specify a color with about 50#008000 mid-green).

Examples:

- `"i=device/server,gold"` creates a gold server icon
- `"i=misc/globe,#808080,100"` makes the icon grayscale
- `"i=block/queue,white,100"` yields a "burnt-in" black-and-white icon

Colorization works with both submodule and message icons.

9.3 The icons

9.3.1 The bitmap path

In the current OMNeT++ version, module icons are GIF files. The icons shipped with OMNeT++ are in the `bitmaps/` subdirectory. Both the GNED editor and Tkenv need the exact location of this directory to load the icons.

Icons are loaded from all directories in the *bitmap path*, a semicolon-separated list of directories. The default bitmap path is compiled into GNED and Tkenv with the value `"`*`omnetpp-dir`*`/bitmaps;./bitmaps"` – which will work fine as long as you don't move the directory, and you'll also be able to load more icons from the `bitmaps/` subdirectory of the current directory. As people usually run simulation models from the model's directory, this practically means that custom icons placed in the `bitmaps/` subdirectory of the model's directory are automatically loaded.

The compiled-in bitmap path can be overridden with the `OMNETPP_BITMAP_PATH` environment variable. The way of setting environment variables is system specific: in Unix, if you're using the bash shell, adding a line

```
export OMNETPP_BITMAP_PATH="/home/you/bitmaps;./bitmaps"
```

to /.bashrc or /.bash_profile will do; on Windows, environment variables can be set via the *My Computer –> Properties* dialog.

You can also add to the bitmap path from `omnetpp.ini`, with the `bitmap-path` setting:

```
[Tkenv]
bitmap-path = "/home/you/model-framework/bitmaps;/home/you/extra-bitmaps"
```

The value should be quoted, otherwise the first semicolon separator will be interpreted as comment sign, which will result in the rest of the directories being ignored.

9.3.2 Categorized icons

Since OMNeT++ 3.0, icons are organized into several categories, represented by folders. These categories include:

- block/ - icons for subcomponents (queues, protocols, etc).
- device/ - network devices: servers, hosts, routers, etc.
- abstract/ - symbolic icons for various devices
- misc/ - node, subnet, cloud, building, town, city, etc.
- msg/ - icons that can be used for messages

Old (pre-3.0) icons are in the `old/` folder.

Tkenv and GNED now load icons from subdirectories of all directories of the bitmap path, and these icons can be referenced from display strings by naming the subdirectory (subdirectories) as well: `"subdir/icon"`, `"subdir/subdir2/icon"`, etc.

For compatibility, if the display string contains a icon without a category (i.e. subdirectory) name, OMNeT++ tries it as "old/icon" as well.

9.3.3 Icon size

Icons come in various sizes: normal, large, small, very small. Sizes are encoded into the icon name's suffix: `_l`, `_s`, `_vs`. In display strings, one can either use the suffix (`"i=device/router_l"`), or the `"is"` (*icon size*) display string tag ("i=device/router;is=l").

9.4 Layouting

OMNeT++ implements an automatic layouting feature, using a variation of the SpringEmbedder algorithm. Modules which have not been assigned explicit positions via the `"p="` tag will be automatically placed by the algorithm.

SpringEmbedder is a graph layouting algorithm based on a physical model. Graph nodes (modules) repent each other like electric charges of the same sign, and connections are sort of springs which try to contract and pull the nodes they're attached to. There is also friction built in, in order to prevent oscillation of the nodes. The layouting algorithm simulates this physical system until it reaches equilibrium (or times out). The physical rules above have been slightly tweaked to get better results.

The algorithm doesn't move any module which has fixed coordinates. Predefined row, matrix, ring or other arrangements (defined via the 3rd and further args of the `"p="` tag) will be preserved – you can think about them as if those modules were attached to a wooden framework so that they can only move as one unit.

Caveats:

- If the full graph is too big after layouting, it is scaled back so that it fits on the screen, *unless it contains any fixed-position module*. (For obvious reasons: if there's a module with manually specified position, we don't want to move that one). To prevent rescaling, you can specify a sufficiently large bounding box in the background display string, e.g. `"b=2000,3000"`.
- Size is ignored by the present layouter, so longish modules (such as an Ethernet segment) may produce funny results.
- The algorithm is prone to produce erratic results, especially when the number of submodules is small, or when using predefined (matrix, row, ring, etc) layouts. The "Re-layout" toobar button can then be very useful. Larger networks usually produce satisfactory results.

Parameters to the layouter algoritm (repulsive/attractive forces, number of iterations,random number seed) can be specified via the `"l="` background display string tag. Its current arguments are (with default values): `"l=`*`<repulsion>`*`=10,`*`<attraction>`*`=0.3,` *`<edgelen>`*`=40,`*`<maxiter>`*`=500,`*`<rng-seed>`*`"`. The `"l="` tag is somewhat experimental and its arguments may change in further releases.

9.5 GNED – Graphical NED Editor

The GNED editor allows you to design compound modules graphically. GNED works directly with NED files – it doesn't have any 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. Other components 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". 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 extensively by removing/adding submodules, gates, parameters, connections, etc.

GNED is 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.

9.5.1 Keyboard and 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

9.6 Enhancing animation

9.6.1 Changing display strings at runtime

Often it is useful to manipulate the display string at runtime. Changing colors, icon, or text may convey status change, and changing a module's position is useful when simulating mobile networks.

Display strings are stored in `cDisplayString` objects inside modules and gates. `cDisplayString` also lets you manipulate the string.

To get a pointer to the `cDisplayString` object, you can call the module's `displayString()` method:

```
cDisplayString *dispStr = displayString();
```

```
cDisplayString *bgDispStr = parentModule()->backgroundDisplayString();
```

```
cDisplayString *gateDispStr = gate("out")->displayString();
```

As far as `cDisplayString` is concerned, a display string (e.g. `"p=100,125;i=cloud"`) is a string that consist of several *tags* separated by semicolons, and each tag has a *name* and after an equal sign, zero or more *arguments* separated by commas.

The class facilitates tasks such as finding out what tags a display string has, adding new tags, adding arguments to existing tags, removing tags or replacing arguments. The internal storage method allows very fast operation; it will generally be faster than direct string manipulation. The class doesn't try to interpret the display string in any way, nor does it know the meaning of the different tags; it merely parses the string as data elements separated by semicolons, equal signs and commas.

An example:

```
dispStr->parse("a=1,2;p=alpha,,3");
dispStr->insertTag("x");
dispStr->setTagArg("x",0,"joe");
dispStr->setTagArg("x",2,"jim");
dispStr->setTagArg("p",0,"beta");
ev << dispStr->getString();  // result: "x=joe,,jim;a=1,2;p=beta,,3"
```

9.6.2 Bubbles

Modules can let the user know about important events (such as a node going down or coming up) by displaying a bubble with a short message ("Going down", "Coming up", etc.) This is done by the `bubble()` method of `cModule`. The method takes the string to be displayed as a `const char *` pointer.

An example:

```
bubble("Going down!");
```

If the module contains a lot of code that modifies the display string or displays bubbles, it is recommended to make these calls conditional on `ev.isGUI()`. The `ev.isGUI()` call returns *false* when the simulation is run under Cmdenv, so one can make the code skip potentially expensive display string manipulation.

```
if (ev.isGUI())
    bubble("Going down!");
```

Chapter 10

Analyzing Simulation Results

10.1 Output vectors

Output vectors are time series data: values with timestamps. You can use output vectors to record end-to-end delays or round trip times of packets, queue lengths, queueing times, link utilization, the number of dropped packets, etc. – anything that is useful to get a full picture of what happened in the model during the simulation run.

Output vectors are recorded from simple modules, by `cOutVector` objects (see section 6.9.1). Since output vectors usually record a large amount of data, in `omnetpp.ini` you can disable vectors or specify a simulation time interval for recording (see section 8.5).

All `cOutVector` objects write to the same, common file. The following sections describe the format of the file, and how to process it.

10.1.1 Plotting output vectors with Plove

Plove features

Typically, you'll get output vector files as a result of a simulation. Data written to `cOutVector` objects from simple modules are written to output vector files. You can use Plove to look into the output vector files and plot vectors from them.

Plove is a handy tool for plotting OMNeT++ output vectors. Line type (lines, dots etc) for each vector can be set as well as the most frequent drawing options like axis bounds, scaling, titles and labels. You can save the graphs to files (as Encapsulated Postscript or raster formats such as GIF) with a click. On Windows, you can also copy the graph to the clipboard in a vector format (Windows metafile) and paste it into other applications. [1]

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 create new filters by parameterizing and aggregating existing ones. You can apply several filters to a vector.

On startup, Plove automatically reads the `.ploverc` file in your home directory. The file contains general application settings, including the custom filters you created.

[1]Note: prior to OMNeT++ 3.0, Plove has been a front-end to gnuplot. This older version of Plove is no longer supported, but it is still available in the OMNeT++ source distribution.

Usage

First, you load an output vector file (`.vec`) into the left pane. You can copy vectors from the left pane to the right pane by clicking the button with the right arrow icon in the middle. The PLOT button will initiate plotting 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 a filter, click 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. These changes will not affect the vector files on disk. (Plove never modifies the output vector files themselves.) 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.

