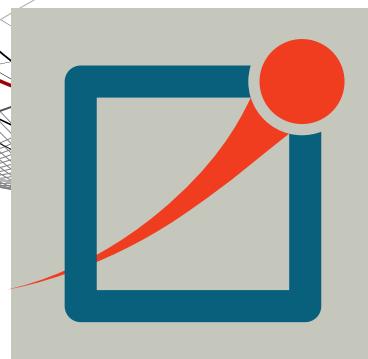


# **OMNeT++**

## User Guide

Version 5.4.1



Copyright © 2016 András Varga and OpenSim Ltd.

---

1. Introduction .....	1
1.1. The Workbench .....	1
1.2. Workspaces .....	2
1.3. The Simulation Perspective .....	3
1.4. Configuring OMNeT++ Preferences .....	3
1.5. Creating OMNeT++ Projects .....	3
1.6. Project References .....	4
1.7. Getting Help .....	5
2. Editing NED Files .....	6
2.1. Overview .....	6
2.2. Opening Older NED Files .....	6
2.3. Creating New NED Files .....	6
2.3.1. NED Source Folders .....	7
2.4. Using the NED Editor .....	8
2.4.1. Editing in Graphical Mode .....	8
2.4.2. Editing in Source Mode .....	12
2.5. Associated Views .....	15
2.5.1. Outline View .....	15
2.5.2. Property View .....	15
2.5.3. Palette View .....	16
2.5.4. Problems View .....	16
2.5.5. NED Inheritance View .....	16
2.5.6. Module Hierarchy View .....	16
2.5.7. Parameters View .....	17
3. Editing INI Files .....	18
3.1. Overview .....	18
3.2. Creating INI Files .....	18
3.3. Using the INI File Editor .....	19
3.3.1. Editing in Form Mode .....	19
3.3.2. Editing in Text Mode .....	22
3.4. Associated Views .....	23
3.4.1. Outline View .....	23
3.4.2. Problems View .....	23
3.4.3. Parameters View .....	23
3.4.4. Module Hierarchy View .....	24
3.4.5. NED Inheritance View .....	24
4. Editing Message Files .....	25
4.1. Creating Message Files .....	25
4.2. The Message File Editor .....	25
5. C++ Development .....	27
5.1. Introduction .....	27
5.2. Prerequisites .....	27
5.3. Creating a C++ Project .....	27
5.4. Editing C++ Code .....	29
5.4.1. The C++ Editor .....	30
5.4.2. Include Browser View .....	31
5.4.3. Outline View .....	32
5.4.4. Type Hierarchy View .....	32
5.5. Building the Project .....	32
5.5.1. Basics .....	32
5.5.2. Console View .....	34
5.5.3. Problems View .....	34
5.6. Configuring the Project .....	35
5.6.1. Configuring the Build Process .....	35
5.6.2. Managing Build Configurations .....	35
5.6.3. Configuring the Project Build System .....	36
5.6.4. Configuring Makefile Generation for a Folder .....	37
5.6.5. Project References and Makefile Generation .....	40

5.7. Project Features .....	41
5.7.1. Motivation .....	41
5.7.2. What is a Project Feature .....	41
5.7.3. The Project Features Dialog .....	42
5.7.4. What Happens When You Enable/Disable a Feature .....	43
5.7.5. Using Features from Command Line .....	43
5.7.6. The <code>.oppfeatures</code> File .....	43
5.7.7. How to Introduce a Project Feature .....	44
5.8. Project Files .....	44
6. Launching and Debugging .....	46
6.1. Introduction .....	46
6.2. Launch Configurations .....	46
6.3. Running a Simulation .....	46
6.3.1. Quick Run .....	46
6.3.2. The Run Configurations Dialog .....	47
6.3.3. Creating a Launch Configuration .....	47
6.3.4. Debug vs. Release Launch .....	50
6.4. Batch Execution .....	50
6.5. Debugging a Simulation .....	51
6.5.1. Starting a Debug Session .....	51
6.5.2. Using the Debugger .....	51
6.5.3. Pretty Printers .....	52
6.6. Just-in-Time Debugging .....	53
6.7. Profiling a Simulation on Linux .....	53
6.8. Controlling the Execution and Progress Reporting .....	53
7. The Qtenv Graphical Runtime Environment .....	56
7.1. Features .....	56
7.2. Overview of the User Interface .....	57
7.3. Using Qtenv .....	58
7.3.1. Starting Qtenv .....	58
7.3.2. Setting Up and Running the Simulation .....	58
7.3.3. Inspecting Simulation Objects .....	61
7.4. Using Qtenv with a Debugger .....	62
7.5. Parts of the Qtenv UI .....	63
7.5.1. The Status Bars .....	63
7.5.2. The Timeline .....	64
7.5.3. The Object Navigator .....	64
7.5.4. The Object Inspector .....	64
7.5.5. The Network Display .....	66
7.5.6. The Log Viewer .....	68
7.6. Inspecting Objects .....	70
7.6.1. Object Inspectors .....	70
7.6.2. Browsing the Registered Components .....	70
7.6.3. Querying Objects .....	71
7.7. The Preferences Dialog .....	72
7.7.1. General .....	73
7.7.2. Logs .....	74
7.7.3. Configuring the Layouting Algorithm .....	75
7.7.4. Configuring Animation .....	76
7.7.5. Timeline and Animation Filtering .....	77
7.7.6. Configuring Fonts .....	78
7.7.7. The <code>.qtenvrc</code> File .....	78
7.8. Qtenv and C++ .....	79
7.8.1. Inspectors .....	79
7.8.2. During Simulation .....	80
7.9. Reference .....	80
7.9.1. Command-Line Options .....	80
7.9.2. Environment Variables .....	80

7.9.3. Configuration Options .....	81
8. The Tkenv Graphical Runtime Environment .....	82
8.1. Features .....	82
8.2. Overview of the User Interface .....	83
8.3. Using Tkenv .....	84
8.3.1. Starting Tkenv .....	84
8.3.2. Setting Up and Running the Simulation .....	84
8.3.3. Inspecting Simulation Objects .....	86
8.4. Using Tkenv with a Debugger .....	88
8.5. Parts of the Tkenv UI .....	88
8.5.1. The Status Bar .....	88
8.5.2. The Timeline .....	89
8.5.3. The Object Navigator .....	89
8.5.4. The Object Inspector .....	90
8.5.5. The Network Display .....	91
8.5.6. The Log Viewer .....	93
8.6. Inspecting Objects .....	94
8.6.1. Object Inspectors .....	94
8.6.2. Browsing the Registered Components .....	96
8.6.3. Querying Objects .....	96
8.7. The Preferences Dialog .....	98
8.7.1. General .....	98
8.7.2. Configuring the Layouting Algorithm .....	99
8.7.3. Configuring Animation .....	100
8.7.4. Timeline and Animation Filtering .....	101
8.7.5. Configuring Fonts .....	102
8.7.6. The .tkenvrc File .....	102
8.8. Tkenv and C++ .....	103
8.8.1. Inspectors .....	103
8.8.2. During Simulation .....	104
8.9. Reference .....	104
8.9.1. Command-Line Options .....	104
8.9.2. Environment Variables .....	104
8.9.3. Configuration Options .....	105
9. Sequence Charts .....	106
9.1. Introduction .....	106
9.2. Creating an Eventlog File .....	106
9.3. Sequence Chart .....	107
9.3.1. Legend .....	107
9.3.2. Timeline .....	108
9.3.3. Zero Simulation Time Regions .....	109
9.3.4. Module Axes .....	109
9.3.5. Gutter .....	110
9.3.6. Events .....	110
9.3.7. Messages .....	110
9.3.8. Displaying Module State on Axes .....	111
9.3.9. Zooming .....	111
9.3.10. Navigation .....	112
9.3.11. Tooltips .....	112
9.3.12. Bookmarks .....	112
9.3.13. Associated Views .....	112
9.3.14. Filtering .....	112
9.4. Eventlog Table .....	113
9.4.1. Display Mode .....	113
9.4.2. Name Mode .....	114
9.4.3. Type Mode .....	114
9.4.4. Line Filter .....	114
9.4.5. Navigation .....	114

9.4.6. Selection .....	115
9.4.7. Searching .....	115
9.4.8. Bookmarks .....	115
9.4.9. Tooltips .....	116
9.4.10. Associated Views .....	116
9.4.11. Filtering .....	116
9.5. Filter Dialog .....	116
9.5.1. Range Filter .....	117
9.5.2. Module Filter .....	117
9.5.3. Message Filter .....	117
9.5.4. Tracing Causes/Consequences .....	117
9.5.5. Collection Limits .....	118
9.5.6. Long-Running Operations .....	118
9.6. Other Features .....	118
9.6.1. Settings .....	119
9.6.2. Large File Support .....	119
9.6.3. Viewing a Running Simulation's Results .....	119
9.6.4. Caveats .....	119
9.7. Examples .....	119
9.7.1. Tictoc .....	120
9.7.2. FIFO .....	121
9.7.3. Routing .....	123
9.7.4. Wireless .....	124
10. Analyzing the Results .....	128
10.1. Overview .....	128
10.2. Creating Analysis Files .....	128
10.3. Using the Analysis Editor .....	129
10.3.1. Input Files .....	129
10.3.2. Datasets .....	132
10.3.3. Charts .....	142
10.4. Associated Views .....	148
10.4.1. Outline View .....	148
10.4.2. Properties View .....	148
10.4.3. Output Vector View .....	149
10.4.4. Dataset View .....	150
11. NED Documentation Generator .....	151
11.1. Overview .....	151
12. Extending the IDE .....	154
12.1. Installing New Features .....	154
12.2. Adding New Wizards .....	154
12.3. Project-Specific Extensions .....	154
A. Specification of the 'Compute Scalars' operation .....	155
A.1. Expressions .....	155
A.2. Computing Scalars .....	159

# Chapter 1. Introduction

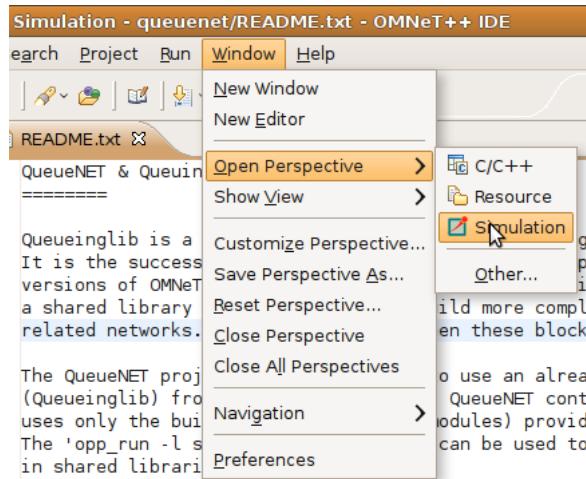
The OMNeT++ simulation IDE is based on the Eclipse platform and extends it with new editors, views, wizards, and other functionality. OMNeT++ adds functionality for creating and configuring models (NED and INI files), performing batch executions and analyzing the simulation results, while Eclipse provides C++ editing, SVN/GIT integration and other optional features (UML modeling, bug-tracker integration, database access, etc.) via various open-source and commercial plug-ins. The environment will be instantly recognizable to those at home with the Eclipse platform.

## 1.1. The Workbench

The Eclipse main window consists of various Views and Editors. These are collected into Perspectives that define which Views and Editors are visible and how they are sized and positioned.

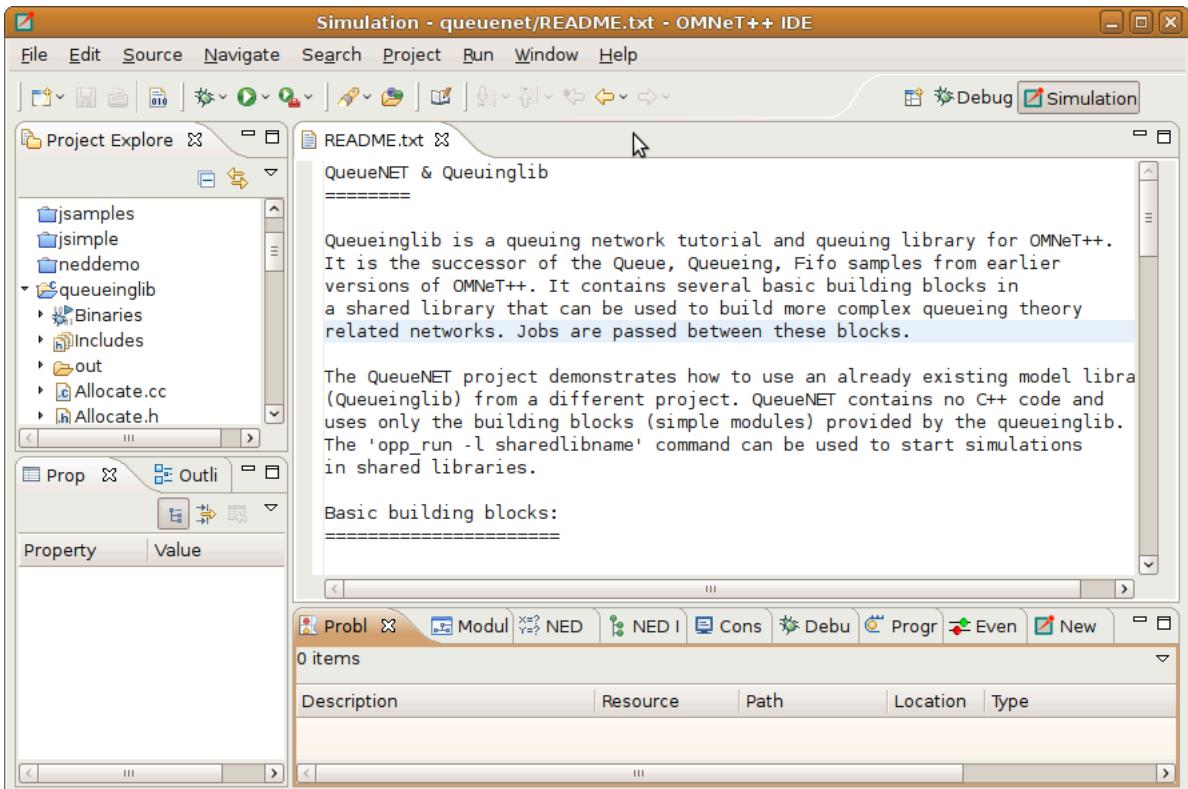
Eclipse is a very flexible system. You can move, resize, hide and show various panels, editors and navigators. This allows you to customize the IDE to your liking, but it also makes it more difficult to describe. First, we need to make sure that we are looking at the same thing.

The OMNeT++ IDE provides a "Simulation perspective" to work with simulation-related NED, INI and MSG files. To switch to the simulation perspective, select *Window | Open Perspective | Simulation*.



**Figure 1.1. Selecting the "Simulation Perspective" in Eclipse**

Most interface elements within Eclipse are can be moved or docked freely so you can construct your own workbench to fit your needs.



**Figure 1.2. Default layout of the OMNeT++ IDE**

The *Project Explorer* on the top left part of the screen shows the projects and their content in your workspace. In the example above, the *queueinglib* demo project is open. You can see the various .ned, .ini and other files inside. A number of views are docked at the bottom of the window.

The screenshot shows the open README.txt file in the editor area. When a user double-clicks on a file, Eclipse automatically launches the editor associated with that particular file type.

The *Properties View* contains information on the particular object that is selected in the editor area, or one of the other views that serve as a selection provider. The *Problems View* references code lines where Eclipse encountered problems.

Several OMNeT++-specific views exist that can be used during development. We will discuss how you can use them effectively in a later chapter. You can open any View by selecting *Window | Show View* from the menu.

## 1.2. Workspaces

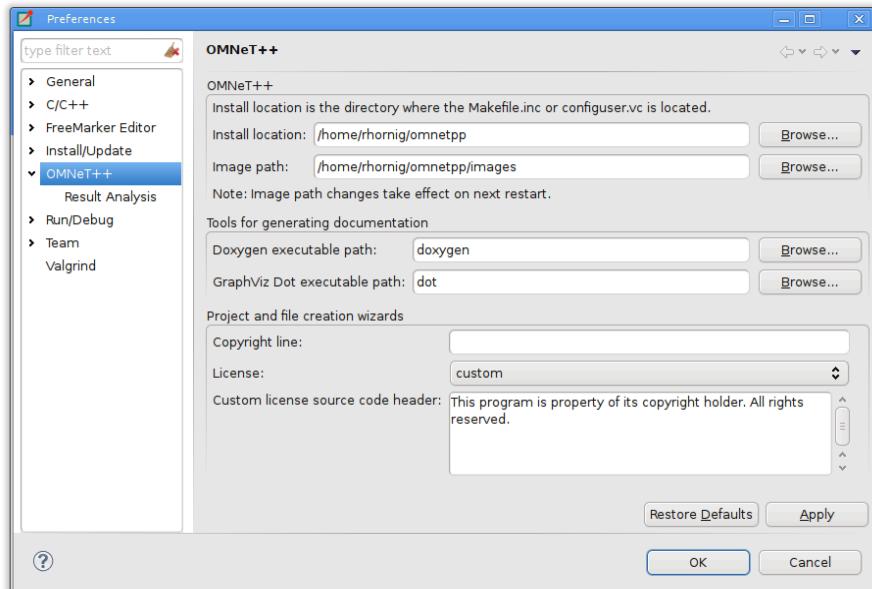
A workspace is basically a directory where all your projects are located. You may create and use several workspaces and switch between them as needed. During the first run, the OMNeT++ IDE offers to open the samples directory as the workspace, so you will be able to experiment with the available examples immediately. Once you start working on your own projects, we recommend that you create your own workspace by selecting *File | Switch Workspace | Other*. You can switch between workspaces, as necessary. Please be aware that the OMNeT++ IDE restarts with each switch in workspaces. This is normal. You can browse workspace content in the *Project Explorer*, *Navigator*, *C/C++ Projects* and similar views. We recommend using *Project Explorer*.

## 1.3. The Simulation Perspective

The OMNeT++ IDE defines the *Simulation Perspective* so that it is specifically geared towards the design of simulations. The *Simulation Perspective* is simply a set of conveniently selected views, arranged to make the creation of NED, INI and MSG files easier. If you are working with INI and NED files a lot, we recommend selecting this perspective. Other perspectives are optimized for different tasks like C++ development or debugging.

## 1.4. Configuring OMNeT++ Preferences

The OMNeT++ IDE preferences dialog is available through the standard preferences menu, which is under the main Window menu item. These settings are global and shared between all projects. The OMNeT++ install locations are automatically filled in for you after installation. The default settings for the NED documentation generation assume that the PATH environment variable is already set, so that third party tools can be found. The license configuration settings specify the preferred license type or a custom license text. The IDE will copy the license into new files and projects. The license will also be shown in the generated NED documentation.

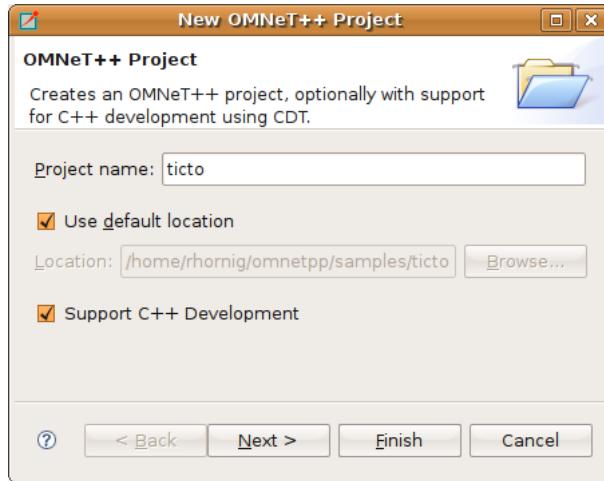


**Figure 1.3. Configuring OMNeT++ preferences**

Use the Browse buttons to find files or folders easily. Specify full path for executables if you do not want to extend the PATH environment variable.

## 1.5. Creating OMNeT++ Projects

In Eclipse, all files are within projects, so you will need a suitable project first. The project needs to be one designated as an OMNeT++ Project (in Eclipse lingo, it should have the OMNeT++ Nature). The easiest way to create such a project is to use a wizard. Choose *File|New|OMNeT++ Project...* from the menu, specify a project name, and click the *Finish* button. If you do not plan to write simple modules, you may unselect the *C++ Support* checkbox which will disable all C++ related features for the project.

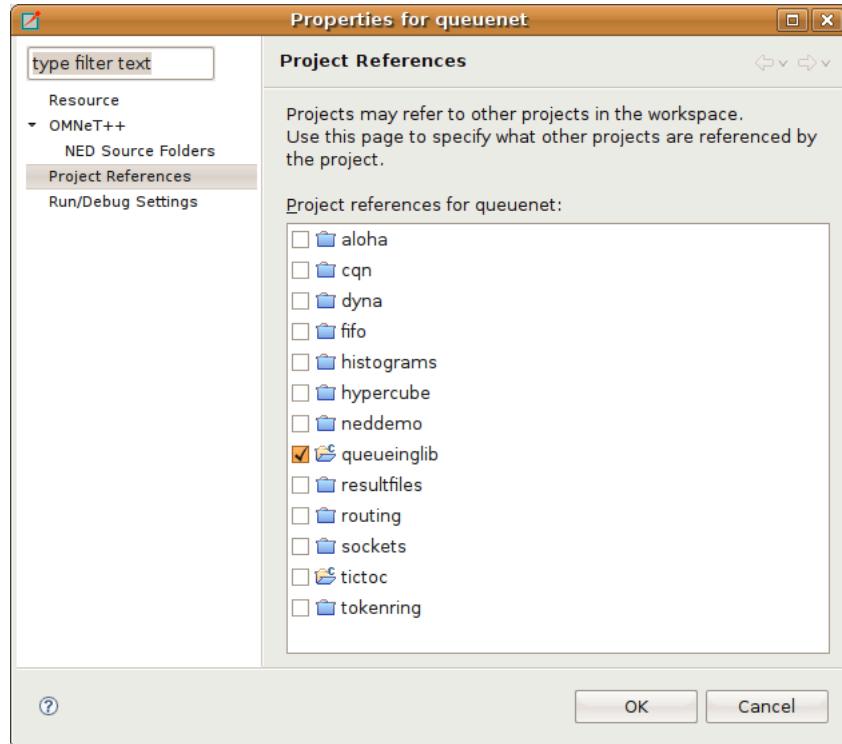


**Figure 1.4.** Creating a new OMNeT++ project

## 1.6. Project References

Most aspects of a project can be configured in the *Project Properties* dialog. The dialog is accessible via the *Project | Properties...* menu item, or by right-clicking the project in *Project Explorer* and choosing *Properties* from the context menu.

An important Eclipse concept is that a project may reference other projects in the workspace; project references can be configured in the *Project References* page of the properties dialog. To update the list of referenced projects, simply check those projects in the list that your project depends on, then click *Apply*. Note that circular references are not allowed (i.e. the dependency graph must be a tree).



**Figure 1.5.** Setting project dependencies

In the OMNeT++ IDE, all NED types, C++ code and build artifacts (executables, libraries) in a project are available to other projects that reference the given project.



To see an example of project references, check the `queuenet` and `queueinglib` example projects. In this example, `queuenet` references `queueinglib`. `Queueinglib` provides simple modules (NED files, and a prebuilt shared library that contains the code of the simple modules), and makes those modules available to `queuenet` that contains simulations (networks and ini files) built from them.

## 1.7. Getting Help

You may access the online help system from the *Help | Help Contents* menu item. The OMNeT++ IDE is built on top of Eclipse, so if you are not familiar with Eclipse, we recommend reading the *Workbench User Guide* and the *C/C++ Development User Guide* before starting to use OMNeT++-specific features.

---

# Chapter 2. Editing NED Files

## 2.1. Overview

When you double-click a .ned file in the IDE, it will open in the NED editor. The new NED editor is a dual-mode editor. In the editor's graphical mode, you can edit the network using the mouse. The textual mode allows you to work directly on the NED source.

When the IDE detects errors in a NED file, the problem will be flagged with an error marker in the *Project Explorer* and the *Problems View* will be updated to show the description and location of the problem. In addition, error markers will appear in the text window or on the graphical representation of the problematic component. Opening a NED file which contains an error will open the file in text mode. Switching to graphical mode is possible only if the NED file is syntactically correct.



As a side effect, if there are two modules with the same name and package in related projects, they will collide and both will be marked with an error. Furthermore, the name will be treated as undefined and any other modules depending on it will also generate an error (thus, a "no such module type" error may mean that there are actually multiple definitions which nullify each other).

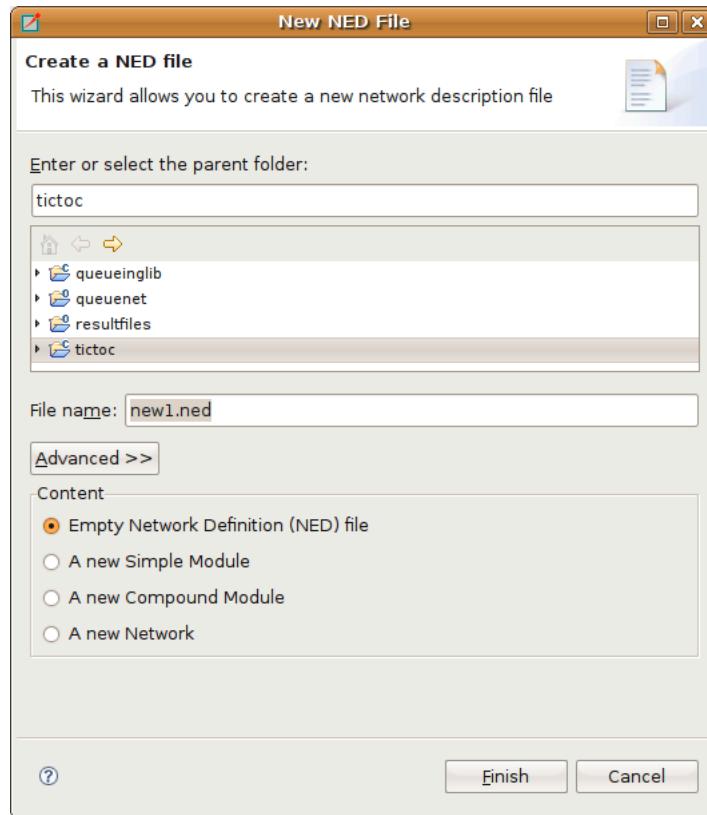
## 2.2. Opening Older NED Files

The syntax of NED files has changed significantly from the 3.x version. The NED editor primarily supports the new syntax. However, it is still possible to read and display NED files with the old syntax. It is important to note that many of the advanced features (syntax highlighting, content assistance, etc.) will not work with the old syntax. There is automatic conversion from the old syntax to the new, available both from the NED editor and as an external utility program (**nedtool**).

The **gned** program from OMNeT++ 3.x viewed NED files in isolation. In contrast, the OMNeT++ IDE gathers information from all .ned files in all open OMNeT++ projects and makes this information available to the NED editor. This is necessary because OMNeT++ 4.x modules may inherit parameters, visual appearance or even submodules and connections from other modules, so it is only possible to display a compound module correctly if all related NED definitions are available.

## 2.3. Creating New NED Files

Once you have an empty OMNeT++ project, you can create new NED files. Choosing *File|New|Network Description File* from the menu will bring up a wizard where you can specify the target directory and the file/module name. You may choose to create an empty NED file, a simple/compound module, or a network. Once you press the *Finish* button, a new NED file will be created with the requested content.



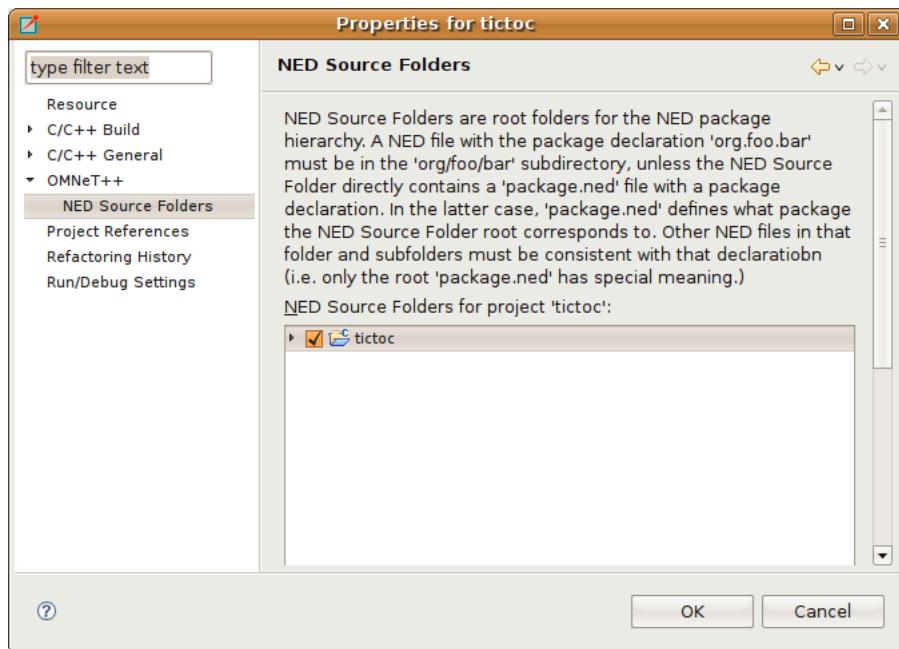
**Figure 2.1. Creating a new NED file**



Make sure that the NED file and the contained module have the same name. For example, a compound module named `Wireless42` should be defined within its own `Wireless42.ned` file.

### 2.3.1. NED Source Folders

It is possible to specify which folders the IDE should scan for NED files and that the IDE will use as the base directory for your NED package hierarchy. The IDE will not use any NED files outside the specified NED Source Folders and those files will be opened in a standard text editor. To specify the directory where the NED files will be stored, right-click on the project in the *Project Explorer* and choose *Properties*. Select the *OMNet++ | NED Source Folders* page and click on the folders where you store your NED files. The default value is the project root.



**Figure 2.2. Specifying which folder will hold your NED files**

## 2.4. Using the NED Editor

If you want to open an NED file, just double-click its icon in the *Project Explorer*. If the NED file can be parsed without an error, the graphical representation of the file will be opened; otherwise, the text view will be opened and the text will be annotated with error markers.