10.1.2 Format of output vector files

An output vector file contains several series of data produced during simulation. The file is textual, and 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"  "queue length"  1
2  16.960  2.00000000000.63663666
1  23.086  2355.66666666
2  24.026  8.00000000000.44766536
```

There two types of lines: vector declaration lines (beginning with the word `vector`), and data lines. A *vector declaration line* introduces a new output vector, and its columns are: vector Id, module of creation, name of `cOutVector` object, and multiplicity (usually 1). Actual data recorded in this vector are on *data lines* which begin with the vector Id. Further columns on data lines are the simulation time and the recorded value.

10.1.3 Working without Plove

In case you have a large number of repeated experiments, you'll probably want to automate processing of the output vector files. OMNeT++ lets you use any tool you see fit for this purpose, because the output vector files are text files and their format is simple enough to be processed by common tools such as *perl*, *awk*, *octave*, etc.

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 "queue length" 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"  "queue length"  1
```

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 digest for 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.

Using splitvec

The `splitvec` script (part of OMNeT++) automates the process described in the previous section: it breaks the vector file into several files which contain one vector each. The command

```
% splitvec mysim.vec
```

would create the files `mysim1.vec`, `mysim2.vec` etc. with contents similar to the following:

```
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"  "queue length"  1
16.960  2.00000000000.63663666
24.026  8.00000000000.44766536
```

As you can see, the vector Id column has been stripped from the files. The resulting files can be directly loaded e.g. into spreadsheets or other programs (10.3).

10.2 Scalar statistics

Output vectors capture the transient behaviour of the simulation run. However, to compare model behaviour under various parameter settings, output scalars are more useful.

10.2.1 Format of output scalar files

Scalar results are recorded with `recordScalar()` calls, usually from the `finish()` methods of modules, with code like this:

```
void EtherMAC::finish()
{
    double t = simTime();
    if (t==0) return;

    recordScalar("simulated time", t);
    recordScalar("rx channel idle (%)", 100*totalChannelIdleTime/t);
    recordScalar("rx channel utilization (%)", 100*totalSuccessfulRxTxTime/t);
    recordScalar("rx channel collision (%)", 100*totalCollisionTime);

    recordScalar("frames sent",    numFramesSent);
    recordScalar("frames rcvd",    numFramesReceivedOK);
    recordScalar("bytes sent",     numBytesSent);
    recordScalar("bytes rcvd",     numBytesReceivedOK);
    recordScalar("collisions",     numCollisions);

    recordScalar("frames/sec sent", numFramesSent/t);
    recordScalar("frames/sec rcvd", numFramesReceivedOK/t);
    recordScalar("bits/sec sent",   8*numBytesSent/t);
    recordScalar("bits/sec rcvd",   8*numBytesReceivedOK/t);
}
```

The corresponding output scalar file (by default, `omnetpp.sca`) will look like this:

```
run 1 "lan"
scalar "lan.hostA.mac" "simulated time"             120.249243
scalar "lan.hostA.mac" "rx channel idle (%)"        97.5916992
scalar "lan.hostA.mac" "rx channel utilization (%)" 2.40820676
scalar "lan.hostA.mac" "rx channel collision (%)"   0.011312
scalar "lan.hostA.mac" "frames sent"                99
scalar "lan.hostA.mac" "frames rcvd"                3088
scalar "lan.hostA.mac" "bytes sent"                 64869
scalar "lan.hostA.mac" "bytes rcvd"                 3529448
scalar "lan.hostA.mac" "frames/sec sent"            0.823290006
scalar "lan.hostA.mac" "frames/sec rcvd"            25.6799953
scalar "lan.hostA.mac" "bits/sec sent"              4315.63632
scalar "lan.hostA.mac" "bits/sec rcvd"              234808.83
scalar "lan.hostB.mac" "simulated time"             120.249243
scalar "lan.hostB.mac" "rx channel idle (%)"        97.5916992
scalar "lan.hostB.mac" "rx channel utilization (%)" 2.40820676
scalar "lan.hostB.mac" "rx channel collision (%)"   0.011312
[...]
scalar "lan.hostC.mac" "simulated time"             120.249243
scalar "lan.hostC.mac" "rx channel idle (%)"        97.5916992
scalar "lan.hostC.mac" "rx channel utilization (%)" 2.40820676
scalar "lan.hostC.mac" "rx channel collision (%)"   0.011312
[...]

run 2 "lan"
scalar "lan.hostA.mac" "simulated time"             235.678665
[...]
```

Every `recordScalar()` call generates one "scalar" line in the file. (If you record statistics objects (`cStatictic` subclasses such as `cStdDev`) via their `recordScalar()` methods, they'll generate several lines: mean, standard deviation, etc.) In addition, several simulation runs can record their results into a single file – this facilitates comparing them, creating x-y plots (*offered load vs throughput*-type diagrams), etc.

10.2.2 The Scalars tool

The `Scalars` program can be used to visualize the contents of the `omnetpp.sca` file. It can draw bar charts, x-y plots (e.g. throughput vs offered load), or export data via the clipboard for more detailed analysis into spreadsheets or other programs.

You can open a scalar file either from the Scalars program's menu or by specifying it as a command-line argument to Scalars.

The program displays the data in a table with columns showing the file name, run number, module name where it was recorded, and the value. There're usually too many rows to get an overview, so you can filter by choosing from (or editing) the three combo boxes at the top. (The filters also accept `*`, `**` wildcards.)

You could actually load further scalar files into the window, and thus analyse them together.

You can copy the selected rows to the clipboard by Edit|Copy or the corresponding toolbar button, and paste them e.g. into OpenOffice Calc, MS Excel or Gnumeric.

The bar chart toolbar button creates – well – a bar chart in a new window. You can customize the chart by right-clicking on it and choosing from the context menu. It can also be exported to EPS, GIF, or as metafile via the Windows clipbard (the latter is not available on Unix of course).

10.3 Analysis and visualization tools

Output vector files (or files produced by `splitvec`) and output scalar files can be analysed and/or plotted by a number of applications in addition to `Plove` and `Scalars`. These programs can produce output in various forms (on the screen, as PostScript, in various image formats, etc.)

One straightforward solution is to import or paste them into spreadsheet programs such as OpenOffice Calc, Microsoft Excel or GNOME Gnumeric. These programs have good charting and statistical features, but the number of rows is usually limited to about 32,000..64,000. One useful functionality spreadsheets offer for analysing scalar files is known as *PivotTable* in Excel, and as *DataPilot* in in OpenOffice. The easiest way to import scalar files into them is via copy/paste from Scalars.

Alternatively, one can use numerical packages such as *Octave*, *Matlab* or the statistics package *R*. In addition to their support for statistical computations, they can also create various plots.

There are also open-source programs directly for plotting, *Gnuplot* still being the most commonly used one. Other, potentially more powerful ones include *Grace*, *ROOT* and *PlotMTV*.

10.3.1 Grace

Grace (also known as *xmgrace*, a successor of *ACE/gr* or *Xmgr*) is a GPL-ed powerful data visualization program with a WYSIWIG point-and-click graphical user interface. It was developed for Unix, but there is a Windows version, too.

You load the appropriate file by selecting it in a dialog box. The icon bar and menu commands can be used to customize the graph.

As of June 2003, Grace 1.5.12 can export graphics to (E)PS, PDF, MIF, SVG, PNM, JPEG and PNG formats. It has many useful features like built-in statistics and analysis functions (e.g. correlation, histogram), fitting, splines, etc., and it also sports its own built-in programming language.

10.3.2 ROOT

ROOT is a powerful object-oriented data analysis framework, with strong support for plotting and graphics in general. ROOT was developed at CERN, and is distributed under a BSD-like license.

ROOT is based on *CINT*, a “C/C++ interpreter” aimed at processing C/C++ scripts. It is probably harder to get started using ROOT than with either Gnuplot or Grace, but if you are serious about analysing simulation results, you will find that ROOT provides power and flexibility that would be unattainable the other two programs.

Curt Brune’s page at Stanford (http://www.slac.stanford.edu/ curt/omnet++/) shows examples what you can achieve using ROOT with OMNeT++.

10.3.3 Gnuplot

Gnuplot has an interactive command interface. To plot the data in `mysim1.vec` and `mysim4.vec` (produced by `splitvec`) 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
```

Several commands are available to adjust ranges, plotting style, labels, scaling etc. Gnuplot can also plot 3D graphs. Gnuplot is available for Windows and other platforms. On Windows, you can copy the resulting graph to the clipboard from the Gnuplot window’s system menu, then insert it into the application you are working with.

Chapter 11

Documenting NED and Messages

11.1 Overview

OMNeT++ provides a tool which can generate HTML documentation from NED files and message definitions. Like Javadoc and Doxygen, `opp_neddoc` makes use of source code comments. `opp_neddoc`-generated documentation lists simple and compound modules, and presents their details including description, gates, parameters, unassigned submodule parameters and syntax-highlighted source code. The documentation also includes clickable network diagrams (exported via the GNED graphical editor) and module usage diagrams as well as inheritance diagrams for messages.

`opp_neddoc` works well with Doxygen, which means that it can hyperlink simple modules and message classes to their C++ implementation classes in the Doxygen documentation. If you also generate the C++ documentation with some Doxygen features turned on (such as *inline-sources* and *referenced-by-relation*, combined with *extract-all*, *extract-private* and *extract-static*), the result is an easily browsable and very informative presentation of the source code. Of course, one still has to write documentation comments in the code.

11.2 Authoring the documentation

11.2.1 Documentation comments

Documentation is embedded in normal comments. All `//` comments that are in the "right place" (from the documentation tool's point of view) will be included in the generated documentation. [1]

Example:

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//
simple Gen
    parameters:
        destAddress: string,  // destination MAC address
        protocolId: numeric,  // value for SSAP/DSAP in Ethernet frame
        waitMean: numeric;    // mean for exponential interarrival times
    gates:
```

[1] In contrast, Javadoc and Doxygen use special comments (those beginning with `/**`, `///`, `//<` or a similar marker) to distinguish documentation from "normal" comments in the source code. In OMNeT++ there's no need for that: NED and the message syntax is so compact that practically all comments one would want to write in them can serve documentation purposes. Still, there is a way to write comments that *don't* make it into the documentation – by starting them with `//#`.

```
        out: out;             // to Ethernet LLC
endsimple
```

You can also place comments above parameters and gates. This is useful if they need long explanations. Example:

```
//
// Deletes packets and optionally keeps statistics.
//
simple Sink
    parameters:
        // You can turn statistics generation on and off. This is
        // a very long comment because it has to be described what
        // statistics are collected (or not).
        statistics: bool;
    gates:
        in: in;
endsimple
```

If you want a comment line *not* to appear in the documentation, begin it with `//#`. Those lines will be ignored by the documentation generation, and can be used to comment out unused NED code or to make "private" comments like `FIXME` or `TBD`.

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//# FIXME above description needs to be refined
//
simple Gen
    parameters:
        destAddress: string,  // destination MAC address
        protocolId: numeric,  // value for SSAP/DSAP in Ethernet frame
        //# burstiness: numeric;  -- not yet supported
        waitMean: numeric;    // mean for exponential interarrival times
    gates:
        out: out;             // to Ethernet LLC
endsimple
```

11.2.2 Text layout and formatting

If you write longer descriptions, you'll need text formatting capabilities. Text formatting works like in Javadoc or Doxygen – you can break up the text into paragraphs and create bulleted/numbered lists without special commands, and use HTML for more fancy formatting.

Paragraphs are separated by empty lines, like in LaTeX or Doxygen. Lines beginning with '–' will be turned into bulleted lists, and lines beginning with '`-#`' into numbered lists.

Example:

```
//
// Ethernet MAC layer. MAC performs transmission and reception of frames.
//
// Processing of frames received from higher layers:
// - sends out frame to the network
// - no encapsulation of frames -- this is done by higher layers.
// - can send PAUSE message if requested by higher layers (PAUSE protocol,
//   used in switches). PAUSE is not implemented yet.
//
// Supported frame types:
// -# IEEE 802.3
// -# Ethernet-II
//
```

11.2.3 Special tags

`OMNeT++_neddoc` understands the following tags and will render them accordingly: `@author`, `@date`, `@todo`, `@bug`, `@see`, `@since`, `@warning`, `@version`. An example usage:

```
//
// @author Jack Foo
// @date 2005-02-11
//
```

11.2.4 Additional text formatting using HTML

Common HTML tags are understood as formatting commands. The most useful of these tags are: `<i>..</i>` (italic), `<b>..</b>` (bold), `<tt>..</tt>` (typewriter font), `<sub>..</sub>` (subscript), `<sup>..</sup>` (superscript), `<br>` (line break), `<h3>` (heading), `<pre>..</pre>` (preformatted text) and `<a href=..>..</a>` (link), as well as a few other tags used for table creation (see below). For example, `<i>Hello</i>` will be rendered as "*Hello*" (using an italic font).

The complete list of HTML tags interpreted by `opp_neddoc` are: `<a>`, `<b>`, `<body>`, `<br>`, `<center>`, `<caption>`, `<code>`, `<dd>`, `<dfn>`, `<dl>`, `<dt>`, `<em>`, `<form>`, `<font>`, `<hr>`, `<h1>`, `<h2>`, `<h3>`, `<i>`, `<input>`, `<img>`, `<li>`, `<meta>`, `<multicol>`, `<ol>`, `<p>`, `<small>`, `<span>`, `<strong>`, `<sub>`, `<sup>`, `<table>`, `<td>`, `<th>`, `<tr>`, `<tt>`, `<kbd>`, `<ul>`, `<var>`.

Any tags not in the above list will not be interpreted as formatting commands but will be printed verbatim – for example, `<what>bar</what>` will be rendered literally as "<what>bar</what>" (unlike HTML where unknown tags are simply ignored, i.e. HTML would display "bar").

If you insert links to external pages (web sites), its useful to add the `target="_blank"` attribute to ensure pages come up in a new browser window and not just in the current frame which looks awkward. (Alternatively, you can use the `target="_top"` attribute which replaces all frames in the current browser).

Examples:

```
//
// For more info on Ethernet and other LAN standards, see the
// <a href="http://www.ieee802.org/" target="_blank">IEEE 802
// Committee's site</a>.
//
```

You can also use the `<a href=..>` tag to create links within the page:

```
//
// See the <a href="#resources">resources</a> in this page.
// ...
// <a name="resources"><b>Resources</b></a>
// ...
//
```

You can use the `<pre>..</pre>` HTML tag to insert souce code examples into the documentation. Line breaks and indentation will be preserved, but HTML tags continue to be interpreted (or you can turn them off with `<nohtml>`, see later).

Example:

```
// <pre>
// // my preferred way of indentation in C/C++ is this:
// <b>for</b> (<b>int</b> i=0; i<10; i++)
// {
//     printf(<i>"%d\n"</i>, i);
// }
// </pre>
```

will be rendered as

```
// my preferred way of indentation in C/C++ is this:
for (int i=0; i<10; i++)
{
    printf("%d\n", i);
}
```

HTML is also the way to create tables. The example below

```
//
// <table border="1">
//   <tr>  <th>#</th> <th>number</th> </tr>
//   <tr>  <td>1</td> <td>one</td>    </tr>
//   <tr>  <td>2</td> <td>two</td>    </tr>
//   <tr>  <td>3</td> <td>three</td>  </tr>
// </table>
//
```

will be rendered approximately as:

#	number
1	one
2	two
3	three

11.2.5 Escaping HTML tags

Sometimes may need to off interpreting HTML tags (`<i>`, `<b>`, etc.) as formatting instructions, and rather you want them to appear as literal `<i>`, `<b>` texts in the documentation. You can achieve this via surrounding the text with the `<nohtml>`...`</nohtml>` tag. For example,

```
// Use the <nohtml><i></nohtml> tag (like <tt><nohtml><i>this</i></nohtml><tt>)
// to write in <i>italic</i>.
```

will be rendered as “Use the <i> tag (like `<i>this</i>`) to write in *italic*.”

`<nohtml>`...`</nohtml>` will also prevent `opp_neddoc` from hyperlinking words that are accidentally the same as an existing module or message name. Prefixing the word with a backslash will achieve the same. That is, either of the following will do:

```
// In <nohtml>IP</nohtml> networks, routing is...

// In \IP networks, routing is...
```

Both will prevent hyperlinking the word *IP* if you happen to have an `IP` module in the NED files.

11.2.6 Where to put comments

You have to put the comments where nedtool will find them. This is a) above the documented item, or b) after the documented item, on the same line.

If you put it above, make sure there's no blank line left between the comment and the documented item. Blank lines detach the comment from the documented item.

Example:

```
// This is wrong! Because of the blank line, this comment is not
// associated with the following simple module!

simple Gen
    parameters:
    ...
endsimple
```

Do not try to comment groups of parameters together. The result will be awkward.

11.2.7 Customizing the title page

The title page is the one that appears in the main frame after opening the documentation in the browser. By default it contains a boilerplate text with the generic title *"OMNeT++ Model Documentation"*. You probably want to customize that, and at least change the title to the name of the documented simulation model.

You can supply your own version of the title page adding a `@titlepage` directive to a file-level comment (a comment that appears at the top of a NED file, but is separated from the first `import`, `channel`, `module`, etc. definition by at least one blank line). In theory you can place your title page definition into any NED or MSG file, but it is probably a good idea to create a separate `index.ned` file for it.

The lines you write after the `@titlepage` line up to the next `@page` line (see later) or the end of the comment will be used as the title page. You probably want to begin with a title because the documentation tool doesn't add one (it lets you have full control over the page contents). You can use the `<h1>..</h1>` HTML tag to define a title.

Example:

```
//
// @titlepage
// <h1>Ethernet Model Documentation</h1>
//
// This documents the Ethernet model created by David Wu and refined by Andras
// Varga at CTIE, Monash University, Melbourne, Australia.
//
```

11.2.8 Adding extra pages

You can add new pages to the documentation in a similar way as customizing the title page. The directive to be used is `@page`, and it can appear in any file-level comment (see above).

The syntax of the `@page` directive is the following:

```
// @page filename.html, Title of the Page
```

Please choose a file name that doesn't collide with the files generated by the documentation tool (such as `index.html`). The page title you supply will appear on the top of the page as well as in the page index.

The lines after the `@page` line up to the next `@page` line or the end of the comment will be used as the page body. You don't need to add a title because the documentation tool automatically adds one.

Example:

```
//
// @page structure.html, Directory Structure
//
// The model core model files and the examples have been placed
// into different directories. The <tt>examples/</tt> directory...
//
//
// @page examples.html, Examples
// ...
//
```

You can create links to the generated pages using standard HTML, using the `<a href="...">...</a>` tag. All HTML files are placed in a single directory, so you don't have to worry about specifying directories.

Example:

```
//
// @titlepage
// ...
// The structure of the model is described <a href="structure.html">here</a>.
//
```

11.2.9 Incorporating externally created pages

You may want to create pages outside the documentation tool (e.g. using a HTML editor) and include them in the documentation. This is possible, all you have to do is declare such pages with the `@externalpage` directive in any of the NED files, and they will be added to the page index. The pages can then be linked to from other pages using the HTML `<a href="...">...</a>` tag.

The `@externalpage` directive is similar in syntax `@page`:

```
// @externalpage filename.html, Title of the Page
```

The documentation tool does not check if the page exists or not. It is your responsibility to copy them manually into the directory of the generated documentation and then to make sure the hyperlinks work.

11.3 Invoking opp_neddoc

The `opp_neddoc` tool accepts the following command-line options:

```
opp_neddoc - NED and MSG documentation tool, part of OMNeT++
(c) 2003-2004 Andras Varga
```

```
Generates HTML model documentation from .ned and .msg files.