Only files located in NED Source Folders will be opened with the graphical editor. If a NED file is not in the NED Source Folders, it will be opened in a standard text editor.

You can switch between graphical and source editing mode by clicking the tabs at the bottom of the editor, or by using the **Alt+PGUP/PGDN** key combinations. The editor will try to keep the selection during the switch. Selecting an element in a graphical view and then switching to text view will move the cursor to the related element in the NED file. When switching back to graphical view, the graphical editor will try to select the element that corresponds to the cursor location in the NED source. This allows you to keep the context, even when switching back and forth.

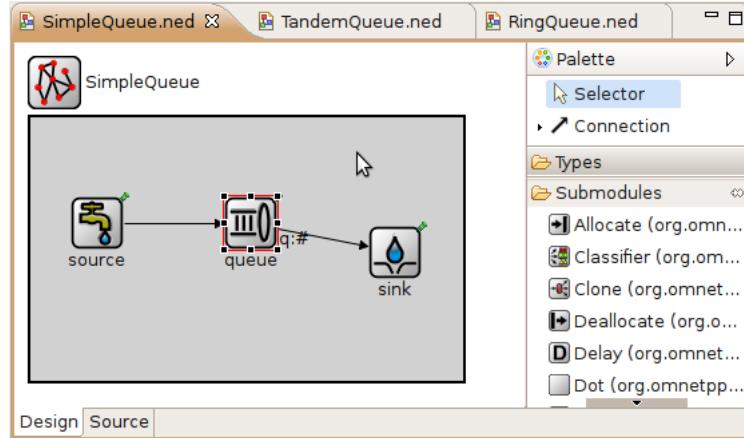
### 2.4.1. Editing in Graphical Mode

The graphical editor displays the visible elements of the loaded NED file. Simple modules, compound modules and networks are represented by figures or icons. Each NED file can contain more than one module or network. If it does, the corresponding figures will appear in the same order as they are found in the NED file.



Place only a single module or network into an NED file, and name the file according to the module name.

Simple modules and submodules are represented as icons while compound modules and networks are displayed as rectangles where other submodules can be dropped. Connections between submodules are represented either by lines or arrows depending on whether the connection was uni- or bi-directional. Submodules can be dragged or resized using the mouse and connected by using the Connection Tool in the palette.



**Figure 2.3. Graphical NED Editor**

The palette is normally to the right of the editor area. The upper part of the palette contains the basic tools: selector, connection selector, and the connection creator tool. To use a palette item, simply click on it. Then, click in the module where you want to place/activate it. The mouse pointer will give you feedback as to whether the requested operation is allowed. The middle part of the toolbox contains the basic elements that can be placed at the top level in a NED file (simple module, compound module, interface, channel, etc.) and a "generic" submodule. Click on any of these and then click into the editor area to create an instance. The bottom part of the palette contains all module types that can be instantiated as a submodule. They are shortcuts for creating a generic submodule and then modifying its type. They will display the default icon (if any) and a short description if you hover the mouse over them. You may configure the palette by right-clicking on a button and selecting *Settings...* or filter its content by selecting *Select Packages...*

Right-clicking any element in the edited NED file will bring up a context menu that allows several actions like changing the icon, pinning/unpinning a submodule, re-layouting a compound module, or deleting/renameing the element. There are also items to activate various views. For example, the *Properties View* allows you to edit properties of the element.

Hovering over an element will display its documentation (the comment in the NED source above the definition) as a tooltip. Pressing **F2** will make the tooltip window persistent, so it can be resized and scrolled for more convenient reading.

### Creating Modules

To create a module or a submodule, click on the appropriate palette item and then click where you want to place the new element. Submodules can be placed only inside compound modules or networks.

### Creating Types and Inner Types

To create a type, or an inner type inside a compound module, click on the appropriate palette item in the "Types" drawer, and then click where you want to place the new element. If you click on the background, a new top-level type will be created. Clicking on an existing compound module or network creates an inner type inside that module.

### Creating and Changing Connections

Select the *connection tool* (if there are channels defined in the project, you can use the dropdown to select the connection channel type). First, click the source module and then, the destination. A popup menu will appear, asking which gates should be

connected on the two selected modules. The tool will offer only valid connections (e.g. it will not offer to connect two output gates).

### **Reconnecting Modules**

Clicking and dragging a connection end point to another module will reconnect it (optionally, asking which gate should be connected). If you want to change only the gate, drag the connection end point and drop it over the original module. A popup will appear asking for the source or destination gate.

### **Selecting Elements**

Selecting an element is done by clicking on it or by dragging a rectangle over the target modules. A compound module can be selected by clicking on its border or title. If you want to select only connections within a selection rectangle, use the *connection selector* tool in the dropdown menu of the *connection tool*. The **Ctrl** and **Shift** keys can be used to add/remove to/from the current selection. Note that the keyboard (arrow keys) can also be used to navigate between submodules. You can also select using a selection rectangle by dragging the mouse around the modules.

### **Undo, Redo, Deleting Elements**

Use **Ctrl+Z** and **Ctrl+Y** for undo and redo, respectively, and the **DEL** key for deletion. These functions are also available in the *Edit* menu and in the context menu of the selected element.

### **Moving and Resizing Elements**

You can move/resize the selected elements with the mouse. Holding down **Shift** during move will perform a constrained (horizontal, diagonal or vertical) move operation. **Shift** + resize will keep the aspect ratio of the element.

If you turn on *Snap to Geometry* in the *View* menu, helper lines will appear to help you align with other modules. Selecting more than one submodule activates the *Alignment* menu (found both in the *View* menu and in the context menu).

### **Copying Elements**

Holding down **Ctrl** while dragging will clone the module(s). Copy/Paste can also be used both on single modules and with group selection.

### **Zooming**

Zooming in and out is possible from the *View* menu, or using **Ctrl+-**, **Ctrl+=**, or holding down **Ctrl** and using the mouse wheel.

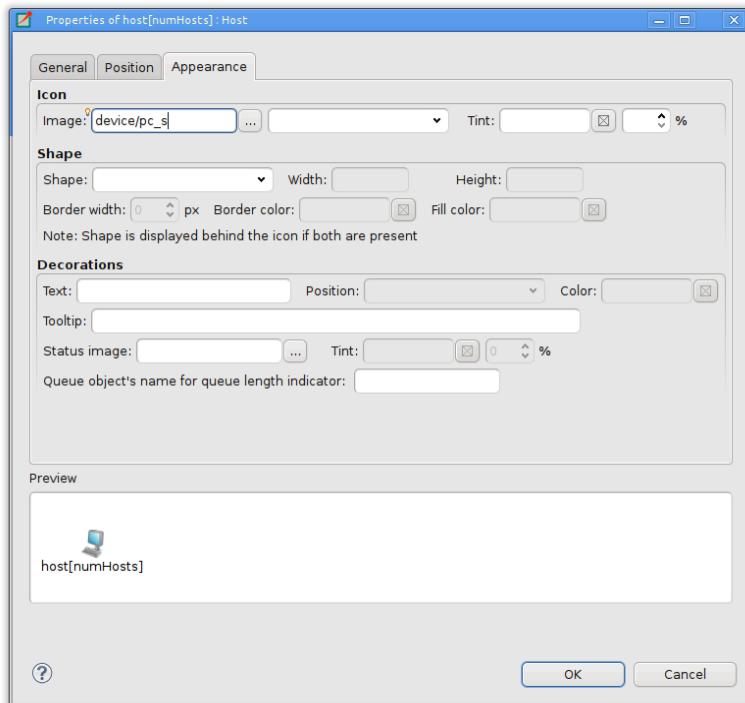
### **Pinning, Unpinning, Re-Layouting**

A submodule display string may or may not contain explicit coordinates for the submodule; if it does not, then the location of the submodule will be determined by the layouting algorithm. A submodule with explicit coordinates is pinned; one without is unpinned. The Pin action inserts the current coordinates into the display string and the Unpin action removes them. Moving a submodule also automatically pins it. The position of an unpinned module is undetermined and may change every time the layouting algorithm runs. For convenience, the layouter does not run when a submodule gets unpinned (so that the submodule does not jump away on unpinning), but this also means that unpinned submodules may appear at different locations next time the same NED file is opened.

### **Changing a Module Property**

To change a module property, right-click on it and select the *Properties...* menu item from the context menu or select the module and modify that property in the *Properties*

*View.* Alternatively, you can press **Ctrl+Enter** when the module is selected. NED properties like name, type and vector size are available on *General* tab. Visual properties like icon, size, color, border etc. can be set on the *Appearance* tab. You may check how your module will look like in the preview panel at the bottom of the dialog.

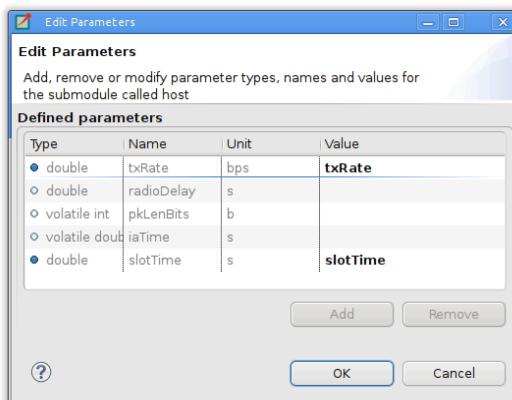


**Figure 2.4. Editing Visual Properties**

 You can select several modules at the same time and open the *Properties* dialog to set their common properties at the same time.

### Changing a Module Parameter

To change a module parameter, right-click on it and select the *Parameters...* menu item from the context menu. The dialog allows you to add or remove module parameters or assign value to them.



**Figure 2.5. Editing Module Parameters**

## Renaming Modules

To rename an existing module select its context menu and choose *Rename* or click on an already selected module a second time. You can specify a new name for the module or even turn a submodule into a vector by adding [vectorsize] after its name. Alternatively the name of a module can be set in the *Properties* dialog or can be edited by pressing **F6** when the module is selected.

## Exporting a Module as an Image

A module can be exported using several image formats by selecting *Export Image...* from the module's context menu.

## Navigation

Double-clicking a submodule will open the corresponding module type in a NED editor. Selecting an element in the graphical editor and then switching to text mode will place the cursor near the previously selected element in the text editor.

Navigating inside a longer NED file is easier if you open the *Outline View* to see the structure of the file. Selecting an element in the outline will select the same element in the graphical editor.

If you want to see the selected element in a different view, select the element and right-click on it. Choose *Show In* from the context menu, and select the desired view.

## Opening a NED Type

If you know only the name of a module type or other NED element, you can use the *Open NED Type* dialog by pressing **Ctrl+Shift+N**. Type the name, or search with wildcards. The requested type will be opened in an editor. This feature is not tied to the graphical editor: the Open NED Type dialog is available from anywhere in the IDE.

## Setting Properties

Elements of the display string and other properties associated with the selected elements can be edited in the *Properties View*. The Property View is grouped and hierarchically organized; however, you can switch off this behavior on the view toolbar. Most properties can be edited directly in the *Properties View*, but some also have specific editors that can be activated by pressing the ellipsis button at the end of the field. Fields marked with a small light bulb support content assist. Use the **Ctrl+SPACE** key combination to get a list of possible values.



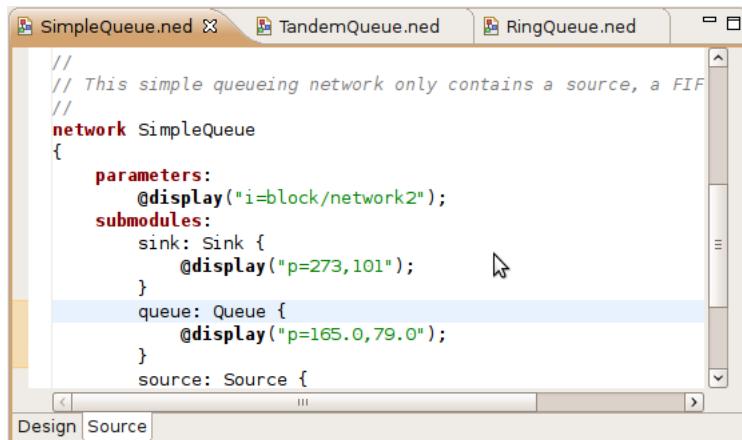
The following functions are available only in source editing mode:

- Creating or modifying gates
- Creating grouped and conditional connections
- Adding or editing properties

### 2.4.2. Editing in Source Mode

The NED source editor supports all functionality that one can expect from an Eclipse-based text editor, such as syntax highlighting, clipboard cut/copy/paste, unlimited undo/redo, folding, find/replace and incremental search.

The NED source is continually parsed as you type, and errors and warnings are displayed as markers on the editor rulers. At times when the NED text is syntactically correct, the editor has full knowledge of "what is what" in the text buffer.



**Figure 2.6. NED Source Editor**

### Basic Functions

- Undo (**Ctrl+Z**), Redo (**Ctrl+Y**)
- Indent/unindent code blocks (**TAB/Shift+TAB**)
- Correct indentation (NED syntax aware) (**Ctrl+I**)
- Find (**Ctrl+F**), incremental search (**Ctrl+J**)
- Move lines (**Alt+UP/DOWN**)



The following functions can help you explore the IDE:

- **Ctrl+Shift+L** pops up a window that lists all keyboard bindings, and
- **Ctrl+3** brings up a filtered list of all available commands.

### Converting to the New NED Syntax

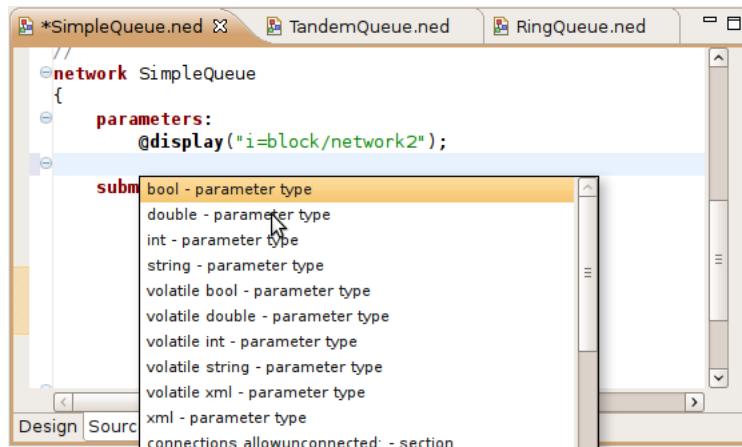
If you have an NED file with older syntax, you can still open it. A context menu item allows you to convert it to the new syntax. If the NED file is already using the new syntax, the *Convert to 4.x Format* menu item is disabled.

### View Documentation

Hovering the mouse over a NED type name will display the documentation in a "tooltip" window, which can be made persistent by hitting **F2**.

### Content Assist

If you need help, just press **Ctrl+SPACE**. The editor will offer possible words or templates. This is context sensitive, so it will offer only valid suggestions. Content assist is also a good way of exploring the new NED syntax and features.



**Figure 2.7. NED Source Editor with content assist activated**

### Searching in NED Files

Selecting a text or moving the cursor over a word and pressing **Ctrl+Shift+G** searches for the selection in all NED files in the open projects. This function lets you quickly find references to the word or type currently under the cursor. The results are shown in the standard *Search View*.

### Organizing Imports

Sometimes, it is very inconvenient to add the necessary import statements to the beginning of the NED file by hand. The IDE can do it for you (almost) automatically. Pressing **Ctrl+Shift+O** will cause the IDE to try to insert all necessary import statements. You will be prompted to specify the used packages in case of ambiguity.

### Cleaning Up NED Files

This function does a general repair on all selected NED files by throwing out or adding import statements as needed, checking (and fixing) the file's package declaration, and reformatting the source code. It can be activated by clicking on the *Project | Clean Up NED Files* menu item from the main menu.

### Commenting

To comment out the selected lines, press **Ctrl+{/}**. To remove the comment, press **Ctrl +{/}** again.

### Formatting the Source Code

It is possible to reformat the whole NED file according to the recommended coding guidelines by activating the *Format Source* context menu item or by pressing the **Ctrl +Shift+F** key combination.

 Using the graphical editor and switching to source mode automatically re-formats the NED source code, as well.

### Navigation

Holding the **Ctrl** key and clicking any identifier type will jump to the definition. Alternatively, move the cursor into the identifier and hit **F3** to achieve the same effect.

If you switch to graphical mode from text mode, the editor will try to locate the NED element under the cursor and select it in the graphical editor.

The Eclipse platform's bookmarking and navigation history facilities also work in the NED editor.

## 2.5. Associated Views

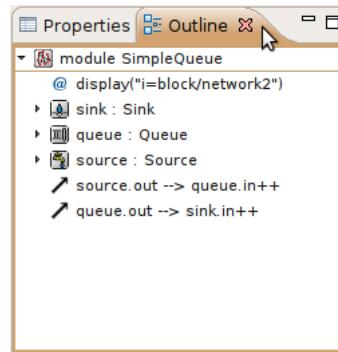
There are several views related to the NED editor. These views can be displayed (if not already open) by choosing *Window | Show View* in the menu or by selecting a NED element in the graphical editor and selecting *Show In* from the context menu.



If you are working with very large NED files, you may improve the performance of the editor by closing all NED file related views you do not need.

### 2.5.1. Outline View

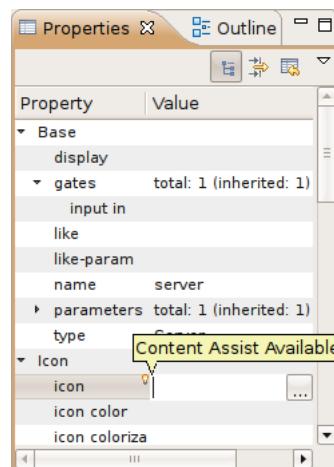
The *Outline View* allows an overview of the current NED file. Clicking on an element will select the corresponding element in the text or graphical view. It has limited editing functionality; you can copy/cut/paste and delete an object.



**Figure 2.8. Outline View**

### 2.5.2. Property View

The *Property View* contains all properties of the selected graphical element. Visual appearance, name, type and other properties can be changed in this view. Some fields have specialized editors that can be activated by clicking on the ellipsis button in the field editor. Fields marked with a small light bulb icon have content assist support. Pressing **Ctrl+SPACE** will display the possible values the field can hold.



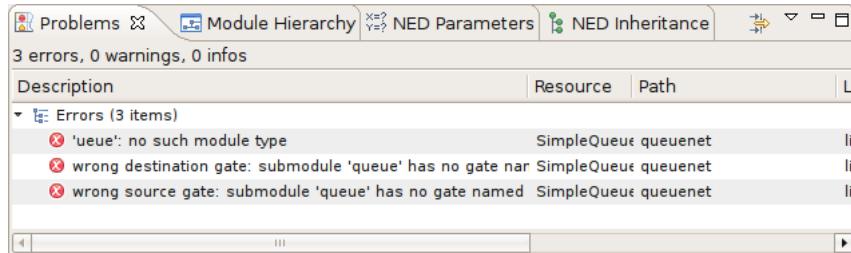
**Figure 2.9. Properties View**

### 2.5.3. Palette View

The Palette is normally displayed on the left or right side of the editor area and contains tools to create various NED elements. It is possible to hide the Palette by clicking on the little arrow in the corner. You can also detach it from the editor and display it as a normal Eclipse View (*Window | Show View | Other... | General | Palette*).

### 2.5.4. Problems View

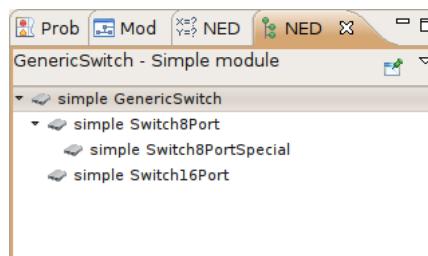
The *Problems View* contains error and warning messages generated by the parser. Double-clicking a line will open the problematic file and move to the appropriate marker.



**Figure 2.10. Problems View**

### 2.5.5. NED Inheritance View

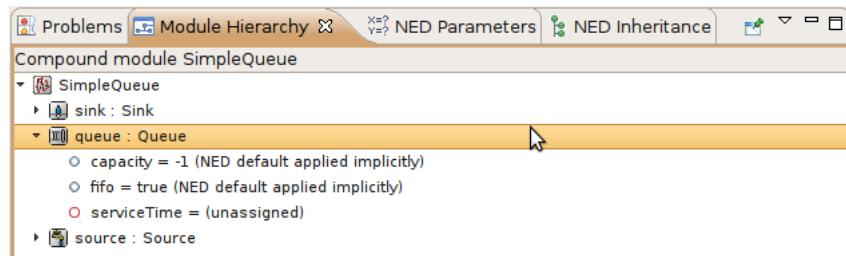
The *Inheritance View* displays the relationship between different NED types. Select a NED element in the graphical editor or move the cursor into a NED definition and the *Inheritance View* will display the ancestors of this type. If you do not want the view to follow the selection in the editor, click the Pin icon on the view toolbar. This will fix the displayed type to the currently selected one.



**Figure 2.11. NED Inheritance View**

### 2.5.6. Module Hierarchy View

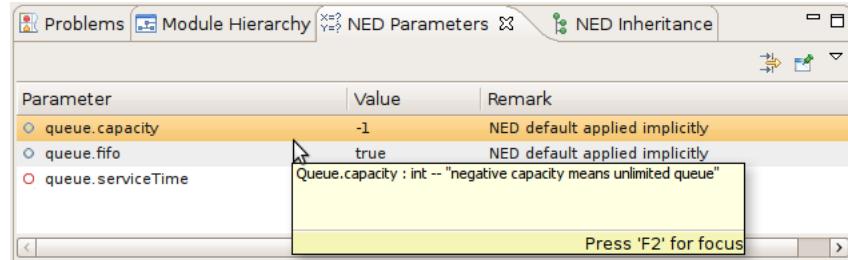
The *Module Hierarchy View* shows the contained submodules and their parameters, several levels deep. It also displays the parameters and other contained features.



**Figure 2.12. Module Hierarchy View**

### 2.5.7. Parameters View

The *Parameters View* shows the parameters of the selected module including inherited parameters.



Parameter	Value	Remark
queue.capacity	-1	NED default applied implicitly
queue fifo	true	NED default applied implicitly
queue.serviceTime		Queue.capacity : int -- "negative capacity means unlimited queue"

**Figure 2.13. Outline View**

The latter two views are used mainly with the INI File Editor.

# Chapter 3. Editing INI Files

## 3.1. Overview

In OMNeT++, simulation models are parameterized and configured for execution using configuration files with the .ini extension, called INI files. Ini files are text files, which can be edited using any text editor. However, OMNeT++ 4.x introduces a tool expressly designed for editing INI files. The INI File Editor is part of the OMNeT++ IDE and is very effective in assisting the user to author INI files. It is a very useful feature because it has detailed knowledge of the simulation model, the INI file syntax, and the available configuration options.



The syntax and features of INI files have changed since OMNeT++ 3.x. These changes are summarized in the "Configuring Simulations" chapter of the "OMNeT++ 4.x. User Manual".

The INI File Editor is a dual-mode editor. The configuration can be edited using forms and dialogs, or as plain text. Forms are organized around topics like general setup, Cmdenv, Tkenv, output files, extensions and so on. The text editor provides syntax highlighting and auto completion. Several views can display information, which is useful when editing INI files. For example you can see the errors in the current INI file or all the available module parameters in one view. You can easily navigate from the module parameters to their declaration in the NED file.

## 3.2. Creating INI Files

To create a new INI file, choose *File | New | Initialization File* from the menu. It opens a wizard where you can enter the name of the new file and select the name of the network to be configured.

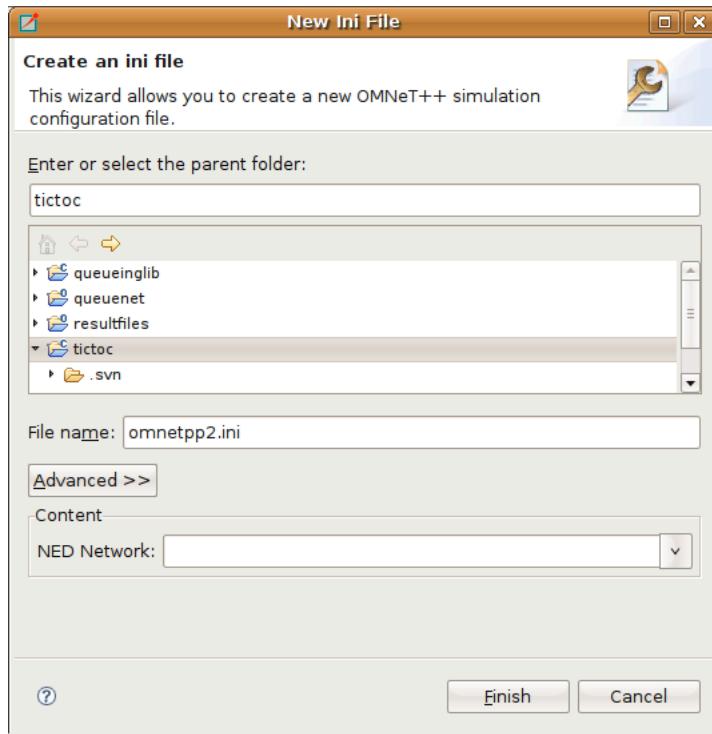


Figure 3.1. New Initialization File dialog

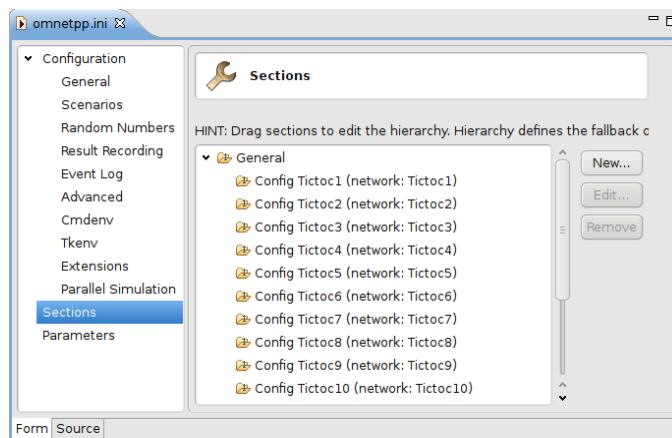
### 3.3. Using the INI File Editor

The INI File Editor has two modes. The *Source* mode provides a text editor with syntax highlighting and auto completion of names. In the *Form* mode, you can edit the configuration by entering the values in a form. You can switch between the modes by selecting the tabs at the bottom of the editor.

#### 3.3.1. Editing in Form Mode

The INI file contains the configuration of simulation runs. The content of the INI file is divided into sections. In the simplest case, all parameters are set in the General section. If you want to create several configurations in the same INI file, you can create named Configuration (Config) sections and refer to them with the `-c` option when starting the simulation. The Config sections inherit the settings from the General section or from other Config sections. This way you can factor out the common settings into a "base" configuration.

On the first page of the form editor, you can edit the sections. The sections are displayed as a tree; the nodes inherit settings from their parents. The icon before the section name shows how many runs are configured in that section (see Table 3.1, "Legend of Icons Before Sections"). You can use drag and drop to reorganize the sections. You can delete, edit, or add a new child to the selected section.

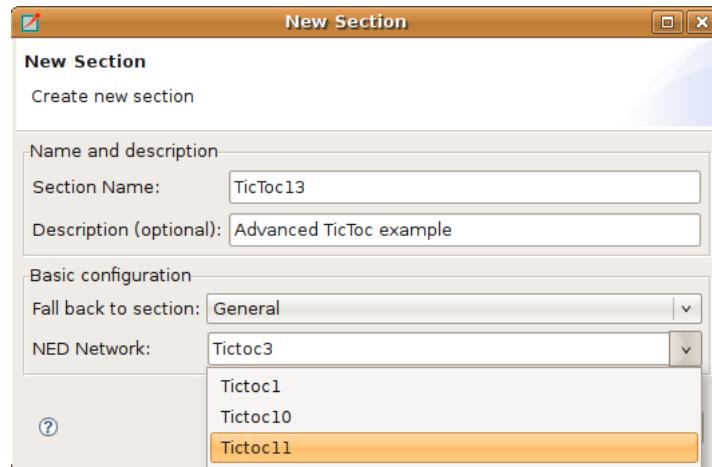


**Figure 3.2. Editing INI file sections**

	contains a single run
	contains multiple replications (specified by 'repeat=...')
	contains iteration variables
	contains multiple replications for each iteration

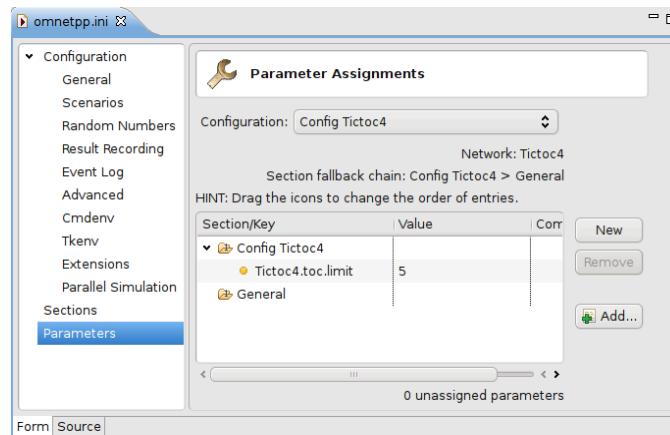
**Table 3.1. Legend of Icons Before Sections**

The Config sections have a name and an optional description. You can specify a fallback section other than General. If the network name is not inherited, it can be specified, as well.