Usage: opp_neddoc options files-or-directories ...
 -a, --all    process all *.ned and *.msg files recursively
              ('opp_neddoc -a' is equivalent to 'opp_neddoc .')
 -o <dir>     output directory, defaults to ./html
 -t <filename>, --doxytagfile <filename>
              turn on generating hyperlinks to Doxygen documentation;
              <filename> specifies name of XML tag file generated by Doxygen
 -d <dir>, --doxyhtmldir <dir>
              directory of Doxygen-generated HTML files, relative to the
              opp_neddoc output directory (-o option). -t option must also be
              present to turn on linking to Doxygen. Default: ../api-doc/html
 -n, --no-figures
              do not generate diagrams
 -p, --no-unassigned-pars
              do not document unassigned parameters
 -x, --no-diagrams
              do not generate usage and inheritance diagrams
 -z, --no-source
              do not generate source code listing
 -s, --silent suppress informational messages
 -g, --debug  print invocations of external programs and other info
 -h, --help   displays this help text

Files specified as arguments are parsed and documented. For directories as
arguments, all .ned and .msg files under them (in that directory subtree) are
documented. Wildcards are accepted and they are NOT recursive, e.g.
foo/*.ned does NOT process files in foo/bar/ or any other subdirectory.

Bugs:  (1) handles only files with .ned and .msg extensions, other files are
silently ignored; (2) does not filter out duplicate files (they will show up
multiple times in the documentation); (3) on Windows, file names are handled
case sensitively.
```

11.3.1 Multiple projects

The generated `tags.xml` can be used to generate other documentation that refers to pages in this documentation via HTML links.

11.4 How does opp_neddoc work?

`*.ned` and `*.msg` files are collected (e.g. via the `find` command if you used the `-a` option on Unix) and processed with `nedtool`. `nedtool` parses them and outputs the resulting syntax tree in XML – a single large XML file which contains all files.

The `*.ned` files are processed with the `-c` (export-diagrams-and-exit) option of `gned`. This causes `gned` to export diagrams for the compound modules in Postscript. Postscript files are then converted to GIFs using `convert` (part of the ImageMagick package). `gned` also exports an `images.xml` file which describes which image was generated from which compound module, and also contains additional info (coordinates of submodule rectangles and icons in the image) for creating clickable image maps.

The XML file containing parsed NED and message files is then processed with an XSLT stylesheet to generate HTML. XSLT is a very powerful way of converting an XML document into another XML (or HTML, or text) document. Additionally, the stylesheet reads `images.xml` and uses its contents to make the compound module images clickable. The stylesheet also outputs a `tags.xml` file which describes what is documented in which .html file, so that external documentation can link to this one.

As a final step, the comments in the generated HTML file are processed with a perl script. The perl script also performs syntax hightlighting of the source listings in the HTML, and puts hyperlinks on module, channel, message, etc. names. (It uses the info in the `tags.xml` file for the latter task.) This last step, comment formatting and source code coloring whould have been very difficult to achieve from XSLT, which (at least in its 1.0 version of the standard) completely lacks powerful string manipulation functions. (Not even simple find/replace is supported in strings, let alone regular expressions. Perhaps the 2.0 version of XSLT will improve on this.)

The whole process is controlled by the `opp_neddoc` script.

Chapter 12

Parallel Distributed Simulation

12.1 Introduction to Parallel Discrete Event Simulation

OMNeT++ supports parallel execution of large simulations. The following paragraphs provide a brief picture of the problems and methods of parallel discrete event simulation (PDES). Interested readers are strongly encouraged to look into the literature.

For parallel execution, the model is to be partitioned into several LPs (logical processes) that will be simulated independently on different hosts or processors. Each LP will have its own local Future Event Set, thus they will maintain their own local simulation times. The main issue with parallel simulations is keeping LPs synchronized in order to avoid violating the causality of events. Without synchronization, a message sent by one LP could arrive in another LP when the simulation time in the receiving LP has already passed the timestamp (arrival time) of the message. This would break causality of events in the receiving LP.

There are two broad categories of parallel simulation algorithms that differ in the way they handle causality problems outlined above:

1. **Conservative algorithms** prevents incausalities from happening. The Null Message Algorithm exploits knowledge of the time when LPs send messages to other LPs, and uses 'null' messages to propagate this information to other LPs. If an LP knows it won't receive any messages from other LPs until $t + \Delta t$ simulation time, it may advance until $t + \Delta t$ without the need for external synchronization. Conservative simulation tends to converge to sequential simulation (slowed down by communication between LPs) if there's not enough parallelism in the model, or parallelism is not exploited by sending a sufficient number of '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.

12.2 Assessing available parallelism in a simulation model

OMNeT++ currently supports conservative synchronization via the classic Chandy-Misra-Bryant (or null message) algorithm [CM79]. To assess how efficiently a simulation can be parallelized with this algorithm, we'll need the following variables:

- P *performance* represents the number of events processed per second (ev/sec). [1] P depends on the performance of the hardware and the computation-intensiveness of processing an event. P is independent of the size of the model. Depending on the nature of the simulation model and the performance of the computer, P is usually in the range of 20,000..500,000 ev/sec.
- E *event density* is the number of events that occur per simulated second (ev/simsec). E depends on the model only, and not where the model is executed. E is determined by the size, the detail level and also the nature of the simulated system (e.g. cell-level ATM models produce higher E values than call center simulations.)
- R *relative speed* measures the simulation time advancement per second (simsec/sec). R strongly depends on both the model and on the software/hardware environment where the model executes. Note that $R = P/E$.
- L *lookahead* is measured in simulated seconds (simsec). When simulating telecommunication networks and using link delays as lookahead, L is typically in the msimsec-μsimsec range.
- τ *latency* (sec) characterizes the parallel simulation hardware. τ is the latency of sending a message from one LP to another. τ can be determined using simple benchmark programs. The authors' measurements on a Linux cluster interconnected via a 100Mb Ethernet switch using MPI yielded τ=22μs which is consistent with measurements reported in [OF00]. Specialized hardware such as Quadrics Interconnect [Qua] can provide τ=5μs or better.

In large simulation models, P, E and R usually stay relatively constant (that is, display little fluctuations in time). They are also intuitive and easy to measure. The OMNeT++ displays these values on the GUI while the simulation is running, see Figure 12.1. Cmdenv can also be configured to display these values.

Figure 12.1: Performance bar in OMNeT++ showing P, R and E

After having approximate values of P, E, L and τ, calculate the λ *coupling factor* as the ratio of LE and τP:

$\lambda = (LE)/(\tau P)$

Without going into the details: if the resulting λ value is at minimum larger than one, but rather in the range 10..100, there is a good change that the simulation will perform well when run in parallel. With $\lambda < 1$, poor performance is guaranteed. For details see the paper [VŞE03].

12.3 Parallel distributed simulation support in OMNeT++

12.3.1 Overview

This chapter presents the parallel simulation architecture of OMNeT++. The design allows simulation models to be run in parallel without code modification – it only requires configuration. The implementation relies on the approach of placeholder modules and proxy gates to instantiate the model on different LPs – the placeholder approach allows simulation techniques such as topology discovery and direct message sending to work unmodified with PDES. The architecture is modular and extensible, so it can serve as a framework for research on parallel simulation.

The OMNeT++ design places a big emphasis on *separation of models from experiments*. The main rationale is that usually a large number of simulation experiments need to be done on a single model before a

[1]Notations: *ev:* events, *sec:* real seconds, *simsec:* simulated seconds

conclusion can be drawn about the real system. Experiments tend to be ad-hoc and change much faster than simulation models, thus it is a natural requirement to be able to carry out experiments without disturbing the simulation model itself.

Following the above principle, OMNeT++ allows simulation models to be executed in parallel without modification. No special instrumentation of the source code or the topology description is needed, as partitioning and other PDES configuration is entirely described in the configuration files.

OMNeT++ supports the Null Message Algorithm with static topologies, using link delays as lookahead. The laziness of null message sending can be tuned. Also supported is the Ideal Simulation Protocol (ISP) introduced by Bagrodia in 2000 [BT00]. ISP is a powerful research vehicle to measure the efficiency of PDES algorithms, both optimistic and conservative; more precisely, it helps determine the maximum speedup achievable by any PDES algorithm for a particular model and simulation environment. In OMNeT++, ISP can be used for benchmarking the performance of the Null Message Algorithm. Additionally, models can be executed without any synchronization, which can be useful for educational purposes (to demonstrate the need for synchronization) or for simple testing.

For the communication between LPs (logical processes), OMNeT++ primarily uses MPI, the Message Passing Interface standard [For94]. An alternative communication mechanism is based on named pipes, for use on shared memory multiprocessors without the need to install MPI. Additionally, a file system based communication mechanism is also available. It communicates via text files created in a shared directory, and can be useful for educational purposes (to analyse or demonstate messaging in PDES algorithms) or to debug PDES algorithms. Implementation of a shared memory-based communication mechanism is also planned for the future, to fully exploit the power of multiprocessors without the overhead of and the need to install MPI.

Nearly every model can be run in parallel. The constraints are the following:

- modules may communicate via sending messages only (no direct method call or member access) unless mapped to the same processor
- no global variables
- there are some limitations on direct sending (no sending to a *sub*module of another module, unless mapped to the same processor)
- lookahead must be present in the form of link delays
- currently static topologies are supported (we are working on a research project that aims to eliminate this limitation)

PDES support in OMNeT++ follows a modular and extensible architecture. New communication mechanisms can be added by implementing a compact API (expressed as a C++ class) and registering the implementation – after that, the new communications mechanism can be selected for use in the configuration.

New PDES synchronization algorithms can be added in a similar way. PDES algorithms are also represented by C++ classes that have to implement a very small API to integrate with the simulation kernel. Setting up the model on various LPs as well as relaying model messages across LPs is already taken care of and not something the implementation of the synchronization algorithm needs to worry about (although it can intervene if needed, because the necessary hooks are provided).

The implementation of the Null Message Algorithm is also modular in itself in that the lookahead discovery can be plugged in via a defined API. Currently implemented lookahead discovery uses link delays, but it is possible to implement more sophisticated ones and select them in the configuration.

12.3.2 Parallel Simulation Example

We will use the Parallel CQN example simulation for demonstrating the PDES capabilities of OMNeT++. The model consists of N tandem queues where each tandem consists of a switch and k single-server queues with exponential service times (Figure 12.2). The last queues are looped back to their switches. Each switch randomly chooses the first queue of one of the tandems as destination, using uniform distribution. The queues and switches are connected with links that have nonzero propagation delays. Our OMNeT++ model for CQN wraps tandems into compound modules.

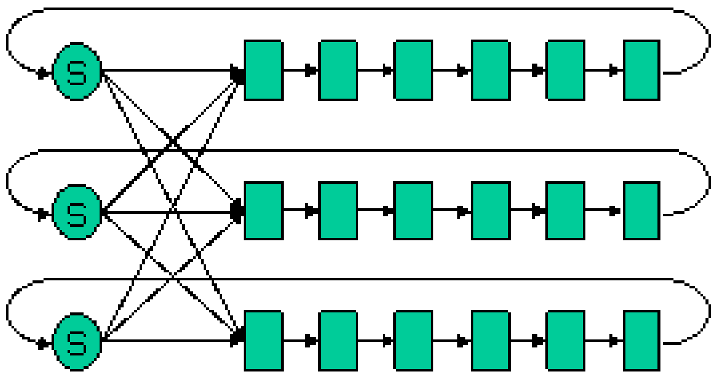


Figure 12.2: The Closed Queueing Network (CQN) model

To run the model in parallel, we assign tandems to different LPs (Figure 12.3). Lookahead is provided by delays on the marked links.

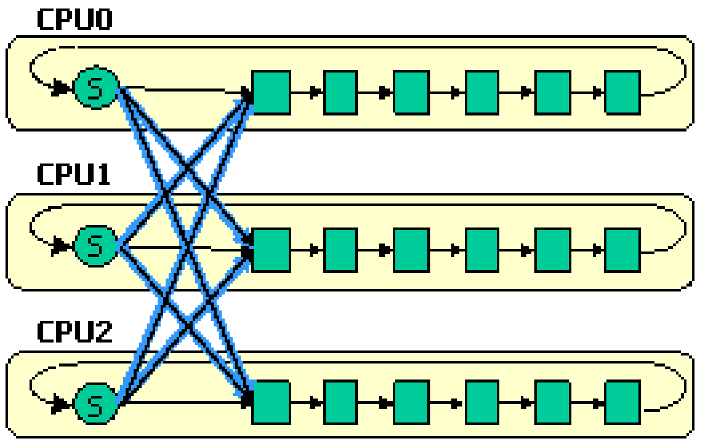


Figure 12.3: Partitioning the CQN model

To run the CQN model in parallel, we have to configure it for parallel execution. In OMNeT++, the configuration is in a text file called `omnetpp.ini`. For configuration, first we have to specify partitioning, that is, assign modules to processors. This is done by the following lines:

```
[Partitioning]
*.tandemQueue[0]**.partition-id = 0
*.tandemQueue[1]**.partition-id = 1
*.tandemQueue[2]**.partition-id = 2
```

The numbers after the equal sign identify the LP.

Then we have to select the communication library and the parallel simulation algorithm, and enable parallel simulation:

```
[General]
```

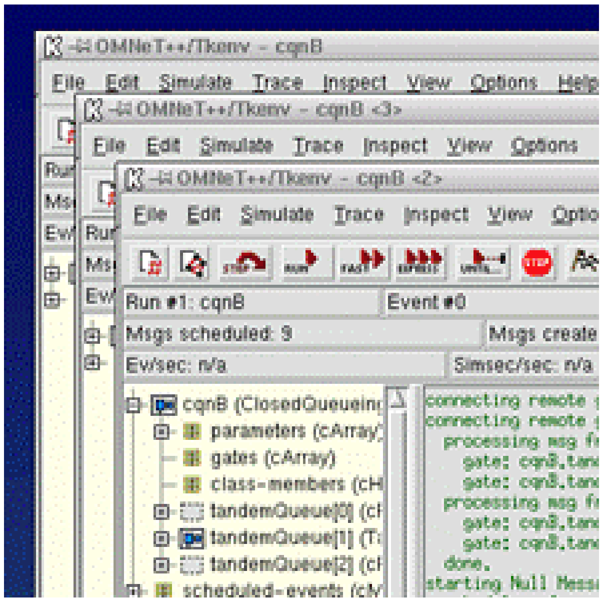
```
parallel-simulation=true
parsim-communications-class = "cMPICommunications"
parsim-synchronization-class = "cNullMessageProtocol"
```

When the parallel simulation is run, LPs are represented by multiple running instances of the same program. When using LAM-MPI [LAM], the mpirun program (part of LAM-MPI) is used to launch the program on the desired processors. When named pipes or file communications is selected, the opp_prun OMNeT++ utility can be used to start the processes. Alternatively, one can run the processes by hand (the -p flag tells OMNeT++ the index of the given LP and the total number of LPs):

```
./cqn -p0,3 &
./cqn -p1,3 &
./cqn -p2,3 &
```

For PDES, one will usually want to select the command-line user interface, and redirect the output to files. (OMNeT++ provides the necessary configuration options.)

The graphical user interface of OMNeT++ can also be used (as evidenced by Figure 12.4), independent of the selected communication mechanism. The GUI interface can be useful for educational or demonstation purposes. OMNeT++ displays debugging output about the Null Message Algorithm, EITs and EOTs can be inspected, etc.


OMNeT++/Tkenv - cqnB
File Edit Simulate Trace Inspect View Options Help
OMNeT++/Tkenv - cqnB <3>
File Edit Simulate Trace Inspect View Options
OMNeT++/Tkenv - cqnB <2>
File Edit Simulate Trace Inspect View Optio
STEP RUN FAST EXPRESS UNTIL STOP

```
Run #1: cqnB            Event #0
Msgs scheduled: 9       Msgs create
Ev/sec: n/a             Simsec/sec: n/a

cqnB (ClosedQueueing
  parameters (cArray)
  gates (cArray)
  class-members (cH
  tandemQueue[0] (cl
  tandemQueue[1] (T
  tandemQueue[2] (cl
scheduled-events (cM