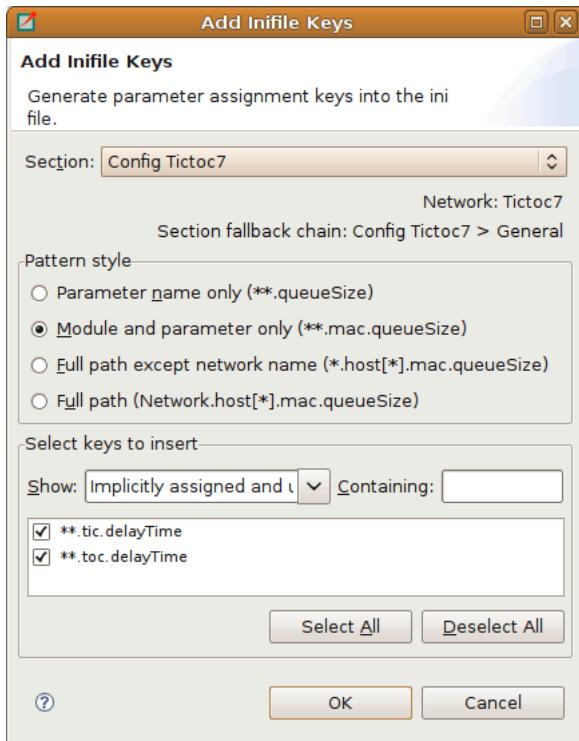
**Figure 3.3. Creating a new INI file section**

On the *Parameters* page of the form editor, you can set module parameters. First, you have to select the section where the parameters are stored. After selecting the section from the list, the form shows the name of the edited network and the fallback section. The table below the list box shows current settings of the section and all other sections from which it has inherited settings. You can move parameters by dragging them. If you click a table cell, you can edit the parameter name (or pattern), its value and the comment attached to it. **Ctrl+Space** brings up a content assist. If you hover over a table row, the parameter is described in the tooltip that appears.



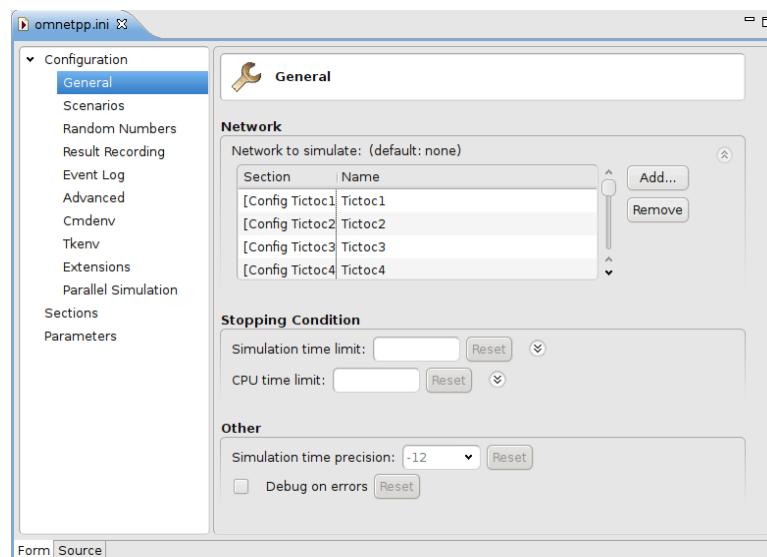
**Figure 3.4. Editing module parameters**

New parameters can be added one by one by pressing the *New* button and filling the new table row. The selected parameters can be removed with the *Remove* button. If you press the *Add...* button, you can add any missing parameters.



**Figure 3.5. Add missing module parameters dialog**

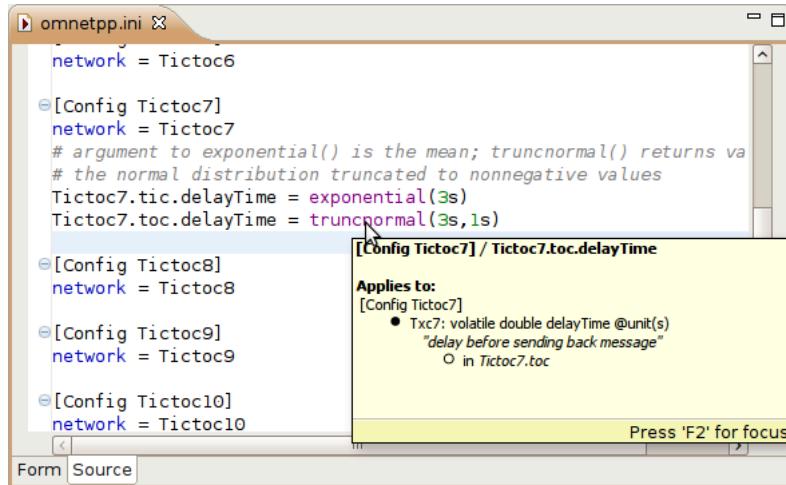
The rest of the settings do not belong to modules (e.g. configuration of random number generators, output vectors, simulation time limit). These settings can be edited from the forms listed under the Configuration node. If the field has a default value and it is not set, the default value is displayed in gray. If its value is set, you can reset the default value by pressing the *Reset* button. These fields are usually set in the General section. If you want to specify them in a Config section, press the  $\downarrow$  button and add a section-specific value to the opening table. If the table contains the Generic section only, then it can be collapsed again by pressing the  $\uparrow$  button. Some fields can be specified in the General section only, so they do not have a  $\downarrow$  button next to them.



**Figure 3.6. Editing general configuration**

### 3.3.2. Editing in Text Mode

If you want to edit the INI file as plain text, switch to the Source mode. The editor provides several features in addition to the usual text editor functions like copy/paste, undo/redo and text search.



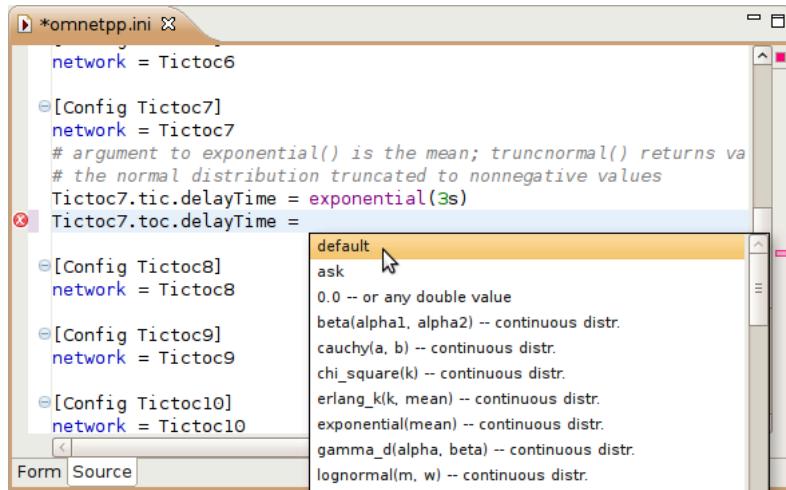
**Figure 3.7. Editing the INI file in text mode**

### Opening Old INI Files

When you open an INI file with the old format, the editor offers to convert it to the new format. It creates Config sections from Run sections and renames old parameters.

### Content Assist

If you press **Ctrl+Space**, you will get a list of proposals valid at the insertion point. The list may contain section names, general options, and parameter names and values of the modules of the configured network.



**Figure 3.8. Content assist in source mode**

### Tooltip

If you hover over a section or parameter, a tooltip appears showing the properties of the section or parameter. The tooltip for sections displays the inheritance chain, the network name, number of errors and warnings and the yet unassigned parameters. For parameters, the definition, description and the module name are displayed.

## Add Unassigned Parameters

You can add the names of unassigned module parameters to a Config section by choosing *Add Missing keys...* from the context menu or pressing **Ctrl+Shift+O**.

## Commenting

To comment out the selected lines, press **Ctrl+/.** To remove the comment, press **Ctrl+/-** again.

## Navigation

If you press the **Ctrl** key and click on a module parameter name, then the declaration of the parameter will be shown in the NED editor. You can navigate from a network name to its definition, too.

## Error Markers

Errors are marked on the left/right side of the editor. You can move to the next/previous error by pressing **Ctrl+.** and **Ctrl+,**, respectively. You can get the error message in a tooltip if you hover over the  marker.

## 3.4. Associated Views

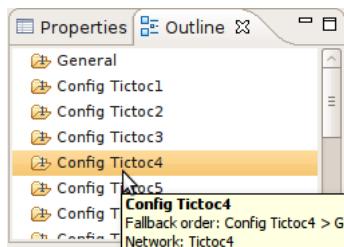
There are several views related to the INI editor. These views can be displayed (if not already open) by choosing the view from the *Window | Show View* submenu.



If you are working with very large NED or INI files, you may improve the performance of the editor by closing all views related to INI files (Parameters, Module Hierarchy and NED Inheritance View).

### 3.4.1. Outline View

The *Outline View* allows an overview of the sections in the current INI file. Clicking on a section will highlight the corresponding element in the text or form view.



**Figure 3.9. Outline View showing the content of an INI file**

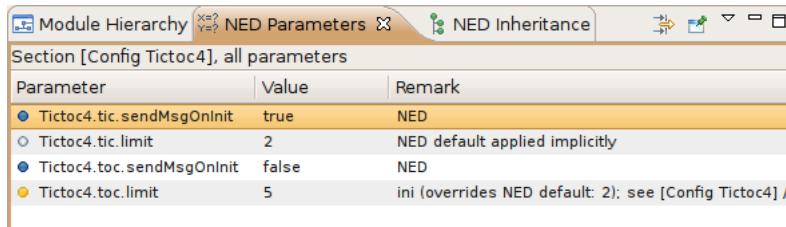
### 3.4.2. Problems View

The *Problems View* contains error and warning messages generated by the parser. Double-clicking on a row will open the problematic file and move to the location of the problem.

### 3.4.3. Parameters View

The *Parameters View* shows parameters of the selected section including inherited parameters. It also displays the parameters that are unassigned in the configuration. When the  toggle button on the toolbar is on, then all parameters are displayed; otherwise, only the unassigned ones are visible.

If you want to fix the content of the view, press the button. After pinning, the content of this view will not follow the selection made by the user in other editors or views.



**Figure 3.10. Parameters View**

	value is set in the NED file
	default from the NED file is explicitly set in the INI file (** .paramname=default)
	default from the NED file is automatically applied, because no value is specified in the INI file
	value is set in the INI file (may override the value from the NED file)
	value is set in the INI file to the same value as the NED default
	will ask the user at runtime (** .paramname=ask)
	unassigned -- has no values specified in the NED or INI files

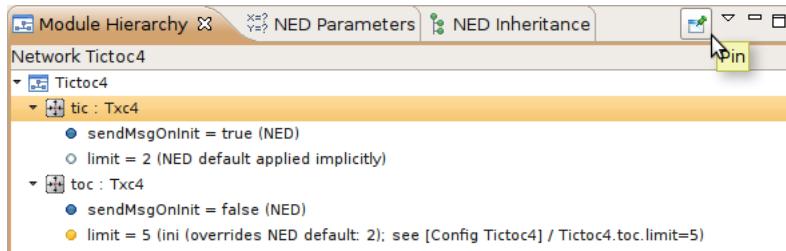
**Table 3.2. Legend of icons before module parameters**



Right-clicking on any line will show a context menu that allows you to navigate to the definition of that parameter or module.

#### 3.4.4. Module Hierarchy View

The *Module Hierarchy View* shows the contained submodules, several levels deep. It also displays the module parameters, and where its value comes from (INI file, NED file or unassigned).



**Figure 3.11. Module Hierarchy View**



Before you use the context menu to navigate to the NED definition, pin down the hierarchy view. This way you will not lose the current context and content if the view will not follow the selection.

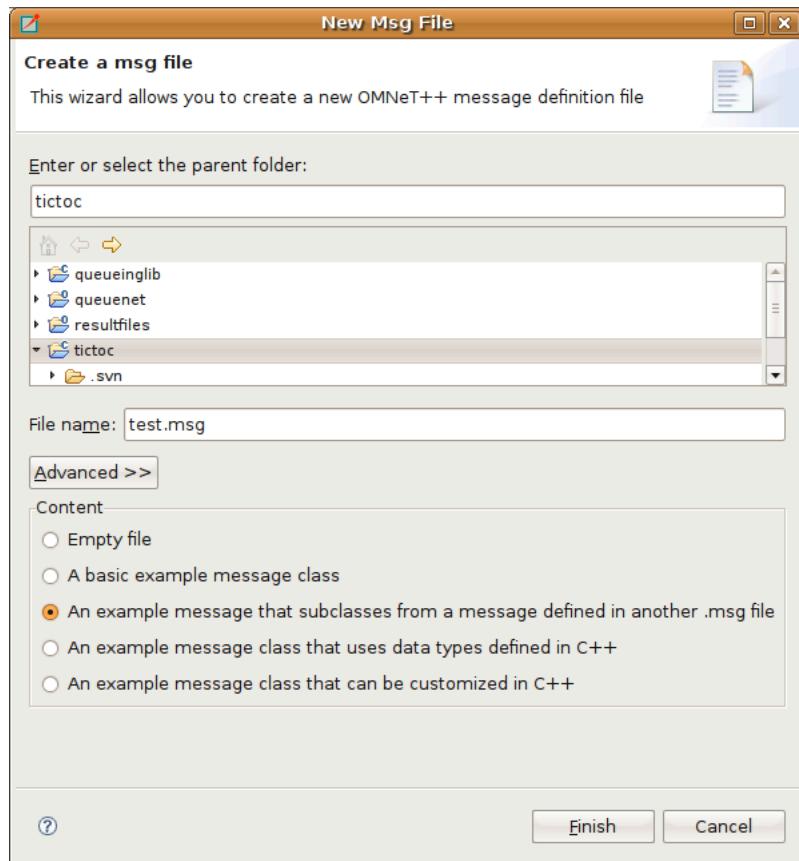
#### 3.4.5. NED Inheritance View

The *NED Inheritance View* shows the inheritance tree of the network configured in the selected section.

# Chapter 4. Editing Message Files

## 4.1. Creating Message Files

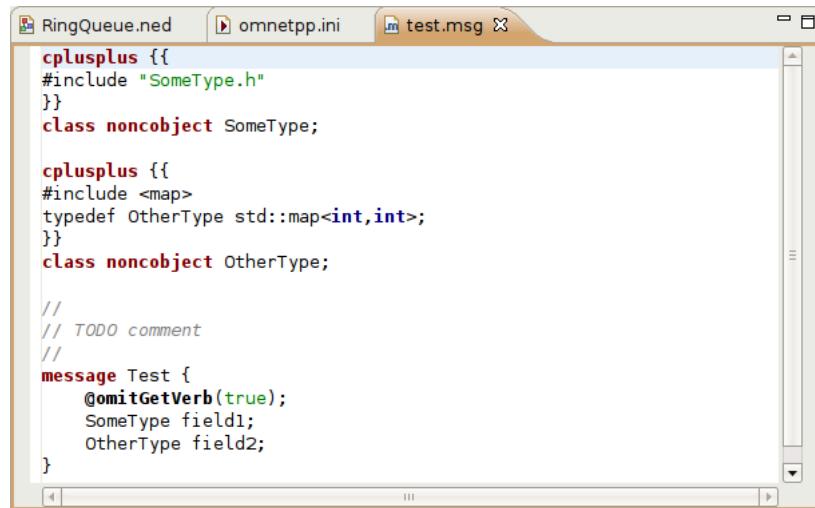
Choosing *File | New | Message Definition (.msg)* from the menu will bring up a wizard where you can specify the target directory and the file name for your message definition. You may choose to create an empty MSG file, or choose from the predefined templates. Once you press the *Finish* button, a new MSG file will be created with the requested content.



**Figure 4.1. Creating a new MSG file**

## 4.2. The Message File Editor

The message file editor is a basic text editor with syntax highlight support.



The screenshot shows a window titled "RingQueue.ned" containing C++ code for message definitions. The code includes declarations for classes `SomeType` and `OtherType`, and a message `Test` with fields `field1` and `field2`. A note in the code indicates a TODO comment.

```
cplusplus {{  
    #include "SomeType.h"  
}  
class noncobject SomeType;  
  
cplusplus {{  
    #include <map>  
    typedef OtherType std::map<int,int>;  
}  
class noncobject OtherType;  
  
//  
// TODO comment  
//  
message Test {  
    @omitGetVerb(true);  
    SomeType field1;  
    OtherType field2;  
}
```

**Figure 4.2. Message File Editor**

 Currently the editor does not support advanced features like content assistance or syntax aware folding.

---

# Chapter 5. C++ Development

## 5.1. Introduction

The OMNeT++ IDE contains editors, views and other tools to assist you while developing your C++ code. C++ files open in the IDE in the C++ source editor. The C++ source editor supports syntax highlighting, documentation tooltips, content assist, automatic indentation, code formatting, refactoring, and several other useful features. The IDE also allows you to configure the build, start the build process, launch simulations, and debug the model without leaving the IDE.

Most of these features are provided by the Eclipse CDT (C/C++ Development Tooling) project (<http://eclipse.org/cdt>). This chapter briefly explains the basics of using CDT for developing simulation models. If you want to learn more about how to use CDT effectively, we recommend that you read the CDT documentation in the IDE help system (Help/Help Content).

The OMNeT++ IDE extends CDT with the following features to make model development easier:

- A new OMNeT++ project creation wizard allows you to create simple, working simulation models in one step.
- Makefiles are automatically generated for your project based on the project build configuration. The built-in makefile generator is compatible with the command line `opp_makemake` tool, and features deep makefiles, recursive make, cross-project references, invoking the message compiler, automatic linking with the OMNeT++ libraries; and can build executables, shared libraries or static libraries.
- Makefile generation and the project build system can be configured using a GUI interface.
- Project Features: Large projects can be partitioned into smaller units which can be independently excluded or included in the build. Disabling parts of the project can significantly reduce build time or make it possible to build the project at all.

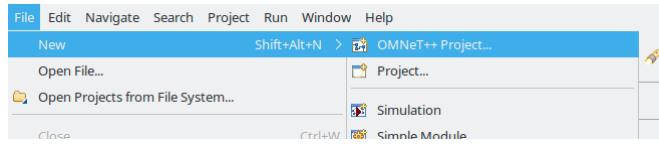
## 5.2. Prerequisites

The OMNeT++ IDE (and the OMNeT++ simulation framework itself) requires a pre-installed compiler toolchain to function properly.

- On Windows: The OMNeT++ distribution comes with a preconfigured MinGW compiler toolchain, there is no need to manually install anything. By default, the IDE uses the Clang compiler from MinGW, but it is also possible to switch to the GCC compiler (also part of MinGW).
- On Linux: By default, the Clang compiler is used, but OMNeT++ falls back to use GCC if Clang is not present on the system. You have to install Clang or GCC on your system before trying to compile a simulation with OMNeT++. Please read the Install Guide for detailed instructions.
- On macOS: You need to install Xcode Developer Tools to get compiler support before trying to compile a simulation with OMNeT++. Please read the Install Guide for detailed instructions.

## 5.3. Creating a C++ Project

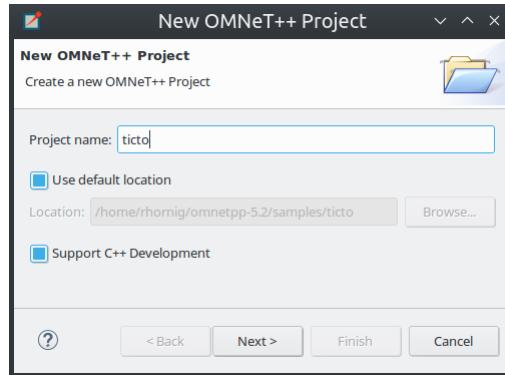
To create an OMNeT++ project that supports C++ development, select *File | New | OMNeT++ Project*.



**Figure 5.1. Creating an OMNeT++ project**

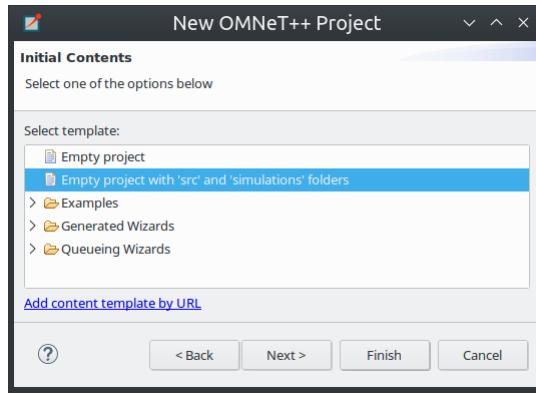
This menu item will bring up the *New OMNeT++ Project* wizard. The wizard lets you create an OMNeT++-specific project, which includes support for NED, MSG and INI file editing, as well as C++ development of simple modules.

On the first page of the wizard, specify the project name and ensure that the *Support C++ Development* checkbox is selected.



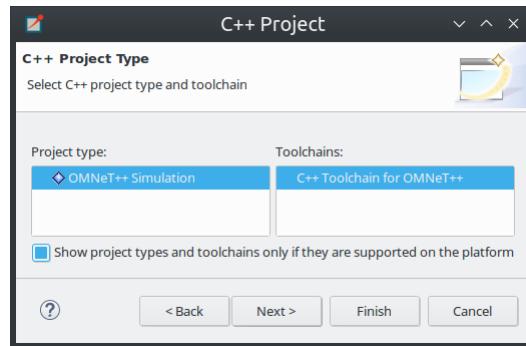
**Figure 5.2. Setting project name and enabling C++ support**

Select a project template. A template defines the initial content and layout of the project.



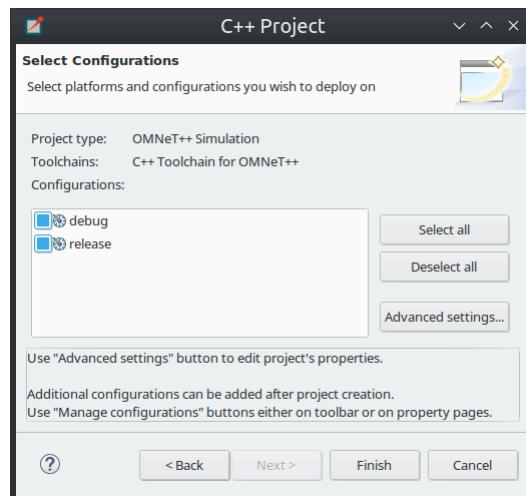
**Figure 5.3. Selecting a project template**

Select a toolchain that is supported on your platform. Usually you will see only a single supported toolchain, so there is no need to change anything on the page.



**Figure 5.4. Selecting a toolchain**

Finally, select one or more from the preset build configurations. A configuration is a set of options that are associated with the build process. It is used mainly to build debug and release versions of your program.

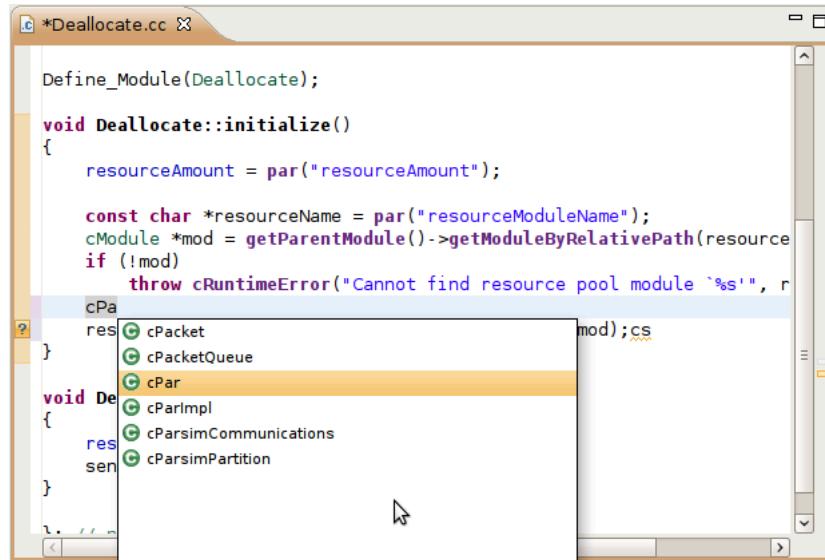


**Figure 5.5. Selecting configurations**

Pressing the *Finish* button will create the project.

## 5.4. Editing C++ Code

The OMNeT++ IDE comes with a C/C++ editor. In addition to standard editing features, the C/C++ editor provides syntax highlighting, content assistance, and other C++ specific functionality. The source is continually parsed as you type, and errors and warnings are displayed as markers on the editor rulers.



**Figure 5.6. C++ source editor**

#### 5.4.1. The C++ Editor

The C++ source editor provides the usual features of Eclipse-based text editors, such as syntax highlighting, clipboard cut/copy/paste, unlimited undo/redo, folding, find/replace and incremental search.

The IDE scans and indexes the C++ files in your project in the background, and provides navigation and code analysis features on top of that knowledge; this database is kept up to date as you edit the source.

##### Basic Functions

Some of the most useful features of the source editor:

- Undo (**Ctrl+Z**), Redo (**Ctrl+Y**)
- Switch between a C++ source and its matching header file (**Ctrl+TAB**)
- Indent/unindent code blocks (**TAB/Shift+TAB**)
- Correct indentation (**Ctrl+I**)
- Move lines (**Alt+UP/DOWN**)
- Find (**Ctrl+F**), incremental search (**Ctrl+J**)



The following functions can help you explore the IDE:

- **Ctrl+Shift+L** pops up a window that lists all keyboard bindings, and
- **Ctrl+3** brings up a filtered list of all available commands.

##### View Documentation

Hovering the mouse over an identifier will display its declaration and the documentation comment in a "tooltip" window. The window can be made persistent by hitting **F2**.



If you are on Ubuntu and you see all-black tooltips, you need to change the tooltip colors in Ubuntu; see the Ubuntu chapter of the *OMNeT++ Installation Guide* for details.

## Content Assist

If you need help, just press **Ctrl+SPACE**. The editor will offer possible completions (variable names, type names, argument lists, etc.).

## Navigation

Hitting **F3** or holding the **Ctrl** key and clicking an identifier will jump to the definition/declaration.

The Eclipse platform's bookmarking and navigation history facilities are also available in the C++ editor.

## Commenting

To comment out the selected lines, press **Ctrl+{/}**. To remove the comment, press **Ctrl+{/}** again.

## Open Type

Pressing **Ctrl+Shift+T** will bring up the *Open Element* dialog which lets you type a class name, method name or other identifier, and opens its declaration in a new editor.

## Exploring the Code

The editor offers various ways to explore the code: Open Declaration (**F3**), Open Type Hierarchy (**F4**), Open Call Hierarchy (**Ctrl+Alt+H**), Quick Outline (**Ctrl+O**), Quick Type Hierarchy (**Ctrl+T**), Explore Macro Expansion (**Ctrl+=**), Search for References (**Ctrl+Shift+G**), etc.

## Refactoring

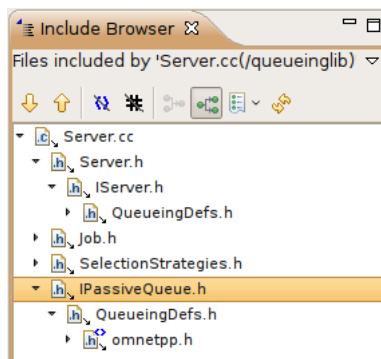
Several refactoring operations are available, for example Rename (**Shift+Alt+R**).



Several features such as content assist, go to definition, type hierarchy and refactorings rely on the *Index*. The index contains the locations of all functions, classes, enums, defines, etc. in the project and referenced projects. Initial indexing of large projects may take a significant amount of time. The index is kept up to date mostly automatically, but occasionally it may be necessary to manually request reindexing the project. Index-related actions can be found in the *Index* submenu of the project's context menu.

### 5.4.2. Include Browser View

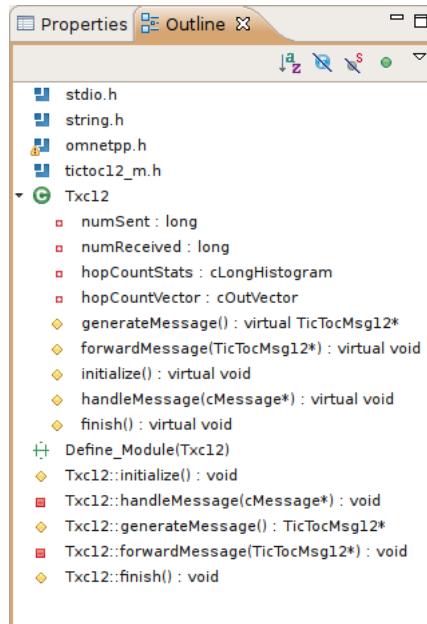
Dropping a C++ file into the *Include Browser View* displays the include files used by the C++ file (either directly or indirectly).



**Figure 5.7. Include Browser**

### 5.4.3. Outline View

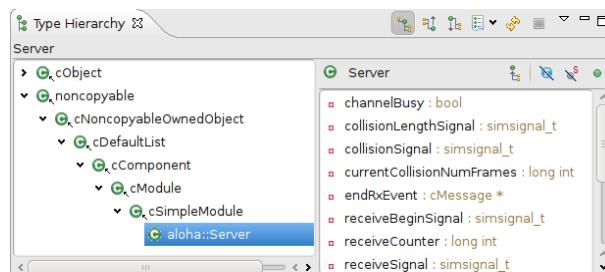
During source editing, the *Outline View* gives you an overview of the structure of your source file and can be used to quickly navigate inside the file.



**Figure 5.8. Navigating with Outline View**

### 5.4.4. Type Hierarchy View

Displaying the C++ type hierarchy may be helpful for understanding the inheritance relationships among your classes (and among OMNeT++ classes).

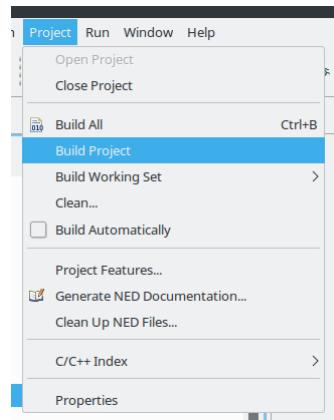


**Figure 5.9. C++ Type hierarchy**

## 5.5. Building the Project

### 5.5.1. Basics

Once you have created your source files and configured your project settings, you can build the project by selecting *Build Project* from the *Project* menu or from the project context menu. You can also press **Ctrl+B** to build all open projects in the workspace.



**Figure 5.10. Building a project**

### Build Output

The build output (standard output and standard error) is displayed in the *Console View* as the build progresses. Errors and warnings parsed from the output are displayed in the *Problems View*. Double-clicking a line in the *Problems View* will jump to the corresponding source line. Conversely, the *Console View* is more useful when you want to look at the build messages in their natural order (*Problems View* is usually sorted), for example when you get a lot of build errors and you want to begin by looking at the first one.

### Makefile Generation

When you start the build process, a makefile will be created or refreshed in each folder where makefile creation is configured. After that, make will be invoked with the `all` target in the folder configured as build root.