connecting remote
connecting remote
  processing msg fr
    gate: cqnB.tan
    gate: cqnB.tan
  processing msg fr
    gate: cqnB.tan
    gate: cqnB.tan
  done.
starting Null Mess
```

Figure 12.4: Screenshot of CQN running in three LPs

12.3.3 Placeholder modules, proxy gates

When setting up a model partitioned to several LPs, OMNeT++ uses placeholder modules and proxy gates. In the local LP, placeholders represent sibling submodules that are instantiated on other LPs. With placeholder modules, every module has all of its siblings present in the local LP – either as placeholder or as the "real thing". Proxy gates take care of forwarding messages to the LP where the module is instantiated (see Figure 12.5).

The main advantage of using placeholders is that algorithms such as topology discovery embedded in the model can be used with PDES unmodified. Also, modules can use direct message sending to any sibling module, including placeholders. This is so because the destination of direct message sending is an input gate of the destination module – if the destination module is a placeholder, the input gate will be a proxy gate which transparently forwards the messages to the LP where the "real" module was instantiated. A limitation is that the destination of direct message sending cannot be a *submodule* of a sibling (which is probably a bad practice anyway, as it violates encapsulation), simply because placeholders are empty and so its submodules are not present in the local LP.

Instantiation of compound modules is slightly more complicated. Since submodules can be on different LPs, the compound module may not be "fully present" on any given LP, and it may have to be present on several LPs (wherever it has submodules instantiated). Thus, compound modules are instantiated wherever they have at least one submodule instantiated, and are represented by placeholders everywhere else (Figure 12.6).

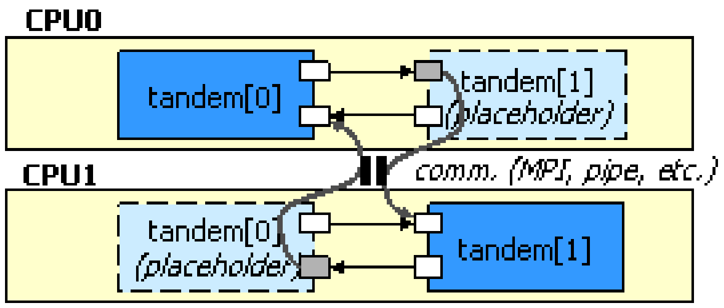


Figure 12.5: Placeholder modules and proxy gates

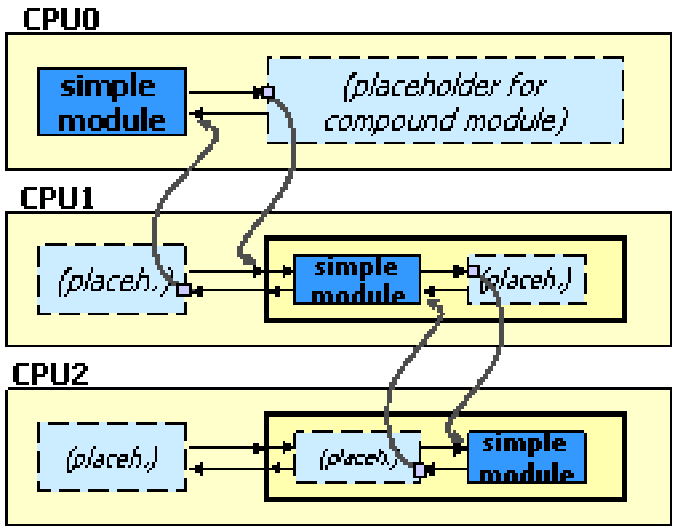


Figure 12.6: Instantiating compound modules

12.3.4 Configuration

Parallel simulation configuration is the `[General]` section of `omnetpp.ini`.

Entry and default value	**Description**
[General]	

parallel-simulation = <true/false> default: false	Enables parallel distributed simulation. The following configuration entries are only examined if `parallel-simulation=true`
parsim-debug = <true/false> default: true	Enables debugging output
parsim-mpicommunications-mpibuffer = <bytes> default: 256K * (numPartitions-1) + 16K	Size of MPI send buffer to allocate; see MPI_Buffer_attach() MPI call. If the buffer is too small, a deadlock can occur.
parsim-namedpipecommunications-prefix = <string> default: "omnetpp" or "comm/"	Controls the naming of named pipes. Windows: default value is "omnetpp", which means that pipe names will be of the form "\\.\pipe\omnetpp-xx-yy" (where xx and yy are numbers). Unix: default value is "comm/", which means that the named pipes will be created with the name "comm/pipe-xx-yy". The "comm/" subdirectory must already exist when the simulation is launched.
parsim-filecommunications-prefix = <string> default: "comm/"	(see below)
parsim-filecommunications-preserve-read = <true/false> default: false	(see below)
parsim-filecommunications-read-prefix = <string> default: "comm/read/"	The above 3 options control the `cFileCommunications` class. By default, it deletes files that were read. By enabling the "preserve-read" setting, you can make it move read files to another directory instead ("comm/read/" by default). BEWARE: for mysterious reasons, it appears that there cannot be more than about 19800 files in a directory. When that point is reached, an exception is thrown somewhere inside the standard C library, which materializes itself in OMNeT++ as an "Error: (null)" message... Strangely, this can be reproduced in both Linux and Windows.
parsim-nullmessageprotocol-lookahead-class = <class name string> default: "cLinkDelayLookahead"	Selects the lookahead class for the Null Message Algorithm; the class must be subclassed from cNMPLookahead.
parsim-nullmessageprotocol-laziness = <0..1> default: 0.5	Controls how often the Null Message Algorithm should send out null messages; the value is understood in proportion to the lookahead, e.g. 0.5 means every lookahead/2 simsec.
parsim-idealsimulationprotocol-tablesize = <int> default: 100,000	Size of chunks (in table entries) in which the external events file (recorded by cISPEventLogger) should be loaded. (one entry is 8 bytes, so 100,000 corresponds to 800K allocated memory)

When you are using cross-mounted home directories (the simulation's directory is on a disk mounted on all nodes of the cluster), a useful configuration setting is