 During the build process, the makefile will print out only the names of the compiled files. If you want to see the full command line used to compile each file, specify `V=1` (verbose on) on the make command line. To add this option, open *Project Properties | C/C++ Build | Behavior (tab)* and replace `all` with `all V=1` on the *Build* target line.

### Cleaning the Project

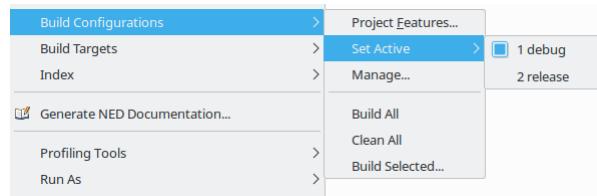
To clean the project, choose `Clean...` from the *Project* menu, or `Clean Project` from the project context menu. This will invoke `make` with the `clean` target in the project's build root folder, and also in referenced projects. To only clean the local project and keep referenced projects intact, use `Clean Local` item from the project context menu (see next section).

### Referenced Projects and the Build Process

When you start the build, the IDE will build the referenced projects first. When you clean the project, the IDE will also clean the referenced projects first. This is often inconvenient (especially if your project depends on a large third party project). To avoid cleaning the referenced projects, use `Clean Local` from the project context menu.

### Build Configurations

A project is built using the active build configuration. A project may have several build configurations, where each configuration selects a compiler toolchain, debug or release mode, defines symbols, etc. To set the active build configuration, choose *Build Configurations | Set Active* from the project context menu.



**Figure 5.11. Activating a build configuration**

### 5.5.2. Console View

The *Console View* displays the output of the build process.

A screenshot of the Eclipse IDE showing the 'Console' view. The tab bar at the top has 'Problems', 'Tasks', 'Console' (which is selected and highlighted in blue), and 'Properties'. Below the tabs, the title bar says 'C-Build [tictoc]'. The main area contains the following text:

```
**** Build of configuration gcc-debug for project tictoc ****

make MODE=debug CONFIGNAME=gcc-debug all
g++ -Wl,--export-dynamic -Wl,-rpath,/home/rhornig/omnetpp/lib -o out/gcc-debug//tictoc
out/gcc-debug//txc7.o out/gcc-debug//txc2.o out/gcc-debug//txc4.o out/gcc-debug//txc1.o
out/gcc-debug//txc3.o out/gcc-debug//txc9.o out/gcc-debug//txc10.o out/gcc-debug//txc6.o
out/gcc-debug//txc5.o out/gcc-debug//txc11.o out/gcc-debug//txc8.o out/gcc-debug//
txc12.o out/gcc-debug//tictoc10_m.o out/gcc-debug//tictoc11_m.o out/gcc-debug//
tictoc12_m.o -Wl,--whole-archive -Wl,--no-whole-archive -L"/home/rhornig/omnetpp/lib/
gcc" -L"/home/rhornig/omnetpp/lib" -u _tkenv_lib -lenvird -ltkenvd -u _cmdenv_lib -
lenvird -lcmdenvd -lsim_std -ldl -lstdc++
ln -s -f out/gcc-debug//tictoc .
```

**Figure 5.12. Build output in a console**

### 5.5.3. Problems View

The *Problems View* contains the errors and warnings generated by the build process. You can browse the problem list and double-click any message to go to the problem location in the source file. NED file and INI file problems are also reported in this view along with C++ problems. The editors are annotated with these markers, too. Hover over an error marker in the editor window to get the corresponding message as a tooltip.

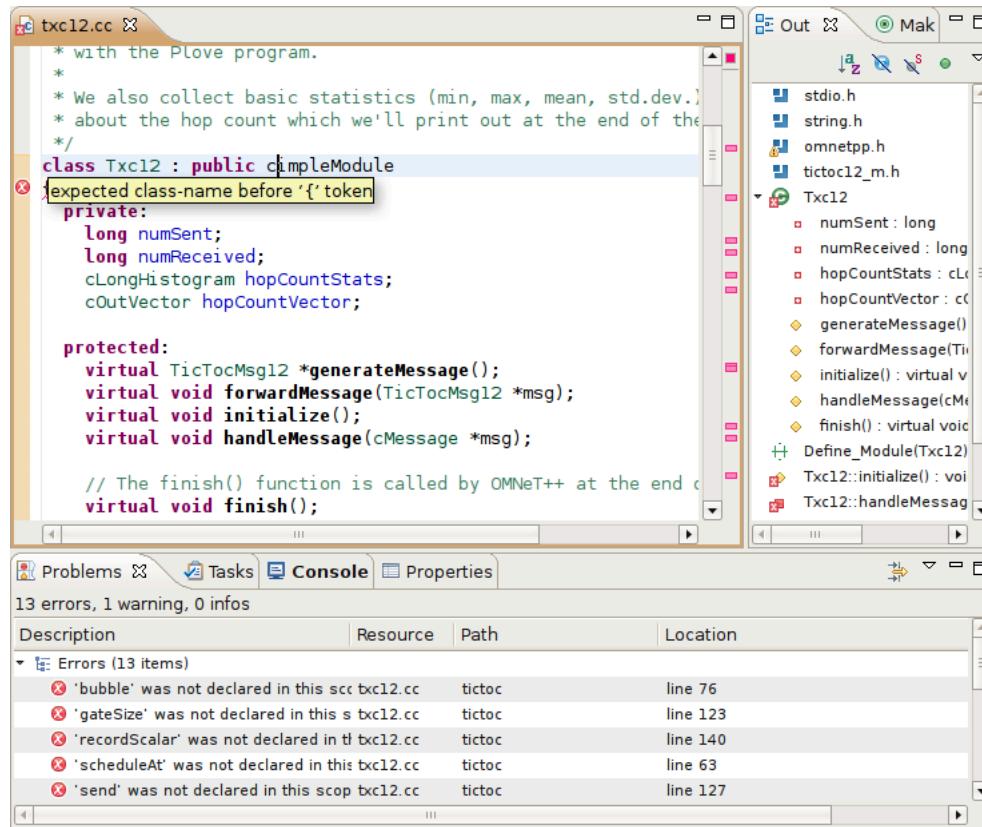


Figure 5.13. C++ problems

## 5.6. Configuring the Project

### 5.6.1. Configuring the Build Process

The make invocation can be configured on *C/C++ Build* page of the *Project Properties* dialog. Most settings are already set correctly and do not need to be changed. One exception is the *Enable parallel build* option on the *Behavior* tab that you may want to enable, especially if you have a multi-core computer.



Do not set the number of parallel jobs to be significantly higher than the number of CPU cores you have. In particular, never turn on the *Use unlimited jobs* option, as it will start an excessive number of compile processes, and can easily consume all available memory in the system.

We do not recommend that you change any setting on property pages under the *C/C++ Build* tree node.

### 5.6.2. Managing Build Configurations

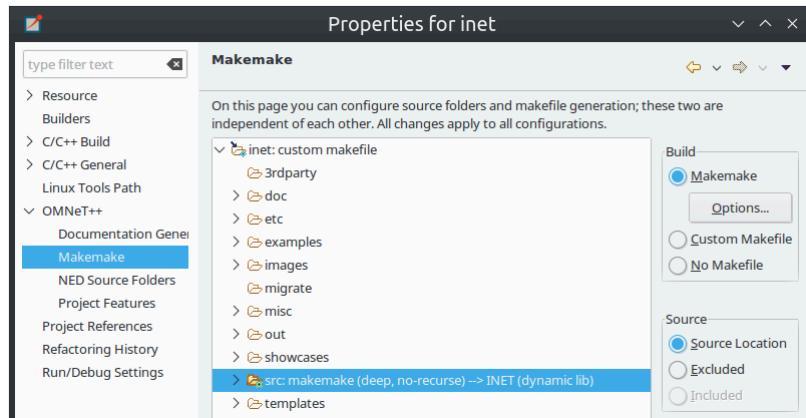
A project may have several build configurations, where each configuration describes the selected compiler toolchain, debug or release mode, any potential extra include and linker paths, defined symbols, etc. You can activate, create or delete build configurations under the *Build Configurations* submenu of the project context menu.

 Make sure that the names of all configurations contain the "debug" or the "release" substring. The IDE launcher uses the name of the configuration to switch the matching configuration depending whether you want to debug or run the simulation.

### 5.6.3. Configuring the Project Build System

OMNeT++ uses makefiles to build the project. You can use a single makefile for the whole project, or a hierarchy of makefiles. Each makefile may be hand-written (i.e. provided by you), or generated automatically. The IDE provides several options for automatically created makefiles.

The build system for an OMNeT++ project can be configured on the *OMNeT++ | Makemake* page of the *Project Properties* dialog. All settings you do in this page will affect all build configurations.



**Figure 5.14. Configuring Makefiles**

#### Folders and Makefiles

The page displays the folder tree of the project. Using controls on the page (*Build* group in the top-right corner), you can declare that a selected folder contains a hand-written (custom) makefile, or tell the IDE to generate a makefile for you. Generated makefiles will be automatically refreshed before each build. If a makefile is configured for a folder, the makefile kind will be indicated with a small decoration on the folder icon.

The build root folder is indicated with a small arrow. This is the folder in which the IDE's *Build* function will invoke the make command, so it should contain a makefile. It is expected that this makefile will build the whole project, so it is supposed to invoke all other makefiles, either directly or indirectly. By default, the build root folder is the project root. This is usually fine, but if you really need to change the project build root, overwrite the *Build location* setting in the *C/C++ Build* page of the same dialog.



All generated makefiles will be named `Makefile`; custom makefiles are also expected to have this name.

#### Source Folders

In addition to makefiles, you also need to specify where your C++ files are located (source folders). This is usually the `src` folder of the project, or, for small projects, the project root. It is also possible to exclude folders from a source folder. The controls on the right-bottom part of the dialog (*Source* group) allow you to set up source folders and exclusions on the project. Source files that are outside source folders or are in an excluded folder will be ignored by both the IDE and the build process.



Source folders and exclusions that you configure on this page actually modify the contents of the *Source Location* tab of the *C++ General | Paths and Symbols* page of the project properties dialog; the changes will affect all build configurations.

Automatically created makefiles are by default *deep*, meaning that they include all (non-excluded) source files under them in the build. That is, a source file will be included in the build if it is both under a source folder and covered by a makefile. (This applies to automatically generated makefiles; the IDE has obviously no control over the behaviour of custom makefiles.)

## Makefile Generation

Makefile generation for the selected folder can be configured on the *Makemake Options* dialog, which can be brought up by clicking the *Options* button on the page. The dialog is described in the next section.

## Command-line Build

To re-create your makefiles on the command line, you can export the settings by pressing the *Export* button. This action will create a file with the name `makemakefiles`. After exporting, execute `make -f makemakefiles` from the command line.

### 5.6.4. Configuring Makefile Generation for a Folder

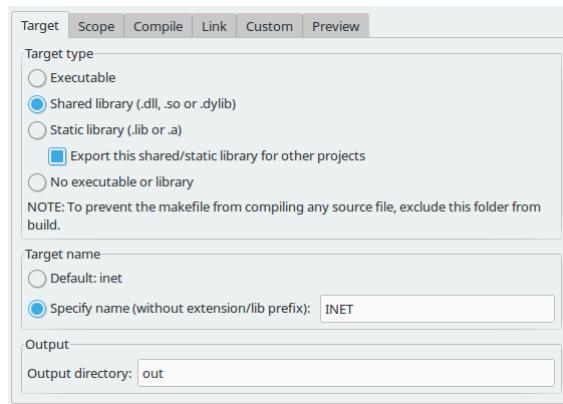
Makefile generation for a folder can be configured in the *Makemake Options* dialog. To access the dialog, open the *OMNet++ | Makemake* page in the *Project Properties* dialog, select the folder, make sure makefile generation is enabled for it, and click the *Options* button.

The following sections describe each page of the dialog.

#### The Target Tab

On the first, *Target* tab of the dialog, you can specify how the final target of the makefile is created.

- *Target type*: The build target can be an executable, a shared or static library, or the linking step may be omitted altogether. Makemake options: `--make-so`, `--make-lib`, `--nolink`
- *Export this shared/static library for other projects*: This option is observed if a library (shared or static) is selected as target type, and works in conjunction with the *Link with libraries exported from referenced projects* option on the *Link* tab. Namely, referencing projects will automatically link with this library if both the library is exported from this project AND linking with exported libraries is enabled in the referencing project. Makemake option: `--meta:export-library`
- *Target name*: You may set the target name. The default value is derived from the project name. Makemake option: `-o` (If you are building a debug configuration, the target name will be implicitly suffixed by the `_dbg` string.)
- *Output directory*: The output directory specifies where the object files and the final target will be created, relative to the project root. Makemake option: `-O`



**Figure 5.15. Target definition**

### The Scope Tab

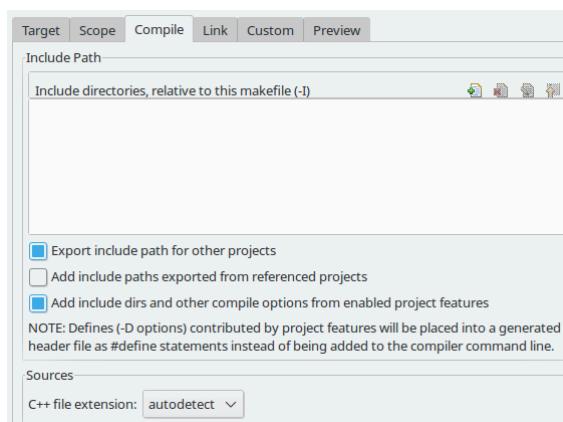


**Figure 5.16. Scope of makefile**

The Scope tab allows you to configure the scope of the makefile and which source files will be included.

- *Deep compile*: When enabled, the makefile will compile the source files in the whole subdirectory tree (except excluded folders and folder covered by other makefiles). When disabled, the makefile only compiles sources in the makefile's folder. Makemake option: `--deep`
- *Recursive make*: When enabled, the build will invoke make in all descendant folders that are configured to contain a makefile. Makemake option: `--meta:recurse` (resolves to multiple `-d` options)
- *More » Additionally invoke make in the following directories*: If you want to invoke additional makefiles from this makefile, specify which directories should be visited (relative to this makefile). This option is useful if you want to invoke makefiles outside this source tree. Makemake option: `-d`

### The Compile Tab



**Figure 5.17. Compiler options**

The *Compile* tab allows you to adjust the parameters passed to the compiler during the build process.

Settings that affect the include path:

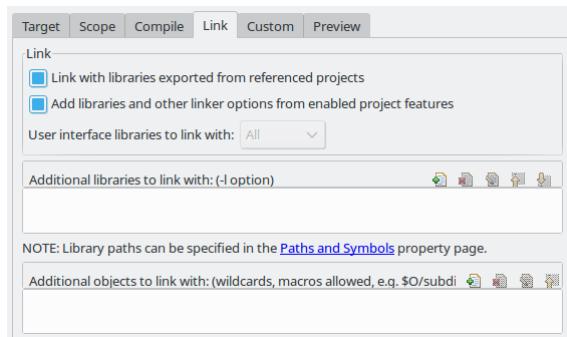
- *Export include path for other projects* makes this project's include path available for other dependent projects. This is usually required if your project expects that other independent models will extend it in the future.
- *Add include paths exported from referenced projects* allows a dependent project to use header files from the dependencies if those projects have exported their include path (i.e. the above option was turned on.)
- *Add include dirs and other compile options from enabled project features*: Project features may require additional include paths and defines to compile properly. Enabling this option will add those command line arguments (specified in the *.oppfeatures* file) to the compiler command line.

Source files:

- *C++ file extension*: You can specify the source file extension you are using in the project (.cc or .cpp). We recommend that you use .cc in your projects. Makemake option: -e

## The **Link** Tab

Link options allow the user to fine-tune the linking steps at the end of the build process.



**Figure 5.18. Linker options**

- *Link with libraries exported from referenced projects*: If your project references other projects that build static or dynamic libraries, you can instruct the linker to automatically link with those libraries by enabling this option. The libraries from the other projects must be exported via the *Export this shared/static library for other projects* option on the *Target* tab. Makemake option: --meta:use-exported-libs
- *Add libraries and other linker options from enabled project features*: Project features may require additional libraries and linker options to build properly. Enabling this option will add those command line arguments (specified in the *.oppfeatures* file) to the linker command line.
- *User interface libraries to link with*: If the makefile target is an executable, you may specify which OMNeT++ user interface libraries (Cmdenv, Tkenv, or both) should be linked into the program. Makemake option: -u
- *More » Additional libraries to link with*: This box allows you to specify additional libraries to link with. Specify the library name without its path, possible prefix (lib) and file extension, and also without the -l option. The library must be on the linker

path; the linker path can be edited on the *Library Paths* tab of the *C/C++ General | Paths and Symbols* page of the *Project Properties* dialog. Makemake option: `-l`

- *More » Additional objects to link with:* Additional object files and libraries can be specified here. The files must be given with their full paths and file extension. Wildcards and makefile macros are also accepted. Example: `$O/subdir/*.o`. Makemake option: none (files will become plain makemake arguments)

### The Custom Tab

The *Custom* tab allows the customization of the makefiles by inserting handwritten makefile fragments into the automatically generated makefile. This lets you contribute additional targets, rules, variables, etc.

- *Makefrag:* If the folder contains a file named `makefrag`, its contents will be automatically copied into the generated makefile, just above the first target rule. `makefrag` allows you to customize the generated makefile to some extent; for example, you can add new targets (e.g. to generate documentation or run a test suite), new rules (e.g. to generate source files during the build), override the default target, add new dependencies to existing targets, overwrite variables, etc. The dialog lets you edit the contents of the `makefrag` file directly (it will be saved when you accept the dialog).
- *More » Fragment files to include:* Here, it is possible to explicitly specify a list of makefile fragment files to include, instead of the default `makefrag`. Makemake option: `-i`

### The Preview Tab

The *Preview* tab displays the command line options that will be passed to `opp_makemake` to generate the makefile. It consists of two parts:

- *Makemake options:* This is an *editable* list of makefile generation options. Most options map directly to checkboxes, edit fields and other controls on the previous tabs of the dialog. For example, if you check the *Deep compile* checkbox on the *Scope* tab, the `--deep` option will be added to the command line. Conversely, if you delete `--deep` from the command line options, that will cause the *Deep compile* checkbox to be unchecked. Some options are directly `opp_makemake` options, others are "meta" options that will be resolved to one or more `opp_makemake` options; see below.
- *Makemake options modified with CDT settings and with meta-options resolved:* This is a read-only text field, displayed for information purposes only. Not all options in the above options list are directly understood by `opp_makemake`; namely, the options that start with `--meta:` denote higher-level features offered by the IDE only. Meta options will be translated to `opp_makemake` options by the IDE. For example, `--meta:auto-include-path` will be resolved by the IDE to multiple `-I` options, one for each directory in the C++ source trees of the project. This field shows the `opp_makemake` options after the resolution of the meta options.

### 5.6.5. Project References and Makefile Generation

When your project references another project (say the INET Framework), your project's build will be affected in the following way:

- *Include path:* Source folders in referenced projects will be automatically added to the include path of your makefile if the *Add include paths exported from referenced projects* option on the *Compile* tab is checked where the referenced projects also enable the *Export include path for other projects* option.
- *Linking:* If the *Link with libraries exported from referenced projects* option on the *Link* tab is enabled, then the makefile target will be linked with those libraries in

referenced projects that have the *Export this shared/static library for other projects* option checked on the *Target* tab.

- NED types: NED types defined in a referenced project are automatically available in referencing projects.

## 5.7. Project Features

### 5.7.1. Motivation

Long compile times are often an inconvenience when working with large OMNeT++-based model frameworks like the INET Framework. The IDE feature named *Project Features* lets you reduce build times by excluding or disabling parts of the model framework that you do not use for your simulation study. For example, when you are working on mobile ad-hoc simulations in INET, you can disable the compilation of Ethernet, IPv6/MIPv6, MPLS, and other unrelated protocol models. The word *feature* refers to a piece of the project codebase that can be turned off as a whole.

Additional benefits of project features include a less cluttered model palette in the NED editor, being able to exclude code that does not compile on your system, and enforcing cleaner separation of unrelated parts in the model framework.



A similar effect could also be achieved via breaking up the model framework (e.g. INET) into several smaller projects, but that would cause other kinds of inconveniences for model developers and users alike.

### 5.7.2. What is a Project Feature

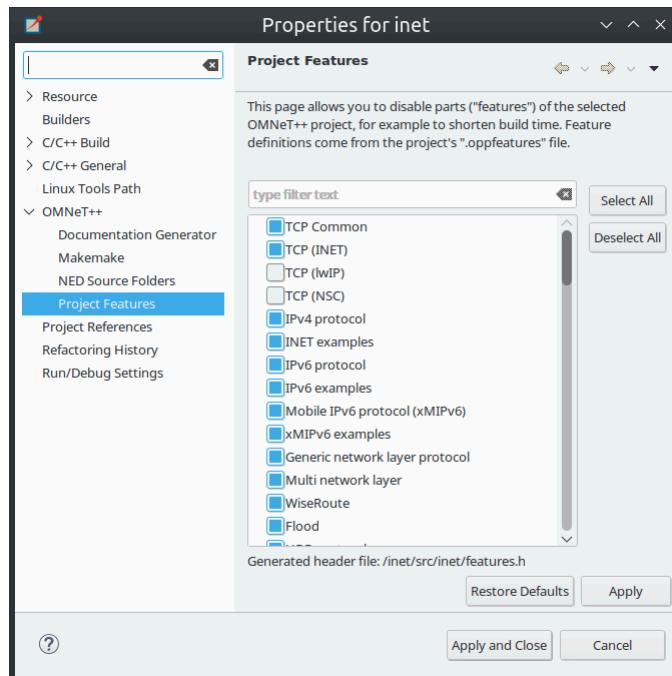
Features can be defined per project. As already mentioned, a feature is a piece of the project codebase that can be turned off as a whole, that is, excluded from the C++ sources (and thus from the build) and also from NED. Feature definitions are typically written and distributed by the author of the project; end users are only presented with the option of enabling/disabling those features. A feature definition contains:

- *ID*, which is a unique identifier inside the feature definition file.
- *Feature name*, for example "UDP" or "Mobility examples".
- *Feature description*. This is a few sentences of text describing what the feature is or does; for example "Implementation of the UDP protocol".
- *Labels*. This is a list of labels or keywords that facilitate grouping or finding features.
- *Initially enabled*. This is a boolean flag that determines the initial enablement of the feature.
- *Required features*. Some features may be built on top of others; for example, a HMIPv6 protocol implementation relies on MIPv6, which in turn relies on IPv6. Thus, HMIPv6 can only be enabled if MIPv6 and IPv6 are enabled as well. This is a space-separated list of feature IDs.
- *NED packages*. This is a space-separated list of NED package names that identify the code that implements the feature. When you disable the feature, NED types defined in those packages and their subpackages will be excluded; also, C++ code in the folders that correspond to the packages (i.e. in the same folders as excluded NED files) will also be excluded.
- *Extra C++ source folders*. If the feature contains C++ code that lives outside NED source folders (nontypical), those folders are listed here.

- *Compile options*, for example `-DWITH_IPv6`. When the feature is enabled, the compiler options listed here are either added to the compiler command line of all C++ files or they can be used to generate a header file containing all these defines so that header file can be included in all C++ files. A typical use of this field is defining symbols (`WITH_xxx`) that allows you to write conditional code that only compiles when a given feature is enabled. Currently only the `-D` option (*define symbol*) is supported here.
- *Linker options*. When the feature is enabled, the linker options listed here are added to the linker command line. A typical use of this field is linking with additional libraries that the feature's code requires, for example `libavcodec`. Currently only the `-l` option (*link with library*) is supported here.

### 5.7.3. The Project Features Dialog

Features can be viewed, enabled and disabled on the *Project Features* page of the *Project Properties* dialog. The *Project | Project Features...* menu item is a direct shortcut to this property page.



**Figure 5.19. The Project Features page**

The central area of the dialog page lists the features defined for the project. Hovering the mouse over a list item will show the description and other fields of the feature in a tooltip window. Check an item to enable the feature, and uncheck to disable.

When you enable a feature that requires other features to work, the dialog will ask for permission to enable the required features as well. Also, if you disable a feature that others depend on, they will be disabled, too.

The *Apply*, *OK* and *Cancel* buttons work as expected. *Restore Defaults* restores the features to their initial state (see the *Initially enabled* attribute above).

Above the list there is a notification area in the dialog. If the IDE detects that your project's configuration is inconsistent with the feature enablements, it will display a warning there, and offer a way to automatically fix the problems. Fixing means that the IDE will adjust the project's NED and C++ settings to make them consistent with the feature enablements. Such check is also performed just before build.

#### 5.7.4. What Happens When You Enable/Disable a Feature

When you enable or disable a feature on the *Project Features* page, a number of project settings will be modified:

- NED package exclusions. This corresponds to the contents of the *Excluded package subtrees* list on the *NED Source Folders* property page. When a feature is disabled, its NED packages will be excluded (added to the list), and vice versa.
- C++ folder exclusions. This can be viewed/edited on the *Makemake* property page, and also on the *Source Location* tab of the *C/C++ General > Paths and Symbols* property page.
- Compile options. For example, if the feature defines preprocessor symbols (-DWITH\_XXX), they can be used to generate a header file that contains the enabled macro definitions and that file can be included in all C++ files.
- Linker options. For example, if the feature defines additional libraries to link with, they will be displayed on the *Libraries* tab of the *C/C++ General > Paths and Symbols* property page.



Feature enablements are saved to the `.oppfeaturestate` file in the project root.

#### 5.7.5. Using Features from Command Line

*Project Features* can be easily configured from the IDE, but command line tools (`opp_makemake`, etc.) can also use them with the help of the `opp_featuretool` command.

If you want to build the project from the command line with the same feature combination the IDE is using, you need to generate the makefiles with the same `opp_makemake` options that the IDE uses in that feature combination. The `opp_featuretool makemakeargs` command (executed in the project's root directory) will show all the required arguments that you need to specify for the `opp_makemake` command to build the same output as the IDE. This allows you to keep the same features enabled no matter how you build your project.

Alternatively, you can choose *Export* on the *Makemake* page, and copy/paste the options from the generated `makemakefiles` file. This method is not recommended, because you must redo it manually each time after changing the enablement state of a feature.

#### 5.7.6. The `.oppfeatures` File

Project features are defined in the `.oppfeatures` file in your project's root directory. This is an XML file, and it currently has to be written by hand (there is no specialized editor for it).

The root element is `<features>`, and it may have several `<feature>` child elements, each defining a project feature. Attributes of the `<features>` element define the root(s) of the source folder(s) (`cppSourceRoots`) and the name of a generated header file that contains all defines specified by the `compilerFlags` attribute in the enabled features. The fields of a feature are represented with XML attributes; attribute names are `id`, `name`, `description`, `initiallyEnabled`, `requires`, `labels`, `nedPackages`, `extraSourceFolders`, `compileFlags` and `linkerFlags`. Items within attributes that represent lists (`requires`, `labels`, etc.) are separated by spaces.

Here is an example feature from the INET Framework:

```
<features cppSourceRoots="src" definesFile="src/inet/features.h">
    <feature
        id="TCP_common"
        name="TCP Common"
        description = "The common part of TCP implementations"
        initiallyEnabled = "true"
        requires = ""
        labels = ""
        nedPackages = "
            inet.transport.tcp_common
            inet.applications.tcpapp
            inet.util.headerserializers.tcp
        "
        extraSourceFolders = ""
        compileFlags = "-DWITH_TCP_COMMON"
        linkerFlags = ""
    />
```

### 5.7.7. How to Introduce a Project Feature

If you plan to introduce a project feature in your project, here's what you'll need to do:

- Isolate the code that implements the feature into a separate source directory (or several directories). This is because only whole folders can be declared as part of a feature, individual source files cannot.
- Check the remainder of the project. If you find source lines that reference code from the new feature, use conditional compilation (#ifdef WITH\_YOURFEATURE) to make sure that code compiles (and either works sensibly or throws an error) when the new feature is disabled. (Your feature should define the WITH\_YOURFEATURE symbol, i.e. -DWITH\_YOURFEATURE will need to be added to the feature compile flags.)
- Add the feature description into the .oppfeatures file of your project including the required feature dependencies.
- Test. At the minimum, test that your project compiles at all, both with the new feature enabled and disabled. More thorough, automated tests can be built using opp\_featuretool.

## 5.8. Project Files

Eclipse, CDT and the OMNeT++ IDE uses several files in the project to store settings. These files are located in the project root directory, and they are normally hidden by the IDE in the *Project Explorer View*. The files include:

- `.project` : Eclipse stores the general project properties in this file including project name, dependencies from other projects, and project type (i.e. whether OMNeT++-specific features are supported or this is only a generic Eclipse project).
- `.cproject` : This file contains settings specific to C++ development, including the build configurations; and per-configuration settings such as source folder locations and exclusions, include paths, linker paths, symbols; the build command, error parsers, debugger settings and so on.
- `.oppbuildspec` : Contains settings specific to OMNeT++. This file stores per-folder makefile generation settings that can be configured on the *Makemake* page of the *Project Properties* dialog.
- `.oppfeatures` : Optionally contains the definitions of project features.

- `.oppfeaturestate` : Optionally contains the current enablement state of the features. (We do not recommend keeping this file under version control.)
- `.nedfolders` : Contains the names of NED source folders; this is the information that can be configured on the *NED Source Folders* page of the *Project Properties* dialog.
- `.nedexclusions` : Contains the names of excluded NED packages.

If you are creating a project where no C++ support is needed (i.e. you are using an existing precompiled simulation library and you edit only NED and Ini files), the `.cproject` and `.oppbuildspec` files will not be present in your project.

---

# Chapter 6. Launching and Debugging

## 6.1. Introduction

The OMNeT++ IDE lets you execute single simulations and simulation batches, and also to debug and, to some extent, profile simulations. You can choose whether you want the simulation to run in graphical mode (using *QtEnv*) or in console (using *CmdEnv*); which simulation configuration and run number to execute; whether to record an eventlog or not; and many other options.

When running simulation batches, you can specify the number of processes allowed to run in parallel, so you can take advantage of multiple processors or processor cores. The progress of the batch can be monitored, and you can also kill processes from the batch if needed. Batches are based on the *parameter study* feature of INI files; you can read more about it in the *OMNeT++ Simulation Manual*.