```
[General]
fname-append-host=yes
```

It will cause the host names to be appended to the names of all output vector files, so that partitions do not overwrite each other's output files. (See section 8.10.3)

12.3.5 Design of PDES Support in OMNeT++

Design of PDES support in OMNeT++ follows a layered approach, with a modular and extensible architecture. The overall architecture is depicted in Figure 12.7.

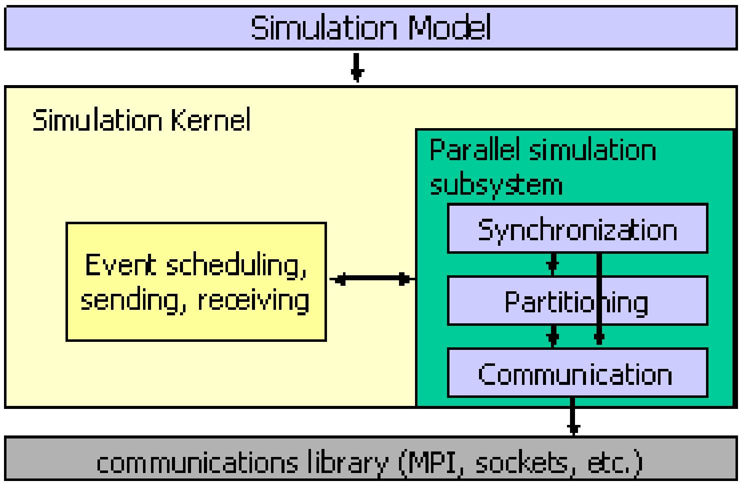

Simulation Model
Simulation Kernel
Parallel simulation subsystem
Synchronization
Event scheduling, sending, receiving
Partitioning
Communication
communications library (MPI, sockets, etc.)

Figure 12.7: Architecture of OMNeT++ PDES implementation

The parallel simulation subsytem is an optional component itself, which can be removed from the simulation kernel if not needed. It consists of three layers, from the bottom up: communication layer, partitioning layer and synchronization layer.

The purpose of the *Communication layer* is to provide elementary messaging services between partitions for the upper layer. The services include send, blocking receive, nonblocking receive and broadcast. The send/receive operations work with *buffers*, which encapsulate packing and unpacking operations for primitive C++ types. The message class and other classes in the simulation library can pack and unpack themselves into such buffers. The Communications layer API is defined in the `cFileCommunications` interface (abstract class); specific implementations like the MPI one (`cMPICommunications`) subclass from this, and encapsulate MPI send/receive calls. The matching buffer class `cMPICommBuffer` encapsulates MPI pack/unpack operations.

The *Partitioning layer* is responsible for instantiating modules on different LPs according to the partitioning specified in the configuration, for configuring proxy gates. During the simulation, this layer also ensures that cross-partition simulation messages reach their destinations. It intercepts messages that arrive at proxy gates and transmits them to the destination LP using the services of the communication layer. The receiving LP unpacks the message and injects it at the gate the proxy gate points at. The implementation basically encapsulates the `cParsimSegment`, `cPlaceHolderModule`, `cProxyGate` classes.

The *Synchronization layer* encapsulates the parallel simulation algorithm. Parallel simulation algorithms are also represented by classes, subclassed from the `cParsimSynchronizer` abstract class. The parallel simulation algorithm is invoked on the following hooks: event scheduling, processing model messages outgoing from the LP, and messages (model messages or internal messages) arriving from other LPs. The first hook, event scheduling is a function invoked by the simulation kernel to determine the next simulation event; it also has full access to the future event set (FES) and can add/remove events for its own use. Conservative parallel simulation algorithms will use this hook to block the simulation if the next event is unsafe, e.g. the null message algorithm implementation (`cNullMessageProtocol`) blocks the simulation if an EIT has been reached until a null message arrives (see [BT00] for terminology); also it uses this hook to periodically send null messages. The second hook is invoked when a model message is sent to another LP; the null message algorithm uses this hook to piggyback null messages on outgoing model messages. The third hook is invoked when any message arrives from other LPs, and it allows the parallel simulation algorithm to process its own internal messages from other partitions; the null message algorithm processes incoming null messages here.

The null message protocol implementation itself is modular, it employs a separate, configurable lookahead discovery object. Currently only link delay based lookahead discovery has been implemented, but it is possible to implement more sophisticated ones.

The Ideal Simulation Protocol (ISP; see [BT00]) implementation consists in fact of two parallel simulation protocol implementations: the first one is based on the null message algorithm and additionally records the external events (events received from other LPs) to a trace file; the second one executes the simulation using the trace file to find out which events are safe and which are not.

Note that although we implemented a conservative protocol, the provided API itself would allow implementing optimistic protocols, too. The parallel simulation algorithm has access to the executing simulation model, so it could perform saving/restoring model state if model objects support this [2].

We also expect that because of the modularity, extensibility and clean internal architecture of the parallel simulation subsystem, the OMNeT++ framework has the potential to become a preferred platform for PDES research.

[2] Unfortunately, support for state saving/restoration needs to be individually and manually added to each class in the simulation, including user-programmed simple modules.

Chapter 13

Customization and Embedding

13.1 Architecture

OMNeT++ has a modular architecture. The following diagram shows the high-level architecture of OMNeT++ simulations:

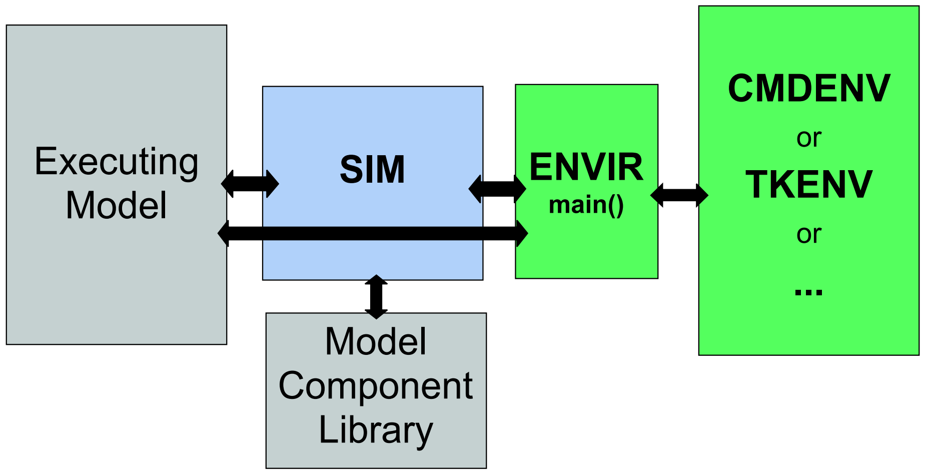


Figure 13.1: Architecture of OMNeT++ simulation programs

The rectangles in the picture represent components:

- **Sim** is the simulation kernel and class library. Sim exists as a library you link your simulation program with. [1]

- **Envir** is another library which contains all code that is common to all user interfaces. `main()` is also in Envir. Envir provides services like ini file handling for specific user interface implementations. Envir presents itself towards Sim and the executing model via the `ev` facade object, hiding all other user interface internals. Some aspects of Envir can be customized via plugin interfaces. Embedding OMNeT++ into applications can be achieved implementing a new user interface in addition to Cmdenv and Tkev, or by replacing Envir with another implementation of `ev` (see sections 13.5.3 and 13.2.)

[1]Use of dynamic (shared) libraries is also possible, but for simplicity we'll use the word *linking* here.

- **Cmdenv and Tkenv** are specific user interface implementations. A simulation is linked with either Cmdenv or Tkenv.
- The **Model Component Library** consists of simple module definitions and their C++ implementations, compound module types, channels, networks, message types and in general everything that belongs to models and has been linked into the simulation program. A simulation program is able to run any model that has all necessary components linked in.
- The **Executing Model** is the model that has been set up for simulation. It contains objects (modules, channels, etc.) that are all instances of components in the model component library.

The arrows in the figure show how components interact with each other:

- **Executing Model vs Sim**. The simulation kernel manages the future events and invokes modules in the executing model as events occur. The modules of the executing model are stored in the main object of Sim, `simulation` (of class `cSimulation`). In turn, the executing model calls functions in the simulation kernel and uses classes in the Sim library.
- **Sim vs Model Component Library**. The simulation kernel instantiates simple modules and other components when the simulation model is set up at the beginning of the simulation run. It also refers to the component library when dynamic module creation is used. The machinery for registering and looking up components in the model component library is implemented as part of Sim.
- **Executing Model vs Envir**. The `ev` object, logically part of Envir, is the facade of the user interface towards the executing model. The model uses `ev` to write debug logs (`ev«`, `ev.printf()`).
- **Sim vs Envir**. Envir is in full command of what happens in the simulation program. Envir contains the `main()` function where execution begins. Envir determines which models should be set up for simulation, and instructs Sim to do so. Envir contains the main simulation loop (*determine-next-event*, *execute-event* sequence) and invokes the simulation kernel for the necessary functionality (event scheduling and event execution are implemented in Sim). Envir catches and handles errors and exceptions that occur in the simulation kernel or in the library classes during execution. Envir presents a single facade object (`ev`) that represents the environment (user interface) toward Sim – no Envir internals are visible to Sim or the executing model. During simulation model setup, Envir supplies parameter values for Sim when Sim asks for them. Sim writes output vectors via Envir, so one can redefine the output vector storing mechanism by changing Envir. Sim and its classes use Envir to print debug information.
- **Envir vs Tkenv and Cmdenv**. Envir defines `TOmnetApp` as a base class for user interfaces, and Tkenv and Cmdenv both subclass from `TOmnetApp`. The `main()` function provided as part of Envir determines the appropriate user interface class (subclassed from `TOmnetApp`), creates an instance and runs it – whatever happens next (opening a GUI window or running as a command-line program) is decided in the `run()` method of the appropriate `TOmnetApp` subclass. Sim's or the model's calls on the `ev` object are simply forwarded to the `TOmnetApp` instance. Envir presents a framework and base functionality to Tkenv and Cmdenv via the methods of `TOmnetApp` and some other classes.)

13.2 Embedding OMNeT++

This section discusses the issues of embedding the simulation kernel or a simulation model into a larger application.

What you'll absolutely need for a simulation to run is the Sim library. 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. You can keep Envir if its philosophy and the infrastructure it provides (`omnetpp.ini`, certain command-line options etc.) fit into your design. Then your application, 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 to replace it. Your Envir replacement (the embedding application, 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 similar to what is currently generated by nedtool and then it can build any network whose components (primarily the simple modules) are linked in. Moreover, it is 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.

13.3 Sim: the simulation kernel and class library

There is little to say about Sim here, since chapters 4 and 6, and part of chapter 5 are all about this topic. Classes covered in those chapters are documented in more detail in the API Reference generated by Doxygen. What we can do here is elaborating on some internals that have not been covered in the general chapters.

The source code for the simulation kernel and class library reside in the `src/sim/` subdirectory.

13.3.1 The global simulation object

The global `simulation` object is an instance of `cSimulation`. It stores the model, and encapsulates much of the functionality of setting up and running a simulation model.

`simulation` has two basic roles:

- it stores modules of the executing model
- it holds the future event set (FES) object

13.3.2 The coroutine package

The coroutine package is in fact made up of two coroutine packages:

- A portable coroutine package creates all coroutine stacks inside the main stack. It is based on Kofoed's solution[Kof95]. It allocates stack by deep-deep recursions and then plays with `setjmp()` and `longjmp()` to switch from one to another.
- On Windows, the Fiber functions (`CreateFiber()`, `SwitchToFiber()`, etc) are used, which are part of the standard Win32 API.

The coroutines are represented by the `cCoroutine` class. `cSimpleModule` has `cCoroutine` as one a base class.

13.4 The Model Component Library

All model components (simple module definitions and their C++ implementations, compound module types, channels, networks, message types, etc.) that you compile and link into a simulation program are registered in the Model Component Library. Any model that has all its necessary components in the component library of the simulation program can be run by that simulation program.

If your simulation program is linked with Cmdenv or Tkenv, you can have the contents of its component library printed, using the -h switch.

```
% ./fddi -h

OMNeT++ Discrete Event Simulation  (C) 1992-2004 Andras Varga
...
Available networks:
  FDDI1
  NRing
  TUBw
  TUBs

Available modules:
  FDDI_MAC
  FDDI_MAC4Ring
  ...

Available channels:
  ...
End run of OMNeT++
```

Information on components are kept on registration lists. There are macros for registering components (that is, for adding them to the registeration lists): `Define_Module()`, `Define_Module_Like()`, `Define_Network()`, `Define_Function()`, `Register_Class()`, and a few others. For components defined in NED files, the macro calls are generated by the NED compiler; in other cases you have to write them in your C++ source.

Let us see the module registrations as an example. The

```
Define_Module(FIFO);
```

macro expands to the following code:

```
static cModule *FIFO__create(const char *name, cModule *parentmod)
{
    return new FIFO(name, parentmod);
}

EXECUTE_ON_STARTUP( FIFO__mod,
    modtypes.instance()->add(
       new cModuleType("FIFO","FIFO",(ModuleCreateFunc)FIFO__create)
    );
)
```

When the simulation program starts up, a new `cModuleType` object will be added to the `modtypes` object, which holds the list of available module types. The `cModuleType` object will act as a factory: when its create() method is called it will produce a new module object of class `FIFO` via the above static function `FIFO__create`.

The `cModuleType` object also stores the name of the corresponding NED module declaration. This makes it possible to add the gates and parameters declared in NED to the module when it is created.

The machinery for managing the registration lists are part of the Sim library. Registration lists are implemented as global objects.

The registration lists are:

List variable	**Macro/ Objects on list**	**Function**

`networks`	`Define_Network()`
	
	`cNetworkType`
		List of available networks. Every `cNetworkType` object is a factory for a specific network type. That is, a `cNetworkType` object has methods for setting up a specific network. `Define_Network()` macros occur in the code generated by the NED compiler.

`modtypes`	`Define_Module()`, `Define_Module_Like()`, `cModuleType`	List of available module types. Every `cModuleType` object is a factory for a specific module type. Usually, `Define_Module()` macros for compound modules occur in the code generated by the NED compiler; for simple modules, the `Define_Module()` lines are added by the user.

`channeltypes`	`Define_Channel()`
	
	`cChannelType`
		List of channel types. Every `cChannelType` object acts as a factory for a channel type, a class derived from `cChannel`.

`classes`	`Register_Class()`
	
	`cClassRegister`
		List of available classes of which one can create an instance. Every `cClassRegister` object is a factory for objects of a specific class. The list is used by the `createOne()` function: it can create an object of any class, given the class name as a string. (E.g. the statement `ptr = createOne("cArray")` creates a `cArray` object.) To enable a class to work with `createOne()`, one has to register it using the `Register_Class(classname)` macro

`functions`	`Define_Function()`
	
	`cFunctionType`
		List of functions taking `double`s and returning a `double` (see type `MathFuncNoArg`...`MathFunc3Args`). A `cFunctionType` object holds a pointer to the function and knows how many arguments it takes.

13.5 Envir, Tkenv and Cmdenv

The source code for the user interface of OMNeT++ resides in the `src/envir/` directory (common part) and in the `src/cmdenv/`, `src/tkenv/` directories.

The classes in the user interface are *not* derived from `cObject`, they are completely separated from the simulation kernel.

13.5.1 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).

13.5.2 The cEnvir interface

The `cEnvir` class has only one instance, a global object called `ev`:

```
cEnvir ev;
```

`cEnvir` basically a facade, its member functions contain little 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 perform the following groups of tasks:

- I/O for module activities; the actual implementation is different for each user interface (e.g. stdin/stdout for Cmdenv, windowing in Tkenv)
- cEnvir provides methods for the simulation kernel to access configuration information (for example, module parameter settings)
- cEnvir also provides methods that are called by simulation kernel to notify the user interface of certain events (an object was deleted; a module was created or deleted; a message was sent or delivered, etc.)

13.5.3 Customizing Envir

Certain aspects of Envir can be customized via plugin interfaces. The following plugin interfaces are supported:

- `cRNG`. Interface for the random number generator.
- `cScheduler`. The scheduler class. This plugin interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation.
- `cConfiguration`. It defines a class from which all configuration will be obtained. In other words, it option lets you replace `omnetpp.ini` with some other implementation, e.g. database input.
- `cOutputScalarManager`. It handles recording the scalar output data, output via the cModule::recordScalar(family of functions. The default output scalar manager is `cFileOutputScalarManager`, defined in the Envir library.
- `cOutputVectorManager`. It handles recording the output for `cOutVector` objects. The default output vector manager is `cFileOutputVectorManager`, defined in the Envir library.
- `cSnapshotManager`. It provides an output stream to which snapshots are written (see section 6.10.5). The default snapshot manager is `cFileSnapshotManager`, defined in the Envir library.

The classes (`cRNG`, `cScheduler`, etc.) are documented in the API Reference.

To actually implement and select a plugin for use:

1. Subclass the given interface class (e.g. for a custom RNG, `cRNG`) to create your own version.
2. Register the class by putting the `Register_Class(MyRNGClass)` line into the C++ source.
3. Compile and link your interface class into the OMNeT++ simulation executable. IMPORTANT: make sure the executable actually contains the code of your class! Over-optimizing linkers (esp. on Unix) tend to leave out code to which there seem to be no external reference.

4. Add an entry to `omnetpp.ini` to tell Envir use your class instead of the default one. For RNGs, this setting is `rng-class` in the `[General]` section.

Ini file entries that allow you to select your plugin classes are `configuration-class`, `scheduler-class`, `rng-class`, `outputvectormanager-class`, `outputscalarmanager-class` and `snapshotmanager-class`, documented in section 8.2.6.

How plugin classes can access the configuration

The configuration is available to plugin classes via the `config()` method of `cEnvir`, which returns a pointer to the configuration object (`cConfiguration`). This enables plugin classes to have their own config entries.

An example which reads the `parsim-debug` boolean entry from the `[General]` section, with `true` as default:

```
bool debug = ev.config()->getAsBool("General", "parsim-debug", true);
```

Startup sequence for the configuration plugin

For the configuration plugin, the startup sequence is the following (see `cEnvir::setup()` in the source code):

1. First, `omnetpp.ini` (or the ini file(s) specified via the "-f" command-line option) are read.
2. Shared libraries in `[General]/load-libs` are loaded. (Also the ones specified with the "-l" command-line option.)
3. `[General]/configuration-class` is examined, and if it is present, a configuration object of the given class is instantiated. The configuration object may read further entries from the ini file (e.g. database connect parameters, or XML file name).
4. The original `omnetpp.ini cInifile` configuration object is deleted. No other settings are taken from it.
5. `[General]/load-libs` from the new configuration object is processed.
6. Then everything goes on as normally, using the new configuration object.

13.5.4 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_`)

Simulation applications:

- add new configuration options
- provide a `run()` function
- implement functions left empty in `TOmnetApp` (like `breakpointHit()`, `objectDeleted()`).

Appendix A

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
     [ paramblock ]
     [ gateblock ]
    endsimple [ simplemoduletype ]

moduledefinition ::=
    module compoundmoduletype
     [ paramblock ]
     [ gateblock ]
     [ submodblock ]
     [ connblock ]
    endmodule [ compoundmoduletype ]

moduletype ::=
    simplemoduletype | compoundmoduletype

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 ]
     [ substparamblock... ]
     [ gatesizeblock... ] }
  | { submodulename : parametername [ vector ] like moduletype
     [ substparamblock... ]
     [ gatesizeblock... ] }

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
     [ 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
```

References

[BT00] R. L. Bagrodia and M. Takai. Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages. 11(4):395–414, 2000.

[CM79] M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, (5):440–452, 1979.

[EHW02] K. Entacher, B. Hechenleitner, and S. Wegenkittl. A Simple OMNeT++ Queuing Experiment Using Parallel Streams. *PARALLEL NUMERICS'02 - Theory and Applications*, pages 89–105, 2002. Editors: R. Trobec, P. Zinterhof, M. Vajtersic and A. Uhl.

[EPM99] G. Ewing, K. Pawlikowski, and D. McNickle. Akaroa2: Exploiting Network Computing by Distributing Stochastic Simulation. In *Proceedings of the European Simulation Multiconference ESM'99, Warsaw, June 1999*, pages 175–181. International Society for Computer Simulation, 1999.

[For94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. 8(3/4):165–414, 1994.

[Gol91] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[Hel98] P. Hellekalek. Don't Trust Parallel Monte Carlo. *ACM SIGSIM Simulation Digest*, 28(1):82–89, jul 1998. Author's page is a great source of information, see `http://random.mat.sbg.ac.at/`.

[HPvdL95] Jan Heijmans, Alex Paalvast, and Robert van der Leij. Network Simulation Using the JAR Compiler for the OMNeT++ Simulation System. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, 1991.

[JC85] Raj Jain and Imrich Chlamtac. The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms without Storing Observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[Kof95] Stig Kofoed. Portable Multitasking in C++. *Dr. Dobb's Journal*, November 1995. Download source from `http://www.ddj.com/ftp/1995/1995.11/mtask.zip/`.

[LAM] LAM-MPI home page. `http://www.lam-mpi.org/`.

[Len94] Gábor Lencse. Graphical Network Editor for OMNeT++. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[LSCK02] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An Objected-Oriented Random-Number Package with Many Long Streams and Substreams. *Operations Research*, 50(6):1073–1075, 2002. Source code can be downloaded from `http://www.iro.umontreal.ca/~lecuyer/papers.html`.

[MN98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998. Source code can be downloaded from `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[MvMvdW95] André Maurits, George van Montfort, and Gerard van de Weerd. OMNeT++ Extensions and Examples. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[OF00] Hong Ong and Paul A. Farrell. Performance Comparison of LAM/MPI, MPICH and MVICH on a Linux Cluster Connected by a Gigabit Ethernet Network. In *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, October 10-14, 2000*. The USENIX Association, 2000.

[PFS86] Bratley P., B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, 1986.

[PJL02] K. Pawlikowski, H. Jeong, and J. Lee. On Credibility of Simulation Studies of Telecommunication Networks. *IEEE Communications Magazine*, pages 132–139, jan 2002.

[Pon91] György Pongor. OMNET: An Object-Oriented Network Simulator. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1991.

[Pon92] György Pongor. Statistical Synchronization: A Different Approach of Parallel Discrete Event Simulation. Technical report, University of Technology, Data Communications Laboratory, Lappeenranta, Finland, 1992.

[Pon93] György Pongor. On the Efficiency of the Statistical Synchronization Method. In *Proceedings of the European Simulation Symposium (ESS'93), Delft, The Netherlands, Oct. 25-28, 1993*. International Society for Computer Simulation, 1993.

[Qua] Quadrics home page. `http://www.quadrics.com/`.

[ŞVE03] Y. Ahmet Şekercioğlu, András Varga, and Gregory K. Egan. Parallel Simulation Made Easy with OMNeT++. In *Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands*. International Society for Computer Simulation, 2003.

[Var92] András Varga. OMNeT++ - Portable Simulation Environment in C++. In *Proceedings of the Annual Students' Scientific Conference (TDK), 1992*. Technical University of Budapest, 1992. In Hungarian.

[Var94] András Varga. Portable User Interface for the OMNeT++ Simulation System. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[Var98a] András Varga. K-split – On-Line Density Estimation for Simulation Result Collection. In *Proceedings of the European Simulation Symposium (ESS'98), Nottingham, UK, October 26-28*. International Society for Computer Simulation, 1998.

[Var98b] András Varga. Parameterized Topologies for Simulation Programs. In *Proceedings of the Western Multiconference on Simulation (WMC'98) Communication Networks and Distributed Systems (CNDS'98), San Diego, CA, January 11-14*. International Society for Computer Simulation, 1998.

[Var99] András Varga. Using the OMNeT++ Discrete Event Simulation System in Education. *IEEE Transactions on Education*, 42(4):372, November 1999. (on CD-ROM issue; journal contains abstract).

[Vas96] Zoltán Vass. PVM Extension of OMNeT++ to Support Statistical Synchronization. Master's thesis, Technical University of Budapest, 1996. In Hungarian.

[VF97] András Varga and Babak Fakhamzadeh. The K-Split Algorithm for the PDF Approximation of Multi-Dimensional Empirical Distributions without Storing Observations. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 94–98. International Society for Computer Simulation, 1997.

[VP97] András Varga and György Pongor. Flexible Topology Description Language for Simulation Programs. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 225–229, 1997.

[VŞE03] András Varga, Y. Ahmet Şekercioğlu, and Gregory K. Egan. A practical efficiency criterion for the null message algorithm. In *Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands*. International Society for Computer Simulation, 2003.

[Wel95] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, 1995.

Index