Debugging support comes from the Eclipse C/C++ Development Toolkit (CDT), and beyond the basics (single-stepping, stack trace, breakpoints, watches, etc.) it also offers you several conveniences and advanced functionality such as inspection tooltips, conditional breakpoints and so on. Debugging with CDT also has extensive literature on the Internet. Currently CDT uses the GNU Debugger (gdb) as the underlying debugger.

Profiling support is based on the *valgrind* program, <http://valgrind.org>. Valgrind is a suite of tools for debugging and profiling on Linux. It can automatically detect various memory access and memory management bugs, and perform detailed profiling of your program. Valgrind support is brought into the OMNeT++ IDE by the Linux Tools Project of Eclipse.

## 6.2. Launch Configurations

Eclipse, and thus the IDE as well, uses *launch configurations* to store particulars of the program to be launched: what program to run, the list of arguments and environment variables, and other options. Eclipse and its C/C++ Development Toolkit (CDT) already comes with several types of launch configurations (e.g. "C/C++ Application"), and the IDE adds *OMNeT++ Simulation*. The same launch configuration can be used with the *Run*, *Debug* and *Profile* buttons alike.

## 6.3. Running a Simulation

### 6.3.1. Quick Run

The easiest way to launch a simulation is by selecting a project, folder, ini or NED file in *Project Explorer*, and clicking the *Run* button on the toolbar. This will create a suitable launch configuration (possibly after asking a few questions to clarify what you want to run) if one does not exist already. Instead of the *Run* button, you can also choose the *Run As... | OMNeT++ Simulation* from the item's context menu.

The details:

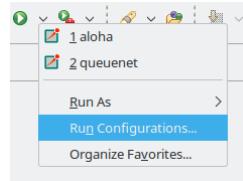
- If a folder is selected and it contains a single INI file, the IDE will use this file to start the simulation.
- If an INI file is selected, it will be used during the launch as the main INI file for the simulation.

- If a NED file is selected which contains a network definition, the IDE will scan for INI files in the active projects and will try to find a configuration that allows this network to start.

### 6.3.2. The Run Configurations Dialog

Launch configurations can be managed in the *Run Configurations* dialog. (Two further dialogs, *Debug Configurations* and *Profile Configurations* are very similar, and allow you to manage debugging/profiling aspects of launch configurations).

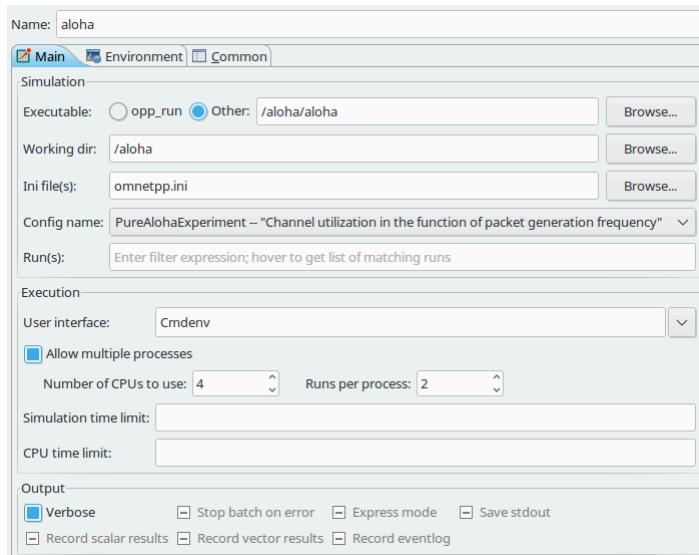
The *Run Configurations* can be opened in various ways: via the main menu (*Run | Run Configurations...*); via the context menu item of a project, folder or file (*Run As | Run Configurations...*); via the green *Run* toolbar button (*Run Configurations...* item of its attached menu, or by **Ctrl**-clicking any other menu item or the toolbar itself).



**Figure 6.1.** One way of opening the *Run Configurations* dialog

### 6.3.3. Creating a Launch Configuration

OMNeT++ IDE adds a new Eclipse launch configuration type, *OMNeT++ Simulation*, that supports launching simulation executables. To create a new run configuration, open the *Run Configurations...* dialog. In the dialog, select *OMNeT++ Simulation* from the tree, and click the *New launch configuration* icon in the top-left corner. A blank launch configuration is created; you can give it a name at the top of the form that appears.



**Figure 6.2.** The Simulation Launcher

The *Main* tab of the configuration dialog was designed to make the launching of simulations as easy as possible. The only required field is *Working directory*; all others have defaults. If you only select the working directory and the simulation program, it will start the first configuration from the `omnetpp.ini` file in the specified working directory.



Hover your mouse above the controls in this dialog and you will receive tooltip help for the selected control.

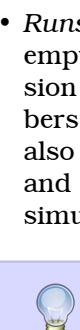


The *Launch* dialog will try to figure out your initial settings automatically. If you select an INI file in the *Project Explorer View*, or the active editor contains an INI file before launching the *Run* dialog, the INI file and working directory field will be automatically populated for you. The dialog will try to guess the executable name based on the settings of your current open projects.

- **Executable:** You must set the name of the simulation executable here. This is a workspace path. You may use the *Browse...* button to select the executable directly. If your project output is a shared library, select *opp\_run*; it will cause the IDE to use the *opp\_run* or the *opp\_run\_dbg* helper executable with the *-l* option to run the simulation. Make sure that the *Dynamic Libraries* field in the advanced section contains the libraries you want to load.
- **Working directory:** Specifies the working directory of the simulation program. This is a workspace path. Note that values in several other fields in the dialog are treated as relative to this directory, so changing the working directory may invalidate or change the meaning of previously selected entries in other fields of the dialog.
- **Initialization file(s):** You should specify one or more INI files that will be used to launch the simulation. The default is *omnetpp.ini*. Specifying more than one file (separated by space) will cause the simulation to load all those files in the specified order.
- **Config name:** Once you specify a legal INI file, the box will present all of the Config sections in that file. In addition, it will display the description of that section and the information regarding which Config section is extended by this section. You may select which Configuration should be launched.



The working directory and the INI file must contain valid entries before trying to set this option.



If the executable name and the INI file were already selected, hover the mouse above the field to get the list of matching runs.

- **User interface:** You can specify which UI environment should be used during execution. The dialog offers *Cmdenv* (command-line UI), *Qtenv* (Qt-based GUI), and *Tkenv* (legacy, Tk-based GUI). If you have a custom user interface, its name can also be specified here. Make sure that the code of the chosen UI library is available (linked into the executable/library or loaded dynamically).



Batch execution and progress feedback during simulation are only supported when using *Cmdenv*.

- **Allow multiple processes:** With batch execution, it is possible to tell the launcher to keep two or more simulations running at a time or to start a new simulation process after a certain number of runs executed. This way you can take advantage of multiple CPUs or CPU cores. You can set the number of CPUs to use and the number of runs to execute in a single process.



Use this option only if your simulation is CPU-limited and you have enough physical RAM to support all of the processes at the same time. Do not set it higher than the number of physical processors or cores you have in your machine.

- **Simulation time limit** and **CPU time limit** can be set also to limit the runtime length if the simulation from the launch dialog in case those were not set from the INI file.
- **Output options:** Various options can be set regarding simulation output. These checkboxes may be in one of three states: checked (on), unchecked (off), and grayed (unspecified). When a checkbox is the grayed state, the launcher lets the corresponding configuration option from the INI file to take effect.
- Clicking on the *More >>>* link will reveal additional controls.
- **Dynamic libraries:** A simulation may load additional DLLs or shared libraries before execution or your entire simulation may be built as a shared library. The *Browse* button is available to select one or more files (use **Ctrl** + click for multiple selection). This option can be used to load simulation code (i.e. simple modules), user interface libraries, or other extension libraries (scheduler, output file managers, etc.). The special macro  `${opp_shared_libs: /workingdir}` expands to all shared libraries provided by the current project or any other project on which you currently depend.



If your simulation is built as a shared library, you must use the `opp_run` stub executable to start it. `opp_run` is basically an empty OMNeT++ executable which understands all command line options, but does not contain any simulation code.



If you use external shared libraries (i.e. libraries other than the ones provided by the current open projects or OMNeT++ itself), you must ensure that the executable part has access to the shared library. On Windows, you must set the `PATH`, while on Linux and Mac you must set the `LD_LIBRARY_PATH` to point to the directory where the DLLs or shared libraries are located. You can set these variables either globally or in the *Environment* tab in the *Launcher Configuration Dialog*.

- **NED Source Path:** The directory or directories where the NED files are read from.



The variable  `${opp_ned_path: /workingdir}` refers to an automatically computed path (derived from project settings). If you want to add additional NED folders to the automatically calculated list, use the  `${opp_ned_path: /workingdir}: /my/ additional/path` syntax.

- **Image path:** A path that is used to load images and icons in the model.
- **Additional arguments:** Other command line arguments can be specified here and will be passed to the simulation process.
- **Build before launch:** This section allows you to configure the behavior of automatic build before launching. Build scope can be set either to build *this project and all its dependencies*, *this project only* or we can turn off autobuild before launch. Active configuration switching on build can be also configured here (*Ask*, *Switch*, *Never switch*.)

## Related Command-Line Arguments

Most settings in the dialog simply translate to command-line options to the simulation executable. This is summarized in the following list:

- Initialization files: maps to multiple `-f <inifile>` options
- Configuration name: adds a `-c <configname>` option
- Run number: adds a `-r <runnumber/filter>` option
- User interface: adds a `-u <userinterface>` option
- Dynamically loaded libraries: maps to multiple `-l <library>` options
- NED Source Path : adds a `-n <nedpath>` option

### 6.3.4. Debug vs. Release Launch

The launcher automatically decides whether the release or debug build of the model should be started. When running, release mode binaries are used automatically. For debugging, debug builds are started (i.e. those where the binary ends with `_dbg` suffix.) Before starting the simulation, the launcher ensures that the binary is up to date and triggers a build process (and changes also the active configuration) if necessary.

## 6.4. Batch Execution

OMNeT++ INI files allow you to run a simulation several times with different parameters. You can specify loops or constraint conditions for specific parameters.

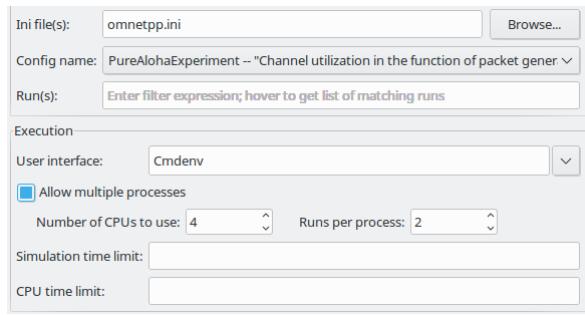
```
[Config PureAlohaExperiment]
description = "Channel utilization in the function of packet generation frequency"
repeat = 2
sim-time-limit = 300min
**.vector-recording = false
Aloha.host[*].iaTime = exponential(${mean=1,1.5,2,3,4,5..21 step 2}s)
```

**Figure 6.3. Iteration variable in the INI file**

 Batch running is supported only in the command line environment.

If you create an INI file configuration ([Config] section) with one or more iteration variables, you will be able to run your simulations to explore the parameter space defined by those variables. Practically, the IDE creates the Cartesian product from these variables and assigns a run number to each product. It is possible to execute one, several or all runs of the simulation by specifying the *Run number* field in the *Run Dialog*. You can specify a single number (e.g. 3), a combination of several numbers (e.g. 2,3,6,7..11), all run numbers (using \*) or boolean expressions using constants and iteration variables (e.g. `$numHosts>5 and $numHosts<10.`)

 If you already have specified your executable, chosen the configuration which should be run and selected the command line environment, you may try to hover over the *Run Number* field. This will give you a description of the possible runs and how they are associated with the iteration variable values (the tooltip is calculated by executing the simulation program with the `-x Configuration -G` options in command line mode).



**Figure 6.4. Iteration loop expansion in a tooltip**

If you have a multi-core or multi-processor system and have ample memory, you may try to set the *Processes to run parallel* field to a higher number. This will allow the IDE to start more simulation processes in parallel, resulting in a much lower overall simulation time for the whole batch.



Be aware that you need enough memory to run all these processes in parallel. We recommend using this feature only if your simulation is CPU-bound. If you do not have enough memory, your operating system may start to use virtual memory, dramatically decreasing the overall performance.

## 6.5. Debugging a Simulation

The OMNeT++ IDE integrates with the CDT (C/C++ Development Tooling) of Eclipse which also includes debugging support. The CDT debugger UI relies on *gdb* for doing the actual work.

### 6.5.1. Starting a Debug Session

Launching a simulation in debug mode is very similar to running it (see previous sections), only you have to select the *Debug* toolbar icon or menu item instead on *Run*. The same launch configurations are used for debugging that for running, that is, if you open the *Debug Configurations...* dialog, you will see the same launch configurations as in the *Run* dialog. The launcher is automatically using the debug build of the model (i.e. the executable that has a `_dbg` suffix.) The dialog will have extra tab pages where you can configure the debugger and other details.



If you have problems with starting the debug session, check whether:

- your executable is built with debug information,
- you can run the same executable without problem (using the same launch configuration, but with adding a `_dbg` suffix to the executable name), and
- the debugger type is set properly on the *Debugger* tab of the *Launch* dialog.



Batch (and parallel) execution is not possible in this launch type, so you may specify only a single run number.

### 6.5.2. Using the Debugger

The CDT debugger provides functionality that can be expected from a good C/C++ debugger: run control (run, suspend, step into, step over, return from function, drop to stack frame); breakpoints (also conditional and counting breakpoints); watchpoints

(a.k.a. expression breakpoints, breakpoints that stop the execution whenever the value of a given expression changes); watching and inspecting variables; and access to machine-level details such as disassembly, registers and memory.

Source code is shown in the editor area; additional information and controls are displayed in various Views: *Debug*, *Breakpoints*, *Expressions*, *Variables*, *Registers*, *Memory*, etc.

CDT's conversation with gdb can also be viewed, in the appropriate pages of the *Console View*. (Click the *Display Selected Console* icon, and choose *gdb* or *gdb traces* from the menu.)



One little tip that we found useful: if you have a pointer in the program that actually points to an array (of objects, etc), you can have it displayed as an array, too. In *Variables*, right-click the variable and choose *Display As Array...* from the menu. You will be prompted for a start index and the number of elements to display.

More information on the debugger is available in the CDT documentation, which is part of the IDE's Help system. See *C/C++ Development User Guide*, chapter *Running and debugging projects*.

### 6.5.3. Pretty Printers

Many programs contain data structures whose contents is difficult to comprehend by looking at "raw" variables in the program. One example is the `std::map<T>` class, which is essentially a dictionary but implemented with a binary tree, so it is practically impossible to figure out with a C++ debugger what data a concrete map instance contains.

The solution gdb offers to this problem is pretty printers. Pretty printers are Python classes that gdb invokes to transform some actual data structure to something that is easier for humans to understand. The `*.py` files that provide and register these pretty printers are usually loaded via gdb's startup script, `.gdbinit` (or some `.gdbinit.py` script, because gdb allows startup scripts to be written in Python, too).

The OMNeT++ IDE comes with pretty printers for container classes in the standard C++ library (`std::map<T>`, `std::vector<T>`, etc.) and also for certain OMNeT++ data types, for example `simevent_t`. These scripts are located under `misc/gdb/` in the OMNeT++ root directory. The IDE also supports project-specific pretty printers: if the debugged project's root folder contains a `.gdbinit.py` file, it will be loaded by gdb. (The project's `.gdbinit.py` can then load further Python scripts, e.g. from an `etc/gdb/` folder of the project.)

Pretty printer loading works in the following way. The IDE invokes gdb with `misc/gdb/.gdbinit.py` as startup script (for new launch configurations, the *GDB command file* field on the *Debugger* tab is set to  `${opp_root}/misc/gdb/gdbinit.py`). This script loads the pretty printers under `misc/gdb`, and also the project-specific pretty printers.



If you want to write your own pretty printers, refer to the gdb documentation. It is available online e.g. here: <http://sourceware.org/gdb/current/onlinedocs/gdb/>

Some pretty-printers may occasionally interfere with the debugged program (especially if the program's state is already corrupted by earlier errors), so at times it may be useful to temporarily turn off pretty printers. To prevent pretty printers from being loaded for a session, clear the *GDB command file* setting in the launch configuration. To disable them for a currently active debug session, switch to the *gdb* page in the *Console*, and enter the following gdb command:

```
disable pretty-printer global
```

Or, to only disable OMNeT++-specific pretty printers (but leave the standard C++ library printers on):

```
disable pretty-printer global omnetpp;.*
```

## 6.6. Just-in-Time Debugging

The OMNeT++ runtime has the ability to launch an external debugger and have it attached to the simulation process. One can configure a simulation to launch the debugger immediately on startup, or when an error (runtime error or crash) occurs. This just-in-time debugging facility was primarily intended for use on Linux.

To turn on just-in-time debugging, set the `debugger-attach-on-startup` or `debugger-attach-on-error` configuration option to `true`. You can do so by e.g. adding the appropriate line to `omnetpp.ini`, or specifying `--debugger-attach-on-startup=true` in the *Additional arguments* field in the launch configuration dialog. It is also possible to configure the debugger command line.



On some systems (e.g. Ubuntu), just-in-time debugging requires extra setup beyond installing external debugger. See the *Install Guide* for more details.

## 6.7. Profiling a Simulation on Linux

On Linux systems, the OMNeT++ IDE supports executing your simulation using the `valgrind` program. Running your program with `valgrind` allows you to find memory-related issues and programming errors in your code. The simulation will run in an emulated environment (much slower than normal execution speeds), but `valgrind` will generate a detailed report when it finishes. The report is shown in a separate *Valgrind View* at the end of the simulation run. The OMNeT++ IDE contains support only for the `memcheck` tool. If you want to use other tools (`cachegrind`, `callgrind`, `massif` etc.), you may try to install the full 'Linux Tools Project' from the Eclipse Marketplace.

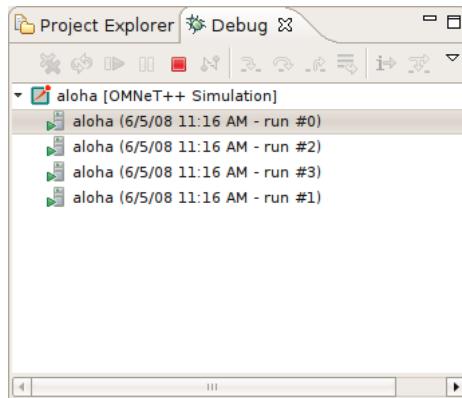
To start profiling, right-click on your project in the *Project Explorer* tree and select *Profile As... | OMNeT++ Simulation*. Valgrind must already be installed on your system.



Simulation executes considerably slower than a normal run. Prepare for long runtimes or limit the simulation time in your .INI file. You do not need statistical convergence here, just run long enough that all the code paths are executed in your model.

## 6.8. Controlling the Execution and Progress Reporting

After starting a simulation process or simulation batch you can keep track of the started processes in the *Debug View*. To open the *Debug View* automatically during launch, check the *Show Debug View on Launch* in the run configuration dialog, or select *Window | Show View... | Other... | Debug | Debug*. Select a process and click the terminate button to stop a specific simulation run or use the context menu for more options to control the process execution.



**Figure 6.5. Debug View**

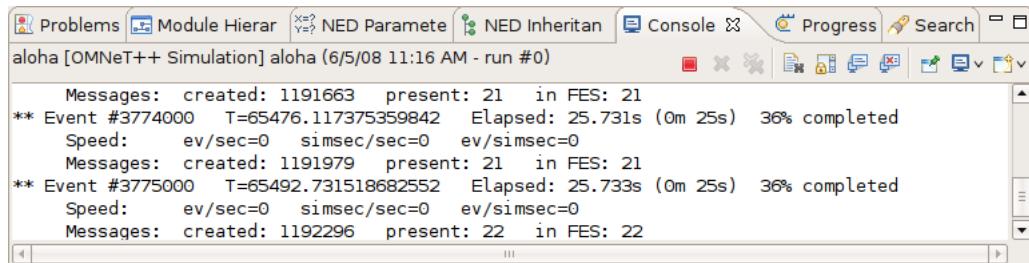


Place the Debug View in a different tab group than the console so you will be able to switch between the process outputs and see the process list at the same time.



You can terminate all currently running processes by selecting the root of the launch. This will not cancel the whole batch; only the currently active processes. If you want to cancel the whole batch, open the *Progress View* and cancel the simulation batch there.

Clicking on the process in the *Debug View* switches to the output of that process in the *Console View*. The process may ask for user input via the console, too. Switch to the appropriate console and enter the requested parameters.



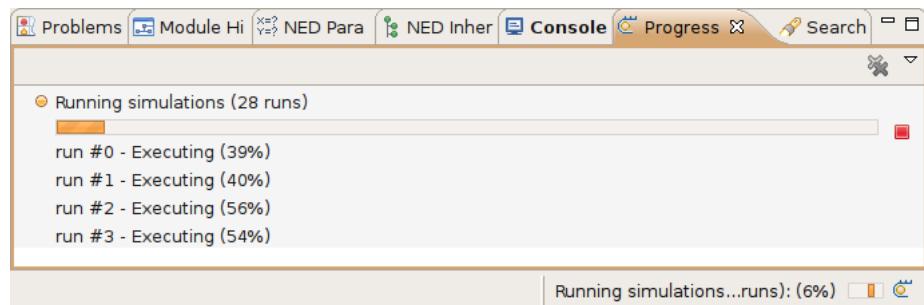
**Figure 6.6. Displaying the output of a simulation process in Console View**



By default, the *Console View* automatically activates when a process is writing to it. If you are running several parallel processes, this might be an annoying behavior and might prevent you from switching to the *Progress View*. You can switch off the auto-activation by disabling the *Show Console When Standard Out/Error Changes* in the *Console View* toolbar.

## Progress Reporting

If you have executed the simulation in the command line environment, you can monitor the progress of the simulation in the *Progress View*. See the status line for the overall progress indicator and click on it to open the detailed progress view. It is possible to terminate the whole batch by clicking on the cancel button in the *Progress View*.



**Figure 6.7. Progress report on four parallel processes**

 When *Progress View* displays "Waiting for user input", the simulation is waiting for the user. Switch to the appropriate console and provide the requested input for the simulation.

 If you need more frequent progress updates, set the `cmdenv-status-frequency` option in your INI file to a lower value.

---

# **Chapter 7. The Qtenv Graphical Runtime Environment**

## **7.1. Features**

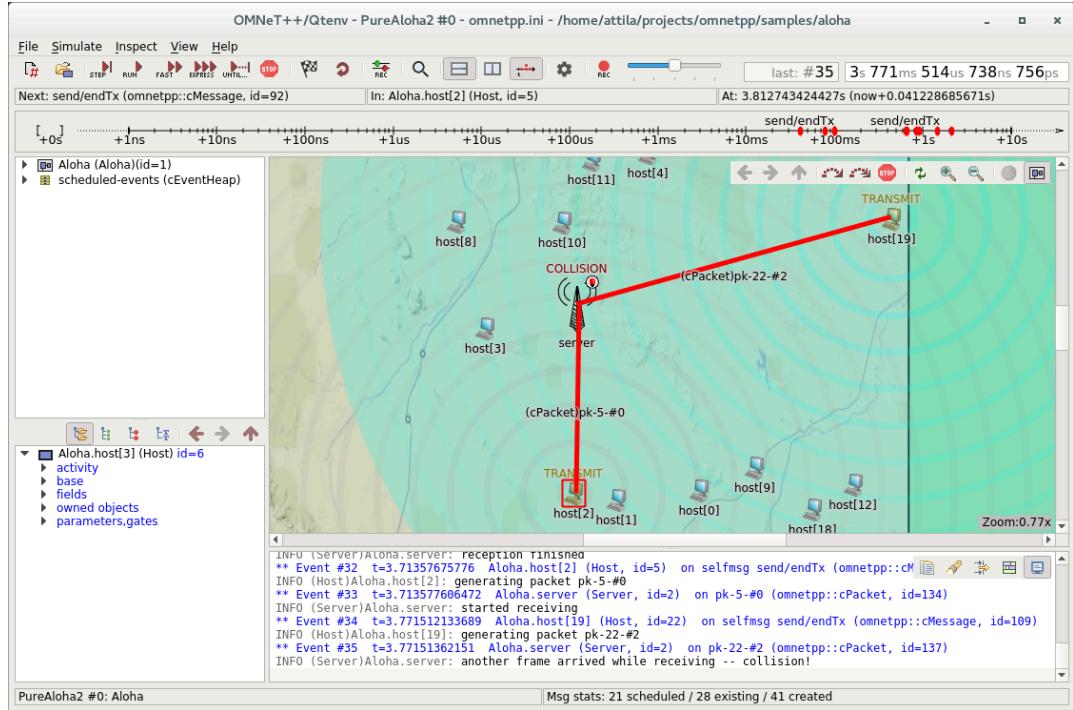
Qtenv is a graphical runtime interface for simulations. Qtenv supports interactive simulation execution, animation, inspection, tracing and debugging. In addition to model development and verification, Qtenv is also useful for presentation and educational purposes, since it allows the user to get a detailed picture of the state and history of the simulation at any point of its execution.

When used together with a C++ source-level debugger, Qtenv can significantly speed up model development.

Its most important features are:

- network visualization
- message flow animation
- various run modes: event-by-event, normal, fast, express
- run until (a scheduled event, any event in a module, or given simulation time)
- simulation can be restarted
- a different configuration/run or network can be set up
- log of message flow
- display of textual module logs
- inspectors for viewing contents of objects and variables in the model
- eventlog recording for later analysis
- capturing a video of the main window
- snapshots (detailed report about the model: objects, variables, etc.)

## 7.2. Overview of the User Interface



**Figure 7.1. The main window of Qtenv**

**NOTE** If you are experiencing graphics glitches, unreadable text, or the desktop color scheme you have set up is not suitable for Qtenv, you can disable the platform integration style plugins of Qt by setting the `QT_STYLE_OVERRIDE` environment variable to `fusion`. This will make the widgets appear in a platform-independent manner, as shown above.

The top of the window contains the following elements below the menu bar:

- **Toolbar:** The toolbar lets you access the most frequently used functions, such as stepping, running and stopping the simulation.
- **Animation speed:** The slider on the end of the toolbar lets you scale the speed of the built-in animations, as well as the playback speed of the custom animations added by the model.
- **Event Number and Simulation Time:** These two labels on the right end of the toolbar display the event number of the last executed or the next future event, and the current simulation time. The display format can be changed from context menu.
- **Top status bar:** Three labels in a row that display either information about the next simulation event (in Step and Normal mode), or performance data like the number of events processed per second (in Fast and Express mode). This can be hidden to free up vertical space.
- **Timeline:** Displays the contents of the Future Events Set (FES) on a logarithmic time scale. The timeline can be turned off to free up vertical space.
- **Bottom status bar:** Displays the current configuration, the run number, and the name of the root module (network) on the left, and a few statistics about the message objects in the model on the right.

The central area of the main window is divided into the following regions:

- *Object Navigator*: Displays the hierarchy of objects in the current simulation and in the FES.
- *Object Inspector*: Displays the contents and properties of the selected object.
- *Network Display*: Displays the network or any module graphically. This is also where animation takes place.
- *Log Viewer*: Displays the log of packets or messages sent between modules, or log messages output by modules during simulation.

Additionally, you can open inspector windows that float on top of the main window.

## 7.3. Using QtEnv

### 7.3.1. Starting QtEnv

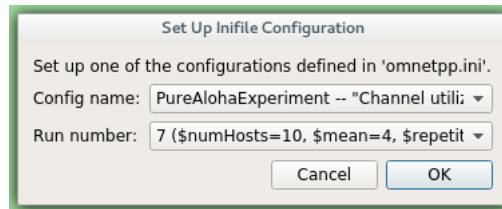
When you launch a simulation from the IDE, by default it will be started with QtEnv. When it does not, you can explicitly select QtEnv in the *Run* or *Debug* dialog.

QtEnv is also the default when you start the simulation from the command line. When necessary, you can force QtEnv by adding the `-u QtEnv` switch to the command line.

The complete list of command-line options, related environment variables and configuration options can be found at the end of this chapter.

### 7.3.2. Setting Up and Running the Simulation

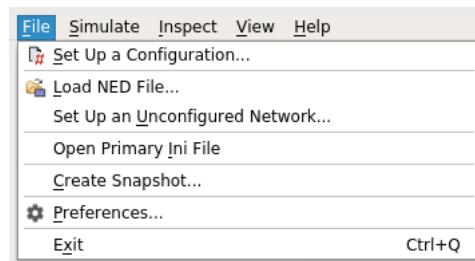
On startup, QtEnv reads the ini file(s) specified on the command line (or `omnetpp.ini` if none is specified), and automatically sets up the simulation described in them. If they contain several simulation configurations, QtEnv will ask you which one you want to set up.



**Figure 7.2. Setting Up a New Simulation**

Once a simulation has been set up (modules have been created and initialized), you can run it in various modes and examine its state. At any time you can restart the simulation, or set up another simulation. If you choose to quit QtEnv before the simulation finishes (or try to restart the simulation), QtEnv will ask you whether to finalize the simulation, which usually translates to saving summary statistics.

Functions related to setting up a simulation are in the *File* menu. Some of these functions are:



**Figure 7.3. The File menu**

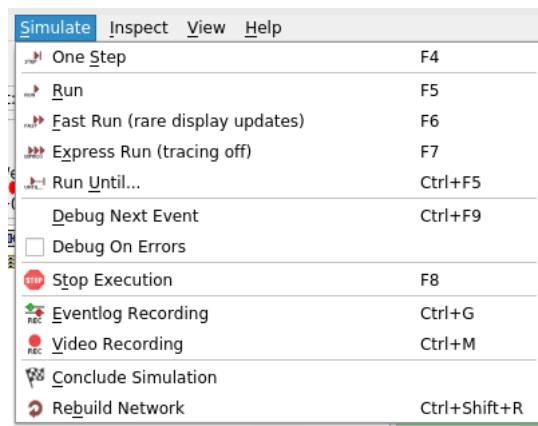
## Set up a Configuration

This function lets you choose a configuration and run number from the ini file.

## Open Primary Ini File

Opens the first ini file in an text window for viewing.

Simulation-related functions are in the *Simulate* menu, and are accessible via toolbar icons and keyboard shortcuts as well.



**Figure 7.4. The Simulate menu**

### Step

*Step* lets you execute one simulation event, that at the front of the FES. The next event is always shown on the status bar. The module where the next event will be delivered is highlighted with a red rectangle on the graphical display.

### Run (or Normal Run)

In *Run* mode, the simulation runs with all tracing aids on. Message animation is active, simulation time is interpolated if the model requested a non-zero animation speed, and inspector windows are constantly updated. 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 (e.g. you can open inspectors, etc.).



If you find this mode too slow or distracting, you may switch off animation features in the *Preferences* dialog.

### Fast Run

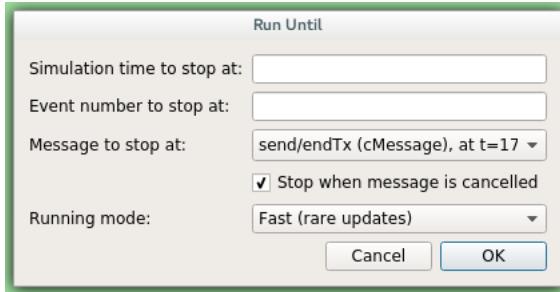
In *Fast* mode, message animation is turned off. The inspectors are updated much less often. Fast mode is several times faster than the Run mode; the speed can increase by up to 10 times (or up to the configured event count).

### Express Run

In *Express* mode, the simulation runs at about the same speed as with Cmdenv, all tracing disabled. Module log is not recorded. The simulation can only be interacted with once in a while, thus the run-time overhead of the user interface is minimal. UI updates can even be disabled completely, in which case you have to explicitly click the *Update now* button to refresh the inspectors.

## Run Until

You can run the simulation until a specified simulation time, event number or until a specific message has been delivered or canceled. This is a valuable tool during debugging sessions (select *Simulate | Run until...*). It is also possible to right-click on an event in the simulation timeline and choose the *Run until this event* menu item.



**Figure 7.5. The Run Until dialog**

## Run Until Next Event

It is also possible to run until an event occurs in a specified module. Browse for the module and choose *Run until next event in this module*. Simulation will stop once an event occurs in the selected module.

## Debug Next Event

This function is useful when you are running the simulation under a C++ source-level debugger. *Debug Next Event* will perform one simulation event just like *Step*, but executes a software debugger breakpoint (`int3` or `SIGTRAP`) just before entering the module's event handling code (`handleMessage()` or `activity()`). This will cause the debugger to stop the program there, letting you examine state variables, single-step, etc. When you resume execution, QtEnv will get back control and become responsive again.

## Debug On Errors

This menu item allows you to change the value of the debug-on-errors configuration variable on the fly. This is useful if you forgot to set this option before starting the simulation, but would like to debug a runtime error. The state of this menu item is reset to the value of debug-on-errors every time QtEnv is started.

## Recording an Event Log

The OMNeT++ simulation kernel allows you to record event related information into a file which later can be used to analyze the simulation run using the *Sequence Chart* tool in the IDE. Eventlog recording can be turned on with the `record-eventlog=true` ini file option, but also interactively, via the respective item in the *Simulate* menu, or using a toolbar button.

Note that the starting QtEnv with `record-eventlog=true` and turning on recording later does not result in exactly the same eventlog file. In the former case, all steps of setting up the network, such as module creations, are recorded as they happen; while for the latter, QtEnv has to "fake" a chain of steps that would result in the current state of the simulation.

## Capturing a Video

When active, this feature will save the contents of the main window into a subfolder named `frames` in the working directory with a regular frequency (in animation time).

Each frame is a PNG image, with a sequence number in its file name. Currently the user has to convert (encode) these images into a video file after the fact by using an external tool (for example `ffmpeg`, `avconv`, or `vlc`). When the recording is started, an info dialog pops up, showing further details on the output, and an example command for encoding in high quality using `ffmpeg`. The resulting video is also affected by the speed slider on the toolbar.



This built-in recording feature is able to produce a smooth video, in contrast to external screen-capture utilities. This is possible because it has access to more information, and has more control over the process than external tools.

### Conclude Simulation

This function finalizes the simulation by invoking the user-supplied `finish()` member functions on all module and channel objects in the simulation. The customary implementation of `finish()` is to record summary statistics. The simulation cannot be continued afterwards.

### Rebuild Network

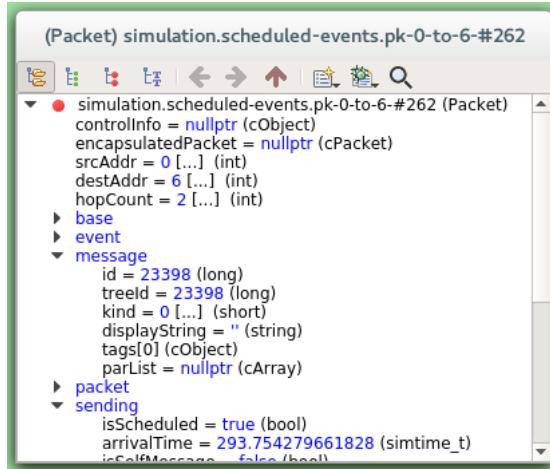
Rebuilds the simulation by deleting the current network and setting it up again. Improperly written simulations often crash when *Rebuild Network* is invoked; this is usually due to incorrectly written destructors in module classes.

### 7.3.3. Inspecting Simulation Objects

#### Inspectors

The *Network Display*, the *Log Viewer* and the *Object Inspector* in the main window share some common properties: they display various aspects (graphical view / log messages / fields or contents) of a given object. Such UI parts are called *inspectors* in Qtenv.

The three inspectors mentioned above are built into the main window, but you can open add additional ones at any time. The new inspectors will open in floating windows above the main window, and you can have any number of them open.



**Figure 7.6. A floating inspector window**

Inspectors come in many flavours. They can be graphical like the network view, textual like the log viewer, tree-based like the object inspector, or something entirely different.

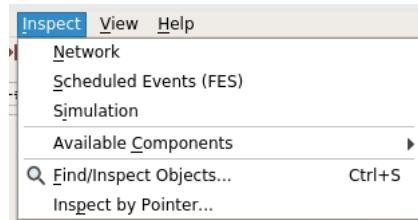


Some window managers might disable/hide the close button of floating inspectors. If this happens, you can still close them with a keyboard shortcut (most commonly

**Alt+F4**), or by right-clicking on the title bar, and choosing the Close option in the appearing menu.

## Opening Inspectors

Inspectors can be opened in various ways: by double-clicking an item in the *Object Navigator* or in other inspectors; by choosing one of the *Open...* menu items from the context menu of an object displayed on the UI; via the *Find/Inspect Objects* dialog (see later); or even by directly entering the C++ pointer of an object as a hex value. Inspector-related menu items are in the *Inspect* menu.



**Figure 7.7. The Inspect menu**

## History

Inspectors always show some aspect of one simulation object, but they can change objects. For example, in the *Network View*, when you double-click a submodule which is itself a compound module, the view will switch to showing the internals of that module; or, the *Object Inspector* will always show information about the object last clicked in the UI. Inspectors maintain a navigable history: the *Back/Forward* functions go to the object inspected before/after the currently displayed object. Objects that are deleted during simulation also disappear from the history.

## Restoring Inspectors

When you exit and then restart a simulation program, QtEnv tries to restore the open inspector windows. However, as object identity is not preserved across different runs of the same program, QtEnv uses the object full path, class name and object ID (where exists) to find and identify the object to be inspected.

Preferences such as zoom level or open/closed state of a tree node are usually maintained per object type (i.e. tied to the C++ class of the inspected object).

## Extending QtEnv

It is possible for the user to contribute new inspector types without modifying QtEnv code. For this, the inspector C++ code needs to include QtEnv header files and link with the QtEnv library. One caveat is that the QtEnv headers are not public API and thus subject to change in a new version of OMNeT++.

## 7.4. Using QtEnv with a Debugger

You can use QtEnv together with a C++ debugger, which is mainly useful when developing new models. When you do that, there are a few things you need to know.

QtEnv is a library that runs as part of the simulation program. This has a lot of implications, the most apparent being that when the simulation crashes (due to a bug in the model's C++ code), it will bring down the whole OS process, including the QtEnv GUI.

The second consequence is that suspending the simulation program in a debugger will also freeze the GUI until it is resumed. Also, QtEnv is single-threaded and runs in the same thread as the simulation program, so even if you only suspend the simulation's thread in the debugger, the UI will freeze.

The QtEnv UI deals with `cObjects` (the C++ methods that the GUI relies on are defined on `cObject`). All other data such as primitive variables, non-`cObject` classes and structs, STL containers etc, are hidden from QtEnv. You may wrap objects into `cObjects` to make them visible for QtEnv, that's what e.g. the `WATCH` macros do as well.

The following sections go into detail about various parts and functions of the QtEnv UI.

## 7.5. Parts of the QtEnv UI

### 7.5.1. The Status Bars

The status bars show the simulation's progress. There is one row at the top of the main window, and one at the bottom. The top one can be hidden using the *View | Status Details* menu item.

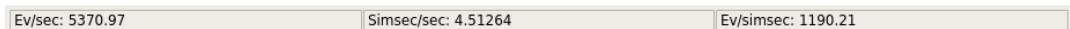


**Figure 7.8. The top status bar**

When the simulation is paused or runs with animation, the top row displays the next expected simulation event. Note the word *expected*: certain schedulers may insert new events before the displayed event in the last moment. Some schedulers that tend to do that are those that accept input from outside sources: real-time scheduler, hybrid or hardware-in-the-loop schedulers, parallel simulation schedulers, etc. Contents of the top row:

1. Name, C++ class and ID of the next message (event) object
2. The module where the next event will occur (i.e. the module where the message will be delivered)
3. The simulation time of the next (expected) simulation event
4. Time of the next event, and delta from the current simulation time

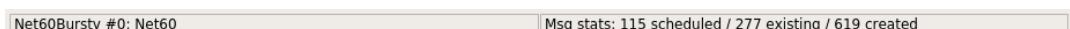
When the simulation is running in *Fast* or *Express* mode, displaying the next event becomes useless, so the contents of the top row are replaced by the following performance gauges:



**Figure 7.9. The top status bar during Fast or Express run**

1. Simulation speed: number of events processed per real second
2. Relative speed of the simulation (compared to real-time)
3. Event density: the number of events per simulated seconds

The bottom row contains the following items:



**Figure 7.10. The bottom status bar**

1. Ini config name, run number, and the name of the network
2. Message statistics: the number of messages currently scheduled (i.e. in the FES); the number of message objects that currently exists in the simulation; and the number of message objects that have been created this far, including the already deleted ones. Out of the three, probably the middle one is the most useful: if it is steadily

growing without apparent reason, the simulation model is probably missing some `delete msg` statements, and needs to be debugged.

### 7.5.2. The Timeline



**Figure 7.11. The timeline**

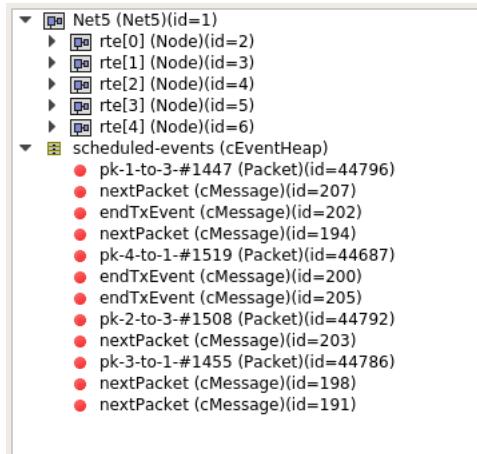
The timeline displays the contents of the Future Events Set on a logarithmic time scale. Each dot represents a message (event). Messages to be delivered in the current simulation time are grouped into a separate section on the left between brackets.

Clicking an event will focus it in the *Object Inspector*, and double-clicking will open a floating inspector window. Right-clicking will bring up a context menu with further actions.

The timeline is often crowded, limiting its usefulness. To overcome this, you can hide uninteresting events from the timeline: right-click the event, and choose *Exclude Messages Like 'x' From Animation* from the context menu. This will hide events with similar name and the same C++ class name from the timeline, and also skip the animation when such messages are sent from one module to another. You can view and edit the list of excluded messages on the *Filtering* page of the *Preferences* dialog. (Tip: the timeline context menu provides a shortcut to that dialog).

The whole timeline can be hidden (and revealed again) using the *View | Timeline* menu item, by pressing a button on the toolbar, or simply by dragging the handle of the separator under it all the way up.

### 7.5.3. The Object Navigator



**Figure 7.12. The object tree**

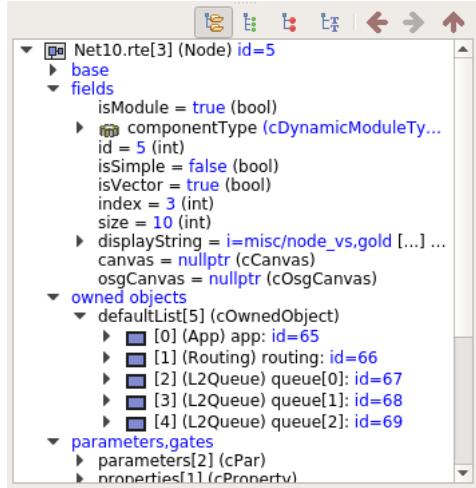
The *Object Navigator* displays inspectable objects reachable from two root objects (the network module and the FES) in a tree form.

Clicking an object will focus it in the *Object Inspector*, and double-clicking will open a floating inspector window. Right-clicking will bring up a context menu with further actions.

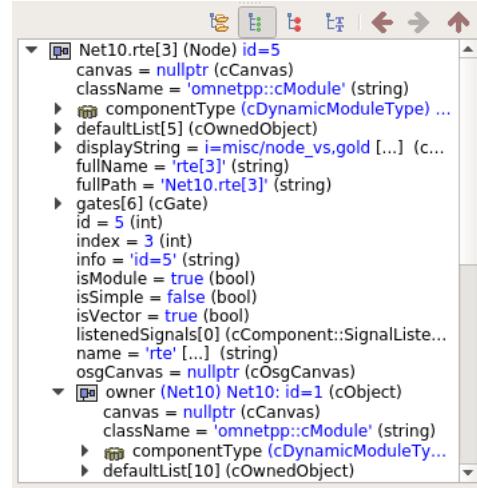
### 7.5.4. The Object Inspector

The *Object Inspector* is located below the *Object Navigator*, and lets you examine the contents of objects in detail. The *Object Inspector* always focuses on the object last

clicked (or otherwise selected) on the Qtenv UI. It can be directly navigated as well, via the *Back*, *Forward*, and *Go to Parent* buttons, and also by double-clicking objects shown inside the inspector's area.



**Figure 7.13. The object inspector in Grouped mode**

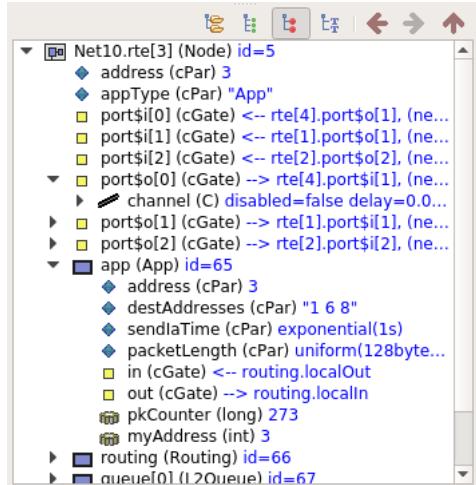


**Figure 7.14. The object inspector in Flat mode**

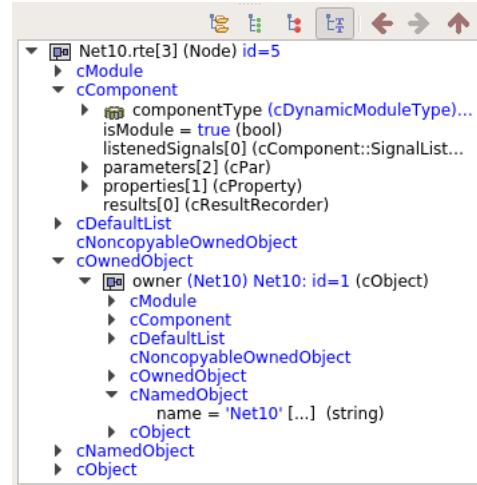
The inspector has four display modes: *Grouped*, *Flat*, *Children* and *Inheritance*. You can switch between these modes using the buttons on the inspector's toolbar.

In *Grouped*, *Flat* and *Inheritance* modes, the tree shows the fields (or data members) of the object. It uses meta-information generated by the message compiler to obtain the list of fields and their values. (This is true even for the built-in classes -- the simulation kernel contains their description of msg format.)

The only difference between these three modes is the way the fields are arranged. In *Grouped* mode, they are organized in categories, in *Flat* mode they form a simple alphabetical list, and in *Inheritance* mode they are organized based on which superclass they are inherited from.



**Figure 7.15. The object inspector in Children mode**



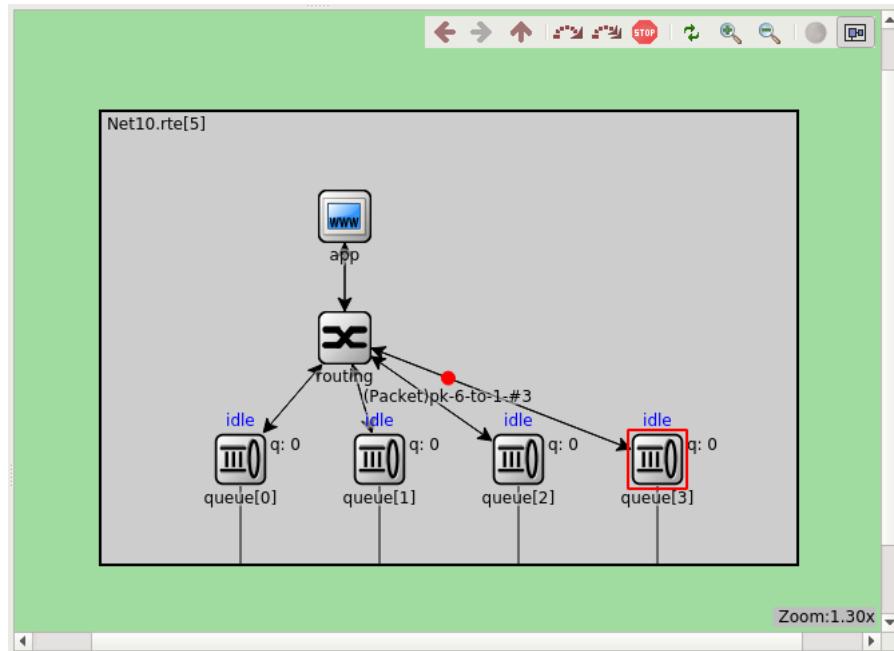
**Figure 7.16. The object inspector in Inheritance mode**

In *Children* mode, the tree shows the child objects of the currently inspected object. The child list is obtained via the `forEachChild()` method of the object. This is very similar to how the *Object Navigator* works, but this can have an arbitrary root.

### 7.5.5. The Network Display

The network view provides a graphical view of the network and in general, modules. Graphical representation is based on display strings (@display properties in the NED file). You can go into any compound module by double-clicking its icon.

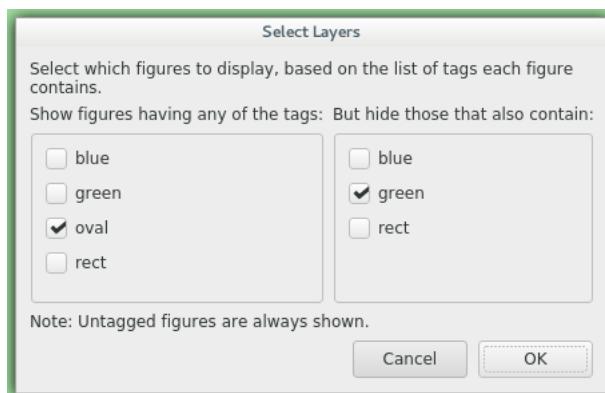
Message sending, method calls and certain other events are animated in the graphical view. You can customize animation in the *Animation* page of the *Preferences* dialog.



**Figure 7.17. The network display**

The built-in cCanvas of the inspected object is also rendered in this view together with the module contents to allow for overlaying custom annotations and animations. This canvas contains the figures declared by the @figure properties in the NED source of the module.

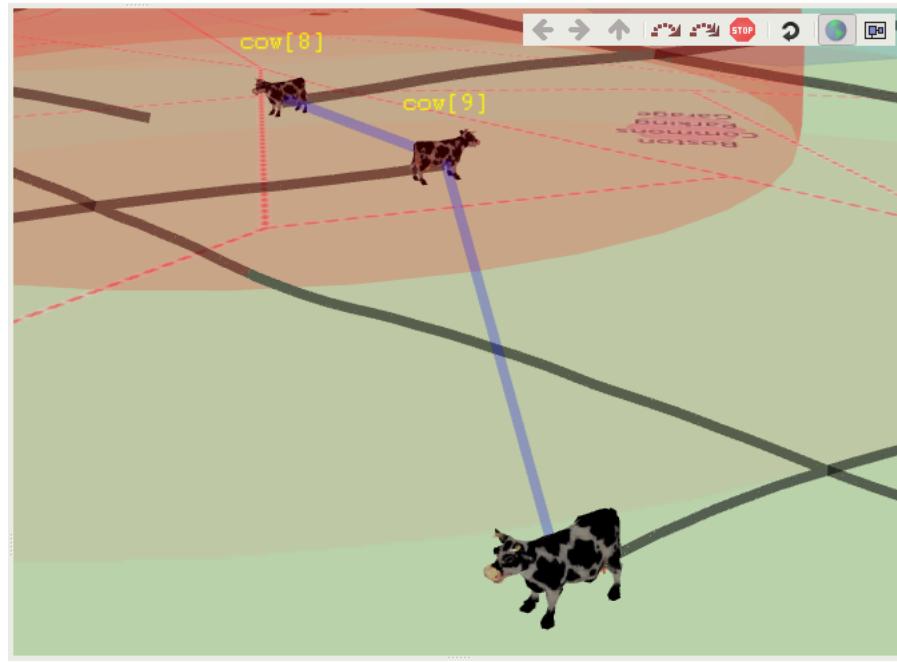
By choosing the *Show/Hide Canvas Layers* item in the context menu of the inspected module, the displayed figures can be filtered based on the tags set on them.



**Figure 7.18. Figure filtering dialog**

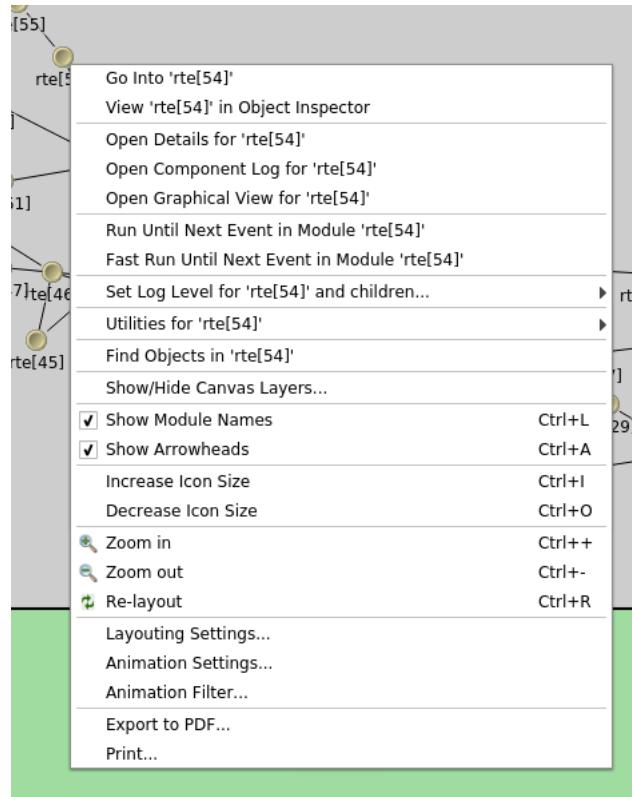
Since any figure can have any number of tags, a two-step filtering mechanism is applied to give sufficient control. The left side is a whitelist, while the right side is a blacklist. The example above would only let all the figures with the "oval" tag appear, except those that also have the "green" tag on them.

If the inspected module has a built-in `cOsgCanvas` (and QtEnv is built with OSG support enabled), this inspector can also be switched into a 3D display mode with the globe icon on its toolbar. In this case, the 2D network and canvas display is replaced by the scene encapsulated by the `cOsgCanvas`.

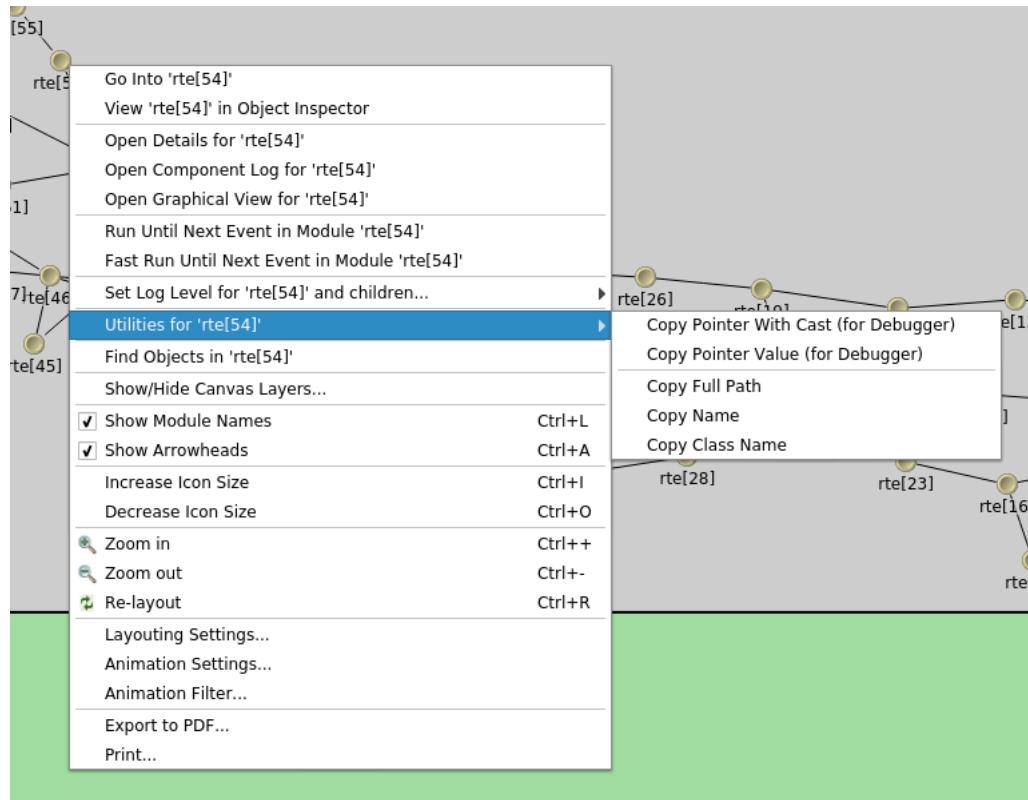


**Figure 7.19. The network display in 3D mode**

The context menu of submodules makes further actions available (see below).



**Figure 7.20. Submodule context menu**



**Figure 7.21. The Utilities submenu**

### Zooming and Panning

There are several ways for zooming the canvas, both using the mouse and the keyboard:

- To zoom in around a point, double-click the canvas; use **Shift** + double-click to zoom out, or scroll while holding down **Ctrl**.

You can also zoom around the center of the viewport with the looking glass buttons on the canvas toolbar.

- For marquee zoom, drag out a rectangle with the left mouse button while holding down **Ctrl**; you can cancel the operation with the right mouse button.
- Panning: moving the mouse while holding down the left mouse button will move the canvas; this is often a more comfortable way to navigate the canvas than using the scroll bars. You can of course scroll in any direction with simply the mouse wheel, or the similar functionality of many touchpads.

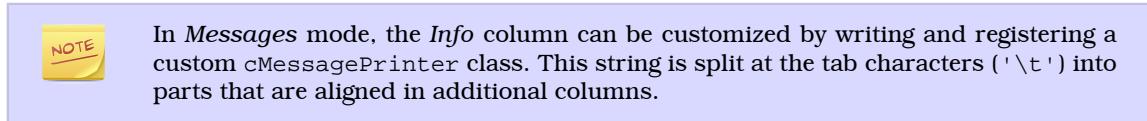
### 7.5.6. The Log Viewer

When you run the simulation, QtEnv will remember the output from logging statements (`EV << "Hello World\n";`) and the messages sent between modules, and can present it to you in a meaningful manner. Only the output from the last N events is preserved (N being configurable in the *Preferences* dialog), and only in Step, Run and Fast Run modes. (Express mode can be so fast because such overhead is turned off while it's active.)

The *Log Viewer* shows log related to one compound module and its subtree. It has two modes: *Messages* and *Log* mode, the default being *Messages*. You can switch between the two modes with tool icons on the inspector's local toolbar.

In *Messages* mode, the window displays messages sent between the (immediate) submodules of the inspected compound module, and messages sent out of, or into the

compound module. The embedded *Log Viewer* shows content related to the module inspected in the *Network Display* above it at any time. You can view details about any message in the *Object Inspector* by clicking on it, and access additional functions in its context menu.



Event#	Time	Relevant Hops	Name	Info
#14575	154.471255512101	queue[4] -->	pk-4->L2-1-#140	id=12079 kind=0 length=643 bytes
#14606	154.809168261136	--> queue[3]	pk-8-to-1-#165	id=12106 kind=0 length=143 bytes
#14608	154.80914321299	queue[3] --> routing	pk-8-to-1-#165	id=12106 kind=0 length=143 bytes
#14609	154.80914321299	routing --> queue[4]	pk-8-to-1-#165	id=12106 kind=0 length=143 bytes
#14610	154.80914321299	queue[4] -->	pk-8-to-1-#165	id=12106 kind=0 length=143 bytes
#14617	155.101191635202	--> queue[4]	pk-1-to-6-#160	id=12118 kind=0 length=290 bytes
#14619	155.104360993061	queue[4] --> routing	pk-1-to-6-#160	id=12118 kind=0 length=290 bytes
#14620	155.104360993061	routing --> queue[3]	pk-1-to-6-#160	id=12118 kind=0 length=290 bytes
#14621	155.104360993061	queue[3] -->	pk-1-to-6-#160	id=12118 kind=0 length=290 bytes
#14637	155.211324910699	app --> routing	pk-2-to-8-#147	id=12136 kind=0 length=787 bytes
#14638	155.211324910699	routing --> queue[3]	pk-2-to-8-#147	id=12136 kind=0 length=787 bytes
#14639	155.211324910699	queue[3] -->	pk-2-to-8-#147	id=12136 kind=0 length=787 bytes
#14771	156.186416651783	--> queue[4]	pk-1-to-6-#161	id=12244 kind=0 length=977 bytes
#14776	156.186416651783	queue[4] --> routing	pk-1-to-6-#161	id=12244 kind=0 length=977 bytes

Figure 7.22. The log viewer showing message traffic

In *Log* mode, the window displays log lines that belong to submodules under the inspected compound module (i.e. the whole module subtree.)

```

** Event #10072 t=171.913021524198 Net10.rte[2].queue[3] (L2Queue, id=63) on selfmsg printFormat (omnetpp)
INFO (L2Queue)Net10.rte[2].queue[3]: Transmission finished.
** Event #16155 t=172.901166916603 Net10.rte[2].app (App, id=58) on selfmsg nextPacket (omnetpp::cMessage)
INFO (App)Net10.rte[2].app: generating packet pk-2-to-8-#166
** Event #16156 t=172.901166916603 Net10.rte[2].routing (Routing, id=59) on pk-2-to-8-#166 (Packet, id=1
INFO (Routing)Net10.rte[2].routing: forwarding packet pk-2-to-8-#166 on gate index 3
** Event #16157 t=172.901166916603 Net10.rte[2].queue[3] (L2Queue, id=63) on pk-2-to-8-#166 (Packet, id=1
INFO (L2Queue)Net10.rte[2].queue[3]: Received (Packet)pk-2-to-8-#166
INFO (L2Queue)Net10.rte[2].queue[3]: Starting transmission of (Packet)pk-2-to-8-#166
** Event #16158 t=172.904080916603 Net10.rte[2].queue[3] (L2Queue, id=63) on selfmsg endTxEvent (omnetpp
INFO (L2Queue)Net10.rte[2].queue[3]: Transmission finished.
** Event #16183 t=173.173456328388 Net10.rte[2].app (App, id=58) on selfmsg nextPacket (omnetpp::cMessage
INFO (App)Net10.rte[2].app: generating packet pk-2-to-6-#167
** Event #16184 t=173.173456328388 Net10.rte[2].routing (Routing, id=59) on pk-2-to-6-#167 (Packet, id=1

```

Figure 7.23. The log viewer showing module log

You can filter the content of the window to only include messages from specific modules. Open the log window's context menu and select *Filter Window Contents*.

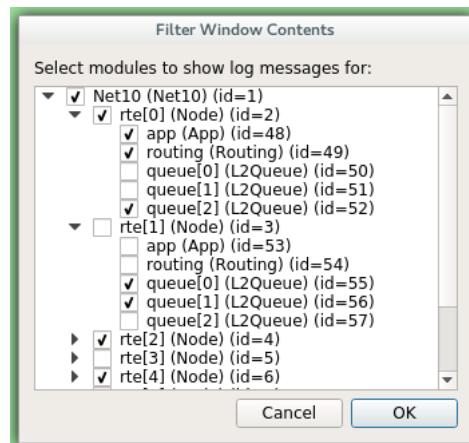


Figure 7.24. The log filter dialog

General logging behavior, such as the prefix format, can be controlled in the *Preferences* dialog. The log level of each module (and its descendants) can be set in its context menu.

It is also possible to open separate log windows for individual modules. A log window for a compound module displays the log from all of its submodule tree. To open a log window, find the module in the module tree or the network display, right-click it and choose *Open Component Log* from the context menu.

## 7.6. Inspecting Objects

### 7.6.1. Object Inspectors

In addition to the inspectors embedded in the main window, Qtenv also lets you open floating inspector windows for objects. The screenshot below shows Qtenv with several inspectors open.

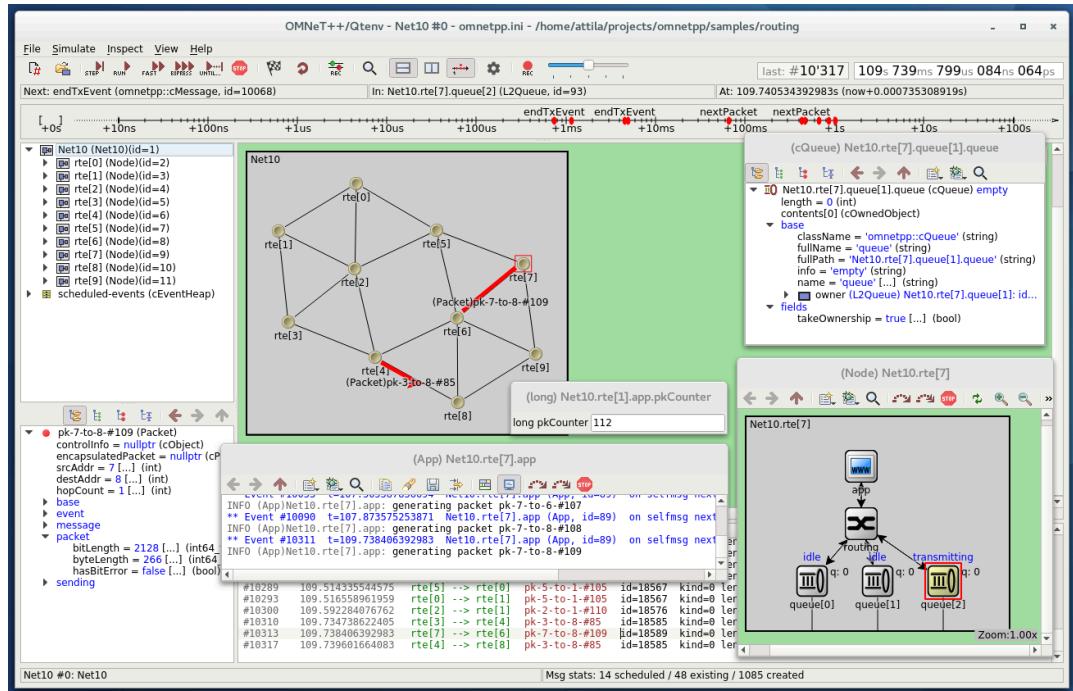


Figure 7.25. Qtenv with several floating inspectors open

### 7.6.2. Browsing the Registered Components

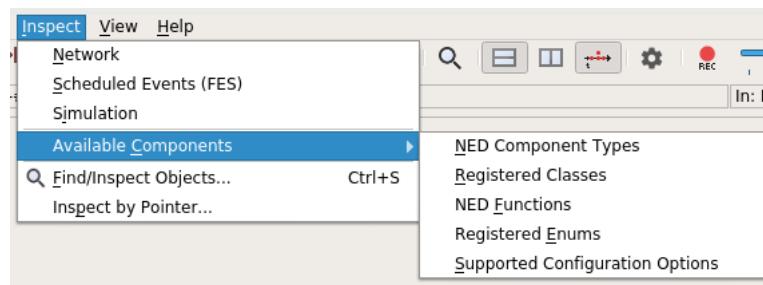


Figure 7.26. The Inspect menu

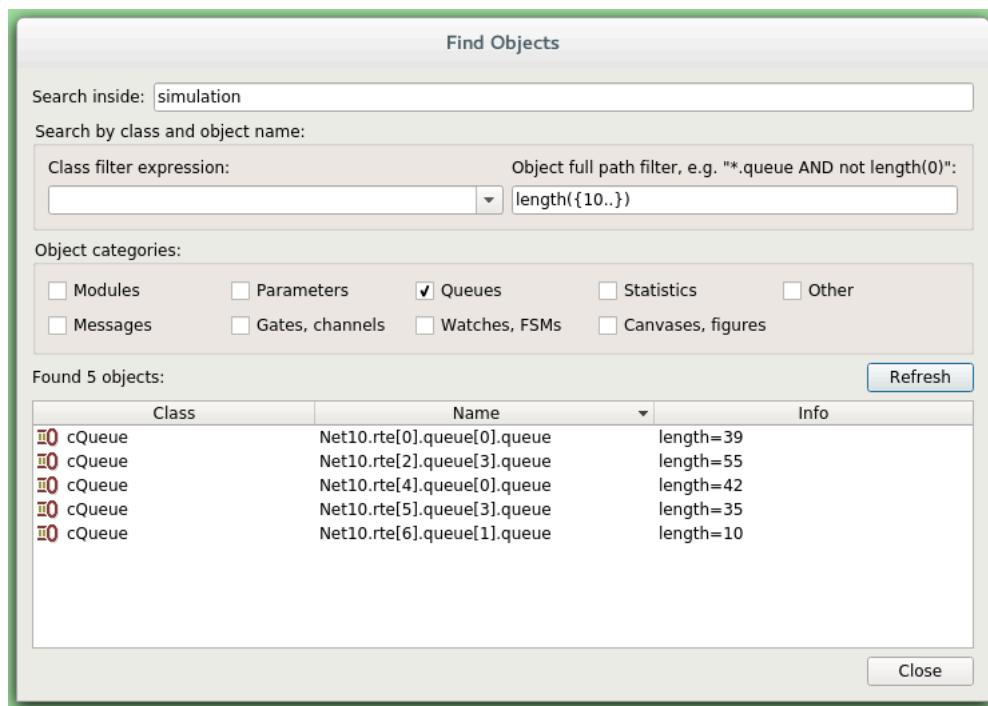
Registered components (NED Types, classes, functions, enums) can be displayed with the *Inspect | Available components* menu item. If an error message reports missing types or classes, you can check here whether the missing item is in fact available, i.e. registered correctly.

### 7.6.3. Querying Objects

The *Find/Inspect Objects* dialog lets you search the simulation for objects that meet certain criteria. The criteria may be the object name, class name, the value of a field of the object, or the combination of those. The results are presented in a table which you can sort by columns, and double-click items in it to inspect them.

Some possible use cases:

- Identifying bottlenecks in the network by looking at the list of all queues, and ordering them by length (i.e. have the result table sorted by the *Info* column)
- Finding nodes with the highest packet drop count. If the drop counts are watched variables (see `WATCH( )` macro), you can get a list of them.
- Finding modules that leak messages. If the live message count on the status bar keeps climbing up, you can issue a search for all message objects, and see where the leaked messages hide.
- Easy access for some data structures or objects, for example routing tables. You can search by name or class name, and use the result list as a collection of hotlinks, sparing you manual navigation in the simulation's object tree.



**Figure 7.27. Using the Find/Inspect Objects dialog for finding long queues**

The dialog lets you specify the search root, and the name and class name of the objects to find. The latter two accept wildcard patterns.

The checkboxes in the dialog can be used to select the object category that interests you. If you select a category, all objects with that type (and any type derived from it) will be included in the search. Alternatively, if you specify object class as a class filter expression, the search dialog will try to match the object's class name with the given string, meaning that objects of derived types will not be included.

You can also provide a generic filter expression, which matches the object's full path by default. Wildcards ("?", "\*") are allowed. "{a-exz}" matches any character in the

range "a" .. "e" plus "x" and "z". You can match numbers: "\* . job{128 .. 191}" will match objects named "job128", "job129", ..., "job191". "job{128 .. }" and "job{ .. 191}" are also understood. You can combine patterns with AND, OR and NOT and parentheses (lowercase and, or, not are also accepted). You can match other object fields such as queue length, message kind, etc., with the syntax "fieldname(pattern)". If the pattern contains parentheses or space, you need to enclose in quotes. (HINT: You will want to start the pattern with "\* ." in most cases to match objects anywhere in the network!).

**Examples:**

\* . subnet2 . \* . destAddr "destAddr" "subnet2"

- \* . destAddr : Matches all objects with the name "destAddr" (likely module parameters).
- \* . node[8 .. 10] . \* : Matches anything inside module node[8], node[9] and node[10] .
- className(cQueue) and not length(0) : Matches non-empty queue objects.
- className(cQueue) and length({10 .. }) : Matches queue objects with length >= 10.
- kind(3) or kind({7 .. 9}) : Matches messages with message kind equal to 3, 7, 8 or 9 (only messages have a "kind" attribute).
- className(IP\*) and \* . data-\* : Matches objects whose class name begins with "IP" and name begins with "data- ."
- not className(omnetpp::cMessage) and byteLength({1500 .. }) : Matches messages whose class is not cMessage and byteLength is at least 1500 (only messages have a "byteLength" attribute).
- "\* (\*" or "\* . msg(ACK)" : Quotation marks needed when pattern is a reserved word or contains parentheses (note: \*.msg(ACK) without parentheses would be interpreted as some object having a "\*.msg" attribute with the value "ACK"!).



Qtenv uses the `cObject::forEachChild` method to collect all objects from a tree recursively. If you have your own objects derived from `cObject`, you should redefine the `cObject::forEachChild` to function correctly with an object search.



The class names have to be fully qualified, that is, they should contain the namespace(s) they are in, regardless of the related setting in the *Preferences dialog*.

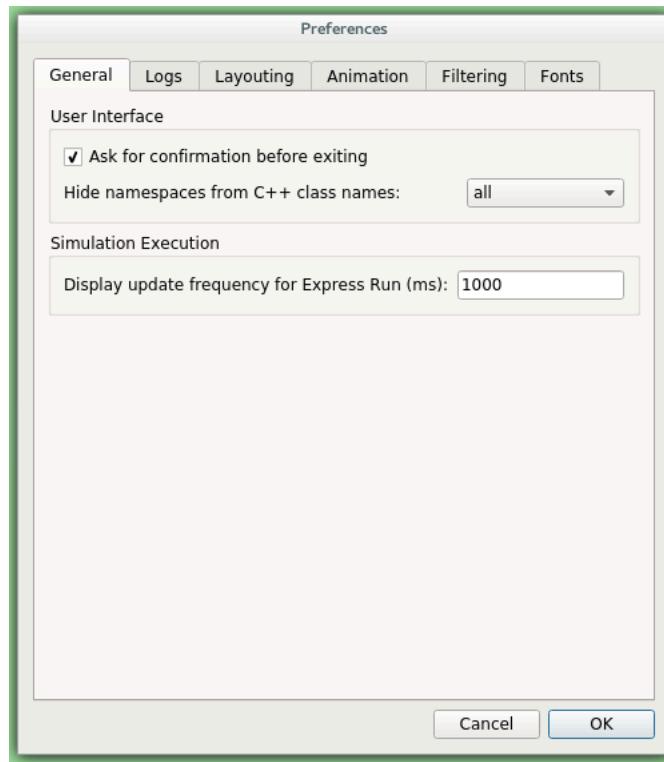


If you are debugging the simulation with a source level debugger, you may also use the *Inspect by pointer* menu item. Let the debugger display the address of the object to be inspected, and paste it into the dialog. Please note that entering an invalid pointer will crash the simulation.

## 7.7. The Preferences Dialog

Select *File | Preferences...* from the menu to display the runtime environment's configuration dialog. The dialog lets you adjust various display, network layouting and animation options.

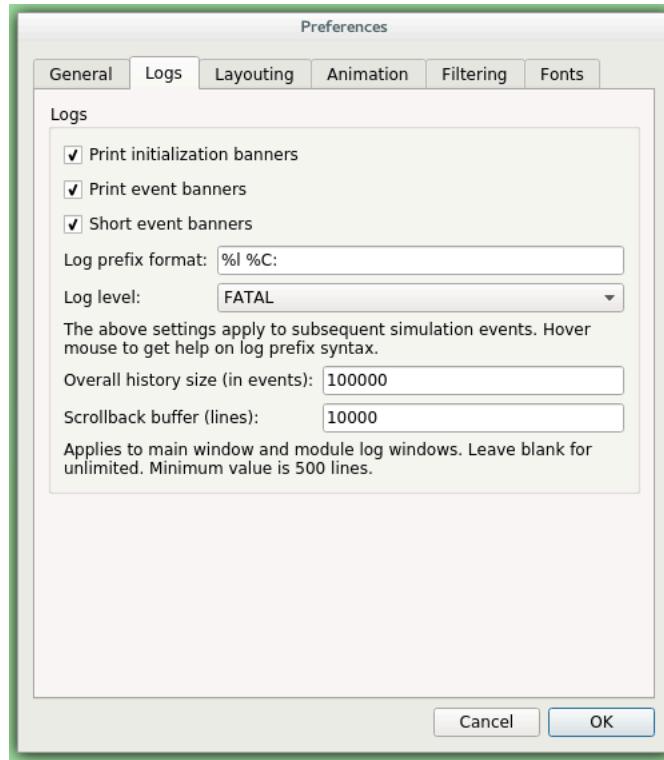
### 7.7.1. General



**Figure 7.28. General settings**

The *General* tab can be used to set the default user interface behavior. It is possible to set whether namespaces should be stripped off the displayed class names, and how often the user interface will be updated while the simulation runs in *Express* mode.

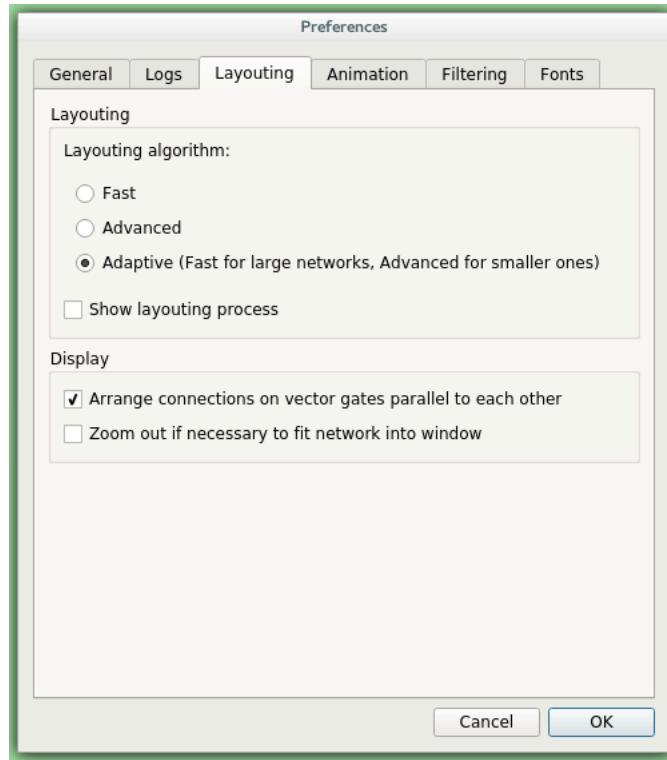
### 7.7.2. Logs



**Figure 7.29. Logging settings**

The *Logs* tab can be used to set the default logging behavior, such as the log level of modules that do not override it, the prefix format of event banners, and the size limit of the log buffer.

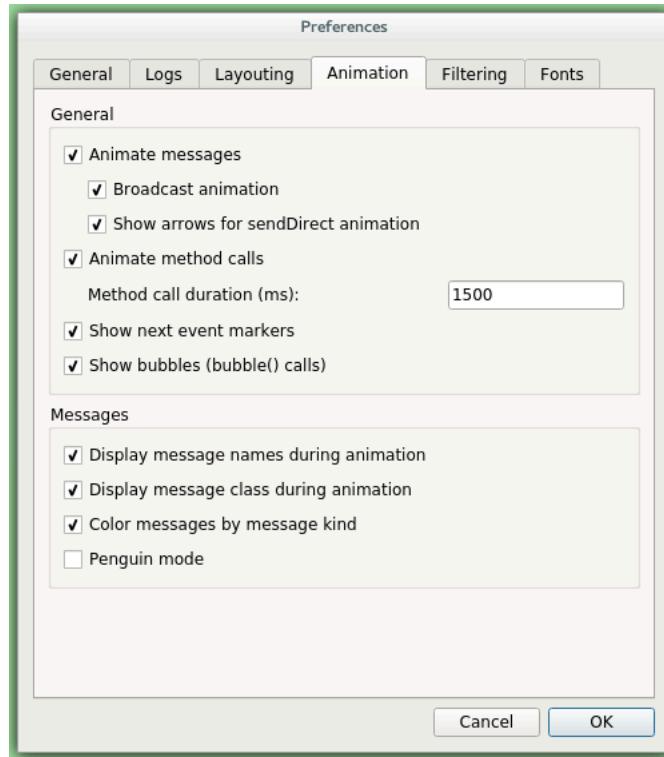
### 7.7.3. Configuring the Layouting Algorithm



**Figure 7.30. Layouting settings**

Qtenv provides automatic layouting for submodules that do not have their locations specified in the NED files. The layouting algorithm can be fine-tuned on the *Layouting* page of this dialog.

#### 7.7.4. Configuring Animation



**Figure 7.31. Animation settings**

Qtenv provides automatic animation when you run the simulation. You can fine-tune the animation settings using the *Animation* page of the settings dialog. If you do not need all visual feedback Qtenv provides, you can selectively turn off some of the features:

- **Animate messages:** Turns on/off the visualization of messages passing between modules.
- **Broadcast animation:** Handles message broadcasts in a special way (zero-time messages sent within the same event will be animated concurrently).
- **Show next event marker:** Highlights the module which will receive the next event.
- **Show a dotted arrow when a `sendDirect()` method call is executed.**
- **Show a flashing arrow when a method call occurs from one module to another.** The call is only animated if the called method contains the `Enter_Method()` macro.
- The display of message names and classes can also be turned off.

### 7.7.5. Timeline and Animation Filtering



**Figure 7.32. Filtering**

The *Filtering* page of the dialog serves two purposes. First, it lets you filter the contents of the *Timeline*. You can hide all self-messages (timers), or all non-self messages, and you can further reduce the number of messages shown on the timeline by also hiding the non-animated messages, explained below.

Second, you can suppress the animation of certain messages. For example, when your focus is routing protocol messages, you can suppress the animation of data traffic.

The text box lets you specify several filters, one per line. You can filter messages by name, class name, or by any other property that appears in the *Fields* page of the *Object Inspector* when you focus it on the given message object.

 When you select *Exclude Messages Like 'x' From Animation* from the context menu of a message object somewhere in the UI, it will add a new filter on this dialog page.

For object names, wildcards ("?", "\*") are allowed. "`{a-exz}`" matches any character in the range "a" .. "e" plus "x" and "z". You can match numbers: "`job{128..191}`" will match "job128", "job129", ..., "job191". "`job{128..}`" and "`job{..191}`" are also understood. You can combine patterns with AND, OR and NOT and parentheses (lowercase and, or, not are also acceptable). You can match against other object fields such as message length, message kind, etc. with the syntax "`fieldname(pattern)`". Put quotation marks around a pattern if it contains parentheses.

Some examples:

- `m*` : matches any object whose name begins with "m"
- `m* AND *-{0..250}` : matches any object whose name begins with "m" and ends with a dash and a number between 0 and 250

- not \*timer\* : matches any object whose name does not contain the substring "timer"
- not (\*timer\* or \*timeout\*) : matches any object whose name contains neither "timer" nor "timeout"
- kind(3) or kind({7..9}) : matches messages with message kind equal to 3, 7, 8 or 9
- className(IP\*) and data-\* : matches objects whose class name begins with "IP" and name begins with "data-"
- not className(cMessage) and byteLength({1500...}) : matches objects whose class is not cMessage and whose byteLength is at least 1500
- "or" or "and" or "not" or "\*(\*" or "msg(ACK)" : quotation marks needed when pattern is a reserved word or contains parentheses (note: msg(ACK) without parentheses would be interpreted as an object having an "msg" attribute with the value "ACK"!).

There is also a per-module setting the models can adjust programmatically that can prevent any animations taking place when inspecting a given module (`setBuiltInAnimationsAllowed()`).

### 7.7.6. Configuring Fonts



**Figure 7.33. Font selection**

The *Fonts* page of the settings dialog lets you select the typeface and font size for various user interface elements.

### 7.7.7. The .qtenvrc File

Settings are stored in `.qtenvrc` files. There are two `.qtenvrc` files: one is stored in the current directory and contains project-specific settings like the list of open inspectors; the other is saved into the user's home directory and contains global settings.



Inspectors are identified by their object names. If you have several components that share the same name (this is especially common for messages), you may end up with a lot of inspector windows when you start the simulation. In such cases, you may simply delete the `.qtenvrc` file.

## 7.8. QtEnv and C++

This section describes which C++ API functions various parts of QtEnv employ to display data and otherwise perform their functions. Most functions are member functions of the `cObject` class.

### 7.8.1. Inspectors

Inspectors display the hierarchical name (i.e. full path) and class name of the inspected object in the title using the `getFullPath()` and `getClassName()` `cObject` member functions. The *Go to parent* feature in inspectors uses the `getOwner()` method of `cObject`.

The *Object Navigator* displays the full name and class name of each object (`getFullName()` and `getClassName()`), and also the ID for classes that have one (`getId()` on `cMessage` and `cModule`). When you hover with the mouse, the tooltip displays the info string (`str()` method). The roots of the tree are the network module (`simulation.getSystemModule()`) and the FES (`simulation.getFES()`). Child objects are enumerated with the help of the `forEachChild()` method.

The *Object Inspector* in *Children* mode displays the full name, class name and info string (`getFullName()`, `getClassName()`, `str()`) of child objects enumerated using `forEachChild()`. `forEachChild()` can only enumerate objects that are subclassed from `cObject`. If you want your non-`cObject` variables (e.g. primitive types or STL containers) to appear in the *Children* tree, you need to wrap them into `cObject`. The `WATCH()` macro does exactly that: it creates an object wrapper that displays the variable's value via the wrapper's `str()` method. There are watch macros for STL containers as well, they present the wrapped object to QtEnv in a more structured way, via custom class descriptors (`cClassDescriptor`, see below).

One might ask how the `forEachChild()` method of modules can enumerate messages, queues, and other objects that are owned by the module. The answer is that the module class maintains a list of owned objects, and `cObject` automatically joins that list.

The *Object Inspector* displays an object's fields by making use of the class descriptor (`cClassDescriptor`) for that class. Class descriptors are automatically generated for new classes by the message compiler. Class descriptors for the OMNeT++ library classes are also generated by the message compiler, see `src/sim/sim_std.msg` in the source tree.

The *Network Display* uses `cSubmoduleIterator` to enumerate submodules, and its *Go to parent module* function uses `getParentModule()`. Background and submodule rendering is based on display strings (`getDisplayString()` method of `cComponent`).

The module log page of *Log Viewer* displays the output to EV streams from modules and channels.

The message/packet traffic page of *Log Viewer* shows information based on stored copies of sent messages (the copy is created using `dup()`), and stored sendhop information. The *Name* column displays the message name (`getFullName()`). However, the *Info* column does not display the string returned from `str()`, but rather, strings produced by a `cMessagePrinter` object. Message printers can be dynamically registered.

## 7.8.2. During Simulation

Qtenv sets up a network by calling `simulation.setupNetwork()`, then immediately proceeds to invoke `callInitialize()` on the root module. During simulation, `simulation.takeNextEvent()` and `simulation.executeEvent()` are called iteratively. When the simulation ends, Qtenv invokes `callFinish()` on the root module; the same happens when you select the *Conclude Simulation* menu item. The purpose of `callFinish()` is to record summary statistics at the end of a successful simulation run, so it will be skipped if an error occurs during simulation. On exit, and before a new network is set up, `simulation.deleteNetwork()` is called.

The *Debug Next Event* menu item issues the `int3 x86` assembly instruction on Windows, and raises a `SIGTRAP` signal on other systems.

## 7.9. Reference

### 7.9.1. Command-Line Options

A simulation program built with Qtenv accepts the following command line switches:

- `-h`: The program prints a help message and exits.
- `-u Qtenv`: Causes the program to start with Qtenv. (This is the default, unless the program hasn't been linked with Qtenv, or has another, custom environment library with a higher priority than Qtenv.)
- `-f filename`: Specifies 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, and a third one the module parameters you change frequently. The `-f` switch is optional and can be omitted.
- `-l filename`: Loads a shared library (.so file on Unix, .dll on Windows, and .dylib on Mac OS X). Multiple `-l` switches are accepted. Shared libraries may contain simple modules and other, arbitrary code. File names may be specified without the file extension and the lib name prefix (i.e. `foo` instead of `libfoo.so`).
- `-n filepath`: When present, overrides the NEDPATH environment variable and sets the source locations for simulation NED files.
- `-c configname`: Selects an INI configuration for execution.
- `-r runnumber`: It has the same effect as (but takes priority over) the `qtenv-default-run=` INI file configuration option. Run filters are also accepted. If there is more than one matching run, they are grouped to the top of the combobox.

### 7.9.2. Environment Variables

- `OMNETPP_IMAGE_PATH`: It controls where Qtenv will load images for network graphics (modules, background, etc.) from. The value should be a semicolon-separated list of directories, but on non-Windows systems, the colon is also accepted as separator. The default is `./bitmaps; ./images; <omnetpp>/images`, that is, by default Qtenv looks into the `bitmaps` and `images` folder of the simulation, and `images` folder in the working directory of your installation. The directories will be scanned recursively, and subdirectory names become part of the icon name; for example, if an `images/ directory` is listed, the file `images/misc/foo.png` will be registered as icon `misc/foo`. PNG, JPG and GIF files are accepted.
- `OMNETPP_DEBUGGER_COMMAND`: When set, overrides the factory default for the command used to launch the just-in-time debugger (`debugger-attach-command`). It

must contain '%u' (which will be substituted with the process ID of the simulation), and must not contain any additional '%' characters. Since the command has to return immediately, on Linux and macOS it is recommended that it ends with an ampersand ('&'). Settings on the command line or in an .ini file take precedence over this environment variable.

### 7.9.3. Configuration Options

Qtenv accepts the following configuration options in the INI file.

- `qtenv-extra-stack`: Specifies the extra amount of stack (in kilobytes) that is reserved for each `activity()` simple module when the simulation is run under Qtenv. This value is significantly higher than the similar one for Cmdenv (handling GUI events requires a large amount of stack space).
- `qtenv-default-config`: Specifies which INI file configuration Qtenv should set up automatically after startup. If there is no such option, Qtenv will ask which configuration to set up.
- `qtenv-default-run`: Specifies which run of the selected configuration Qtenv should set up after startup. If there is no such option, Qtenv will ask.

All other Qtenv settings can be changed via the GUI, and are saved into the `.qtenvrc` file in the user's home directory or in the current directory.

---

# **Chapter 8. The Tkenv Graphical Runtime Environment**

## **8.1. Features**

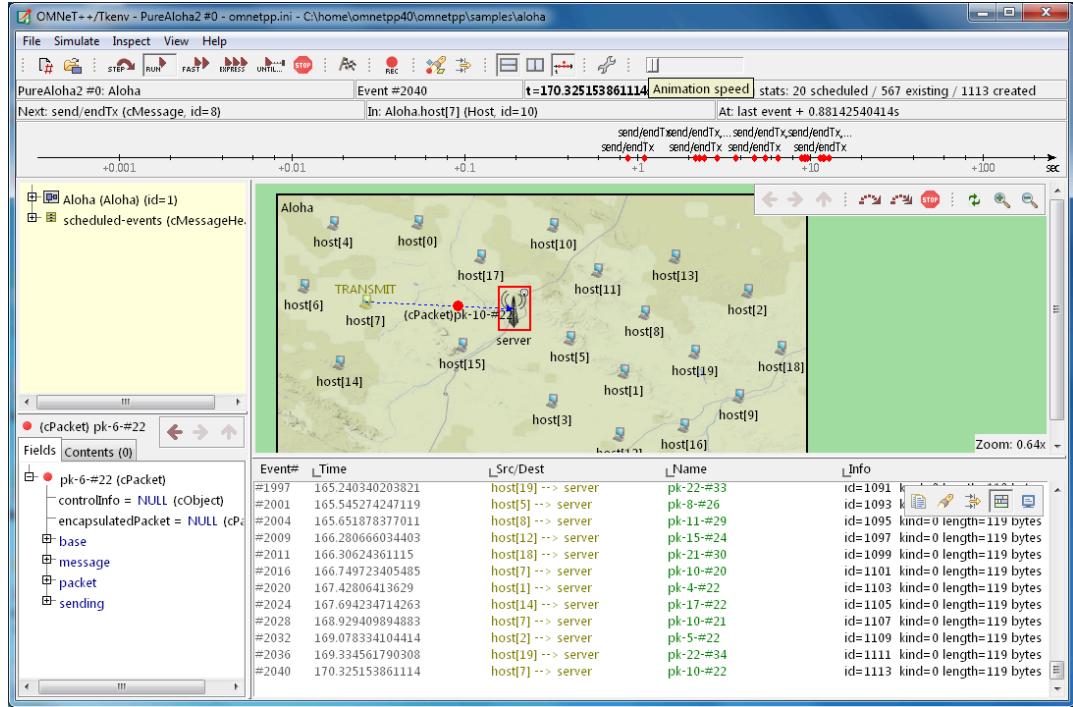
Tkenv is a graphical runtime interface for simulations. Tkenv supports interactive simulation execution, animation, inspection, tracing and debugging. In addition to model development and verification, Tkenv is also useful for presentation and educational purposes, since it allows the user to get a detailed picture of the state and history of the simulation at any point of its execution.

When used together with a C++ source-level debugger, Tkenv can significantly speed up model development.

Its most important features are:

- network visualization
- message flow animation
- various run modes: event-by-event, normal, fast, express
- run until (event or simulation time)
- simulation can be restarted
- log of message flow
- display of textual module logs
- inspectors for viewing contents of objects and variables in the model
- visualization of statistics (histograms, etc.) during simulation execution
- eventlog recording for later analysis
- snapshots (detailed report about the model: objects, variables, etc.)

## 8.2. Overview of the User Interface



**Figure 8.1. The main window of Tkenv**

The top of the window contains the following elements below the menu bar:

- **Toolbar:** The toolbar lets you access the most frequently used functions, such as stepping, running and stopping the simulation.
- **Status bar:** Two rows of various fields and gauges, displaying the current event number, simulation time, information about the next simulation event, and other details. When the simulation is running, it displays performance data like the number of events processed per second. The second row can be turned off to free up vertical space.
- **Timeline:** Displays the contents of the Future Events Set (FES) on a logarithmic time scale. The timeline can be turned off to free up vertical space.

The main window is divided into the following areas:

- **Object Navigator:** Displays the hierarchy of objects in the current simulation and in the FES.
- **Object Inspector:** Displays the contents and properties of the selected object.
- **Network Display:** Displays the network or any module graphically. This is also where animation takes place.
- **Log Viewer:** Displays the log of packets or messages sent between modules, or log messages output by modules during simulation.

Additionally, you can open inspector windows that float on top of the main window.

## 8.3. Using Tkenv

### 8.3.1. Starting Tkenv

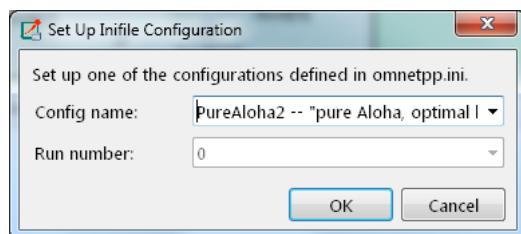
When you launch a simulation from the IDE, by default it will be started with Tkenv. When it does not, you can explicitly select Tkenv in the *Run* or *Debug* dialog.

Tkenv is also the default when you start the simulation from the command line. When necessary, you can force Tkenv by adding the `-u Tkenv` switch to the command line.

The complete list of command-line options, related environment variables and configuration options can be found at the end of this chapter.

### 8.3.2. Setting Up and Running the Simulation

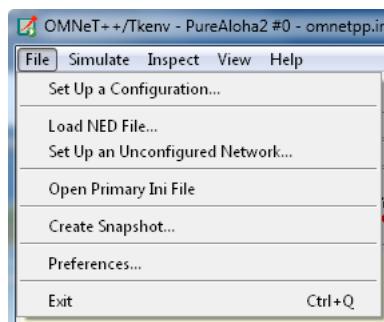
On startup, Tkenv reads the ini file(s) specified on the command line (or `omnetpp.ini` if none is specified), and automatically sets up the simulation described in them. If they contain several simulation configurations, Tkenv will ask you which one you want to set up.



**Figure 8.2. Setting Up a New Simulation**

Once a simulation has been set up (modules have been created and initialized), you can run it in various modes and examine its state. At any time you can restart the simulation, or set up another simulation. If you choose to quit Tkenv before the simulation finishes (or try to restart the simulation), Tkenv will ask you whether to finalize the simulation, which usually translates to saving summary statistics.

Functions related to setting up a simulation are in the *File* menu. Some of these functions are:



**Figure 8.3. The File menu**

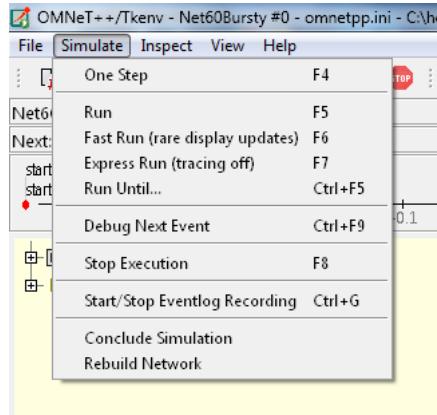
#### Set up a Configuration

This function lets you choose a configuration and run number from the ini file.

#### Open Primary Ini File

Opens the first ini file in an text window for viewing.

Simulation-related functions are in the *Simulate* menu, and are accessible via toolbar icons and keyboard shortcuts as well.



**Figure 8.4. The Simulate menu**

### Step

*Step* lets you execute one simulation event, that at the front of the FES. The next event is always shown on the status bar. The module where the next event will be delivered is highlighted with a red rectangle on the graphical display.

### Run (or Normal Run)

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 (e.g. you can open inspectors, etc.).

 If you find this mode too slow or distracting, you may switch off animation features in the *Preferences* dialog.

### Fast Run

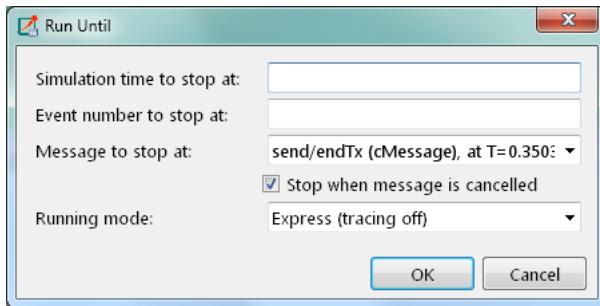
In *Fast* mode, animation is turned off. The inspectors and the message output windows are updated every 500 milliseconds (the actual number can be set in *File | Preferences...*). Fast mode is several times faster than the Run mode; the speed can increase by up to 10 times (or up to the configured event count).

### Express Run

In *Express mode*, the simulation runs at about the same speed as with Cmdenv, all tracing disabled. Module log is not recorded. You can interact with the simulation only once in a while, thus the run-time overhead of the user interface is minimal. You have to explicitly click the *Update inspectors* button if you want a display update.

### Run Until

You can run the simulation until a specified simulation time, event number or until a specific message has been delivered or canceled. This is a valuable tool during debugging sessions (select *Simulate | Run until...*). It is also possible to right-click on an event in the simulation timeline and choose the *Run until this event* menu item.



**Figure 8.5. The Run Until dialog**

### Run Until Next Event

It is also possible to run until an event occurs in a specified module. Browse for the module and choose *Run until next event in this module*. Simulation will stop once an event occurs in the selected module.

### Debug Next Event

This function is useful when you are running the simulation under a C++ source-level debugger. *Debug Next Event* will perform one simulation event just like *Step*, but executes a software debugger breakpoint (`int3` or `SIGTRAP`) just before entering the module's event handling code (`handleMessage()` or `activity()`). This will cause the debugger to stop the program there, letting you examine state variables, single-step, etc. When you resume execution, Tkenv will get back control and become responsive again.

### Recording an Event Log

The OMNeT++ simulation kernel allows you to record event related information into a file which later can be used to analyze the simulation run using the *Sequence Chart* tool in the IDE. Eventlog recording can be turned on with the `record-eventlog=true` ini file option, but also interactively, via the respective item in the *Simulate* menu, or using a toolbar button.

Note that the starting Tkenv with `record-eventlog=true` and turning on recording later does not result in exactly the same eventlog file. In the former case, all steps of setting up the network, such as module creations, are recorded as they happen; while for the latter, Tkenv has to "fake" a chain of steps that would result in the current state of the simulation.

### Conclude Simulation

This function finalizes the simulation by invoking the user-supplied `finish()` member functions on all module and channel objects in the simulation. The customary implementation of `finish()` is to record summary statistics. The simulation cannot be continued afterwards.

### Rebuild Network

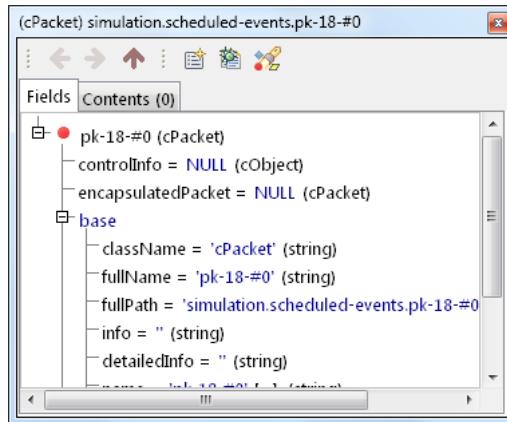
Rebuilds the simulation by deleting the current network and setting it up again. Improperly written simulations often crash when *Rebuild Network* is invoked; this is usually due to incorrectly written destructors in module classes.

## 8.3.3. Inspecting Simulation Objects

### Inspectors

The *Network Display*, the *Log Viewer* and the *Object Inspector* in the main window share some common properties: they display various aspects (graphical view / log messages / fields or contents) of a given object. Such UI parts are called *inspectors* in Tkenv.

The three inspectors mentioned above are built into the main window, but you can open add additional ones at any time. The new inspectors will open in floating windows above the main window, and you can have any number of them open.

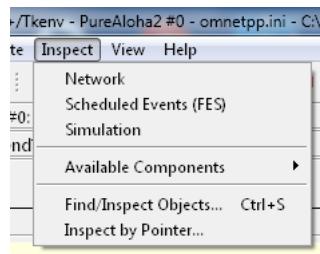


**Figure 8.6. A floating inspector window**

Inspectors come in many flavours. They can be graphical like the network view or a histogram inspector, textual like a log viewer, tree-based like an object inspector, or something entirely different.

### Opening Inspectors

Inspectors can be opened in various ways: by double-clicking an item in the *Object Navigator* or in other inspectors; by choosing one of the *Open...* menu items from the context menu of an object displayed on the UI; via the *Find/Inspect Objects* dialog (see later); or even by directly entering the C++ pointer of an object as a hex value. Inspector-related menu items are in the *Inspect* menu.



**Figure 8.7. The Inspect menu**

### History

Inspectors always show some aspect of one simulation object, but they can change objects. For example, in the *Network View*, when you double-click a submodule which is itself a compound module, the view will switch to showing the internals of that module; or, the *Object Inspector* will always show information about the object last clicked in the UI. Inspectors maintain a navigable history: the *Back/Forward* functions go to the object inspected before/after the currently displayed object. Objects that are deleted during simulation also disappear from the history.

### Restoring Inspectors

When you exit and then restart a simulation program, Tkenv tries to restore the open inspector windows. However, as object identity is not preserved across different runs of the same program, Tkenv uses the object full path, class name and object ID (where exists) to find and identify the object to be inspected.

Preferences such as zoom level or open/closed state of a tree node are usually maintained per object type (i.e. tied to the C++ class of the inspected object).

## Extending Tkenv

It is possible for the user to contribute new inspector types without modifying Tkenv code. For this, the inspector C++ code needs to include Tkenv header files and link with the Tkenv library. One caveat is that the Tkenv headers are not public API and thus subject to change in a new version of OMNeT++.

## 8.4. Using Tkenv with a Debugger

You can use Tkenv together with a C++ debugger, which is mainly useful when developing new models. When you do that, there are a few things you need to know.

Tkenv is a library that runs as part of the simulation program. This has a lot of implications, the most apparent being that when the simulation crashes (due to a bug in the model's C++ code), it will bring down the whole OS process, including the Tkenv GUI.

The second consequence is that suspending the simulation program in a debugger will also freeze the GUI until it is resumed. Also, Tkenv is also single-threaded and runs in the same thread as the simulation program, so even if you only suspend the simulation's thread in the debugger, the UI will freeze.

The Tkenv UI deals with `cObjects` (the C++ methods that the GUI relies on are defined on `cObject`). All other data such as primitive variables, non-`cObject` classes and structs, STL containers etc, are hidden from Tkenv. You may wrap objects into `cObjects` to make them visible for Tkenv, that's what e.g. the `WATCH` macros do as well.

The following sections go into detail about various parts and functions of the Tkenv UI.

## 8.5. Parts of the Tkenv UI

### 8.5.1. The Status Bar

The status bar shows the simulation's progress. It contains two rows, the second of which can be hidden using the *View | Status Details* menu item.

Net60Bursty #0: Net60	Event #1085	t=30.70712665501s	Msg stats: 116 scheduled / 998 existing / 1027 created
Next: endTxEvent (cMessage, id=260)	In: Net60.rte[53].queue[0] (L2Queue, id=319)		At: last event + 0.0000006s

**Figure 8.8. The status bar**

The first row contains the following items:

1. Ini config name, run number, and the name of the network
2. The event number of the next simulation event
3. The simulation time of the next (expected) simulation event.
4. Message statistics: the number of messages currently scheduled (i.e. in the FES); the number of message objects that currently exists in the simulation; and the number of message objects that have been created this far, including the already deleted ones. Out of the three, probably the middle one is the most useful: if it is steadily growing without apparent reason, the simulation model is probably missing some `delete msg` statements, and needs to be debugged.

When the simulation is paused or runs with animation, the second row displays the next expected simulation event. Note the word *expected*: certain schedulers may insert new events before the displayed event in the last moment. Some schedulers that tend to do that are those that accept input from outside sources: real-time scheduler, hybrid

or hardware-in-the-loop schedulers, parallel simulation schedulers, etc. Contents of the second row:

1. Name, C++ class and ID of the next message (event) object
2. The module where the next event will occur (i.e. the module where the message will be delivered)
3. Time of the next event since the last executed event.

When the simulation is running, displaying the next event becomes meaningless, and second row is replaced by the following performance gauges:

Net60Bursty #0: Net60	Event #375296	t=795.234291225081s	Msg stats: 116 scheduled / 6764 existing / 18104 created
Ev/sec: 69006.6	Simsec/sec: 146.995		Ev/simsec: 469.449

**Figure 8.9. The status bar during Fast or Express run**

1. Simulation speed: number of events processed per real second
2. Relative speed of the simulation (compared to real-time)
3. Event density: the number of events per simulated seconds

### 8.5.2. The Timeline



**Figure 8.10. The timeline**

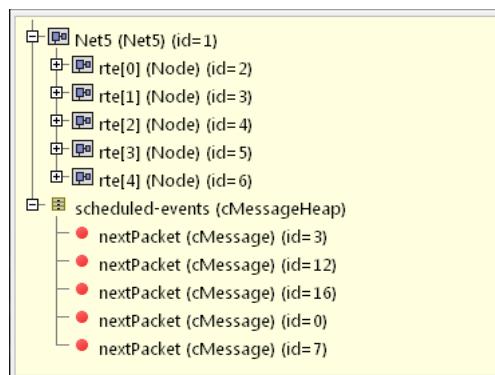
The timeline displays the contents of the Future Events Set on a logarithmic time scale. Each dot represents a message (event).

Clicking an event will focus it in the *Object Inspector*, and double-clicking will open a floating inspector window. Right-clicking will bring up a context menu with further actions.

The timeline is often crowded, limiting its usefulness. To overcome this, you can hide uninteresting events from the timeline: right-click the event, and choose *Exclude Messages Like 'x' from the Animation* from the context menu. This will hide events with same name and C++ class name from the timeline, and also skip the animation when such messages are sent from one module to another. You can view and edit the list of excluded messages on the *Filtering* page of the *Preferences* dialog. (Tip: the timeline context menu provides a shortcut to that dialog).

The timeline can be hidden (and revealed again) using the *View | Timeline* menu item.

### 8.5.3. The Object Navigator



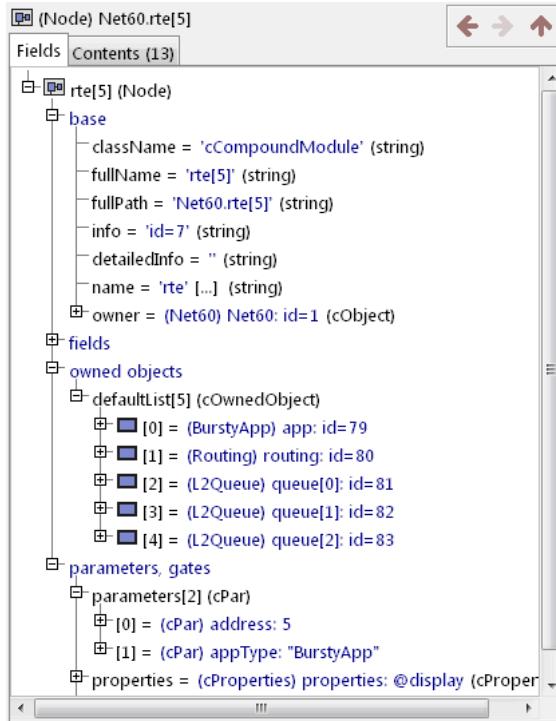
**Figure 8.11. The object tree**

The *Object Navigator* displays inspectable objects reachable from two root objects (the network module and the FES) in a tree form.

Clicking an object will focus it in the *Object Inspector*, and double-clicking will open a floating inspector window. Right-clicking will bring up a context menu with further actions.

#### 8.5.4. The Object Inspector

The *Object Inspector* is located below the *Object Navigator*, and lets you examine the contents of objects in detail. The *Object Inspector* always focuses on the object last clicked (or otherwise selected) on the Tkenv UI. It can be directly navigated as well, via the *Back*, *Forward*, and *Go to Parent* buttons, and also by double-clicking objects shown inside the inspector's area.



**Figure 8.12. The object inspector**

The inspector features a tabbed interface with two pages: *Fields* and *Contents*. The *Fields* page shows the fields (or data members) of the object, organized in categories. It uses meta-information generated by the message compiler to obtain the list of fields and their values. (This is true even for the built-in classes -- the simulation kernel contains their description of msg format.)

Class	Name	Info
cPar	address	5
cPar	appType	"BurstyApp"
cGate	port\$[0]	<-- rte[4].port\$o[2], (netw
cGate	port\$[1]	<-- rte[6].port\$o[0], (netw
cGate	port\$[2]	<-- rte[10].port\$o[0], (netw
cGate	port\$o[0]	--> rte[4].port\$[2], (netw
cGate	port\$o[1]	--> rte[6].port\$[0], (netw
cGate	port\$o[2]	--> rte[10].port\$[0], (netw
BurstyApp	app	id=79
Routing	routing	id=80
L2Queue	queue[0]	id=81
L2Queue	queue[1]	id=82
L2Queue	queue[2]	id=83

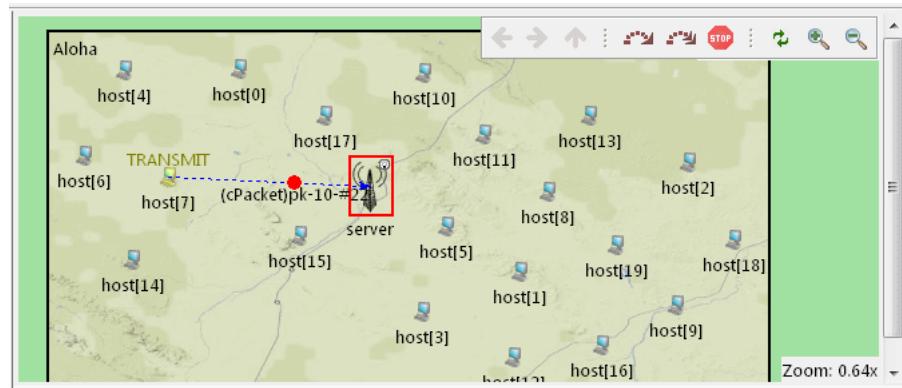
**Figure 8.13. The object inspector**

The *Contents* page lists the child objects of the currently inspected object. The child list is obtained via the `forEach()` method of the object.

### 8.5.5. The Network Display

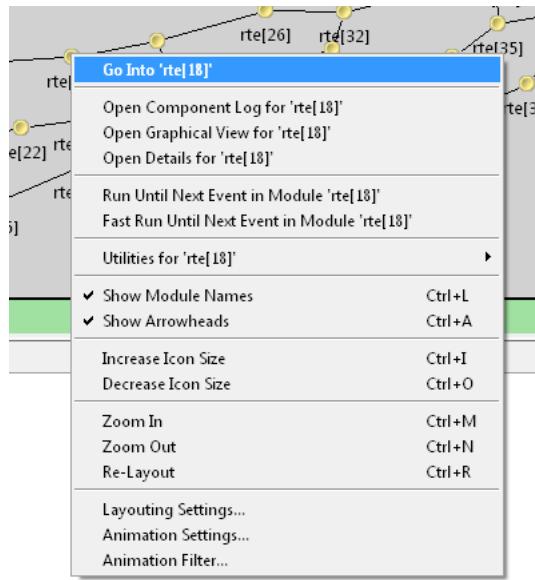
The network view provides a graphical view of the network and in general, modules. Graphical representation is based on display strings (@display properties in the NED file). You can go into any compound module by double-clicking its icon.

Message sending, method calls and certain other events are animated in the graphical view. You can customize animation in the *Animation* page of the *Preferences* dialog.

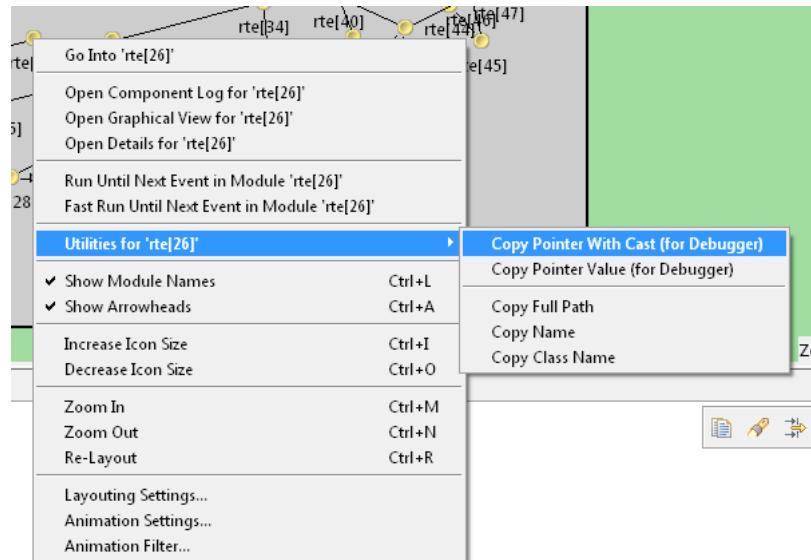


**Figure 8.14. The network display**

The context menu of submodules makes further actions available (see below).



**Figure 8.15. Submodule context menu**



**Figure 8.16. The Utilities submenu**

### Zooming and Panning

There are several ways for zooming the canvas, both using the mouse and the keyboard:

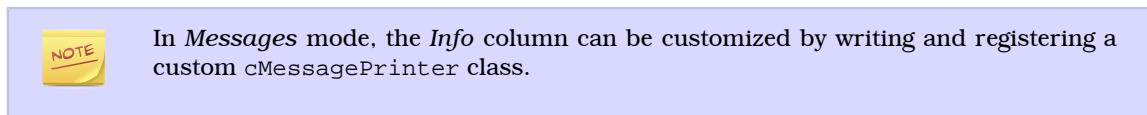
- Use **Ctrl+M** to zoom in, **Ctrl+N** to zoom out; the same functionality is also available in the canvas toolbar
- To zoom in around a point, double-click the canvas; use **Shift** + double-click to zoom out
- For marquee zoom, drag out a rectangle with the left mouse button while holding down **Ctrl**; you can cancel the operation with the right mouse button.
- Panning: moving the mouse while holding down the left mouse button will move the canvas; this is often a more comfortable way to navigate the canvas than using the scroll bars.

### 8.5.6. The Log Viewer

When you run the simulation, Tkenv will remember the output from logging statements (`EV << "Hello World\n";`) and the messages sent between modules, and can present it to you in a meaningful manner. Only the output from the last N events is preserved (N being configurable in the *Preferences* dialog), and only in Step, Run and Fast Run modes. (Express mode can be so fast because such overhead is turned off while it's active.)

The *Log Viewer* shows log related to one compound module and its subtree. It has two modes: *Messages* and *Log* mode, the default being *Messages*. You can switch between the two modes with tool icons on the inspector's local toolbar.

In *Messages* mode, the window displays messages sent between the (immediate) submodules of the inspected compound module, and messages sent out of, or into the compound module.



Event#	Time	Src/Dest	Name	Info
#1997	165.240340203821	host[19] --> server	pk-22-#33	id=1091 kind=0 length=119 bytes
#2001	165.545274247119	host[5] --> server	pk-8-#26	id=1093 kind=0 length=119 bytes
#2004	165.651878377011	host[8] --> server	pk-11-#29	id=1095 kind=0 length=119 bytes
#2009	166.280666034403	host[12] --> server	pk-15-#24	id=1097 kind=0 length=119 bytes
#2011	166.30624361115	host[18] --> server	pk-21-#30	id=1099 kind=0 length=119 bytes
#2016	166.749723405485	host[7] --> server	pk-10-#20	id=1101 kind=0 length=119 bytes
#2020	167.42806413629	host[1] --> server	pk-4-#22	id=1103 kind=0 length=119 bytes
#2024	167.694234714263	host[14] --> server	pk-17-#22	id=1105 kind=0 length=119 bytes
#2028	168.929409894883	host[7] --> server	pk-10-#21	id=1107 kind=0 length=119 bytes
#2032	169.078334104414	host[2] --> server	pk-5-#22	id=1109 kind=0 length=119 bytes
#2036	169.334561790308	host[19] --> server	pk-22-#34	id=1111 kind=0 length=119 bytes
#2040	170.325153861114	host[7] --> server	pk-10-#22	id=1113 kind=0 length=119 bytes

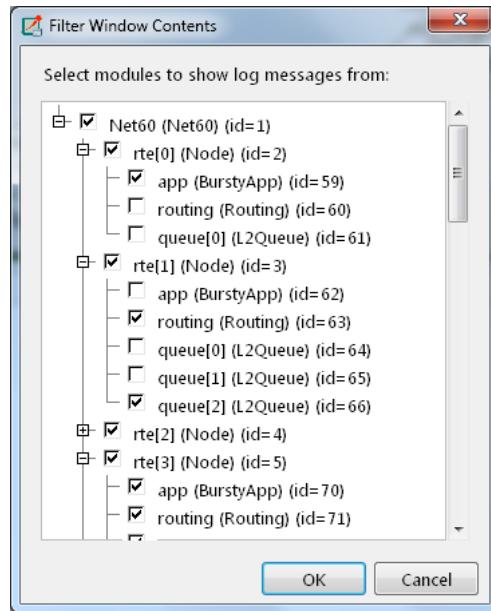
**Figure 8.17. The log viewer showing message traffic**

In *Log* mode, the window displays log lines that belong to submodules under the inspected compound module (i.e. the whole module subtree.)

** Event #1968 t=162.547823636021 Aloha.server (Server, id=2), on `pk-15-#23` (cPacket, id=1075) started receiving	
** Event #1969 t=162.585612830108 Aloha.host[6] (Host, id=9), on selfmsg `send/endTx` (cMessage, id=7) generating packet pk-9-#31	
** Event #1970 t=162.595612830108 Aloha.server (Server, id=2), on `pk-9-#31` (cPacket, id=1077) another frame arrived while receiving -- collision!	
** Event #1971 t=162.636990302687 Aloha.host[12] (Host, id=15), on selfmsg `send/endTx` (cMessage, id=13)	
** Event #1972 t=162.684779496774 Aloha.host[6] (Host, id=9), on selfmsg `send/endTx` (cMessage, id=7)	
** Event #1973 t=162.694779496774 Aloha.server (Server, id=2), on selfmsg `end-reception` (cMessage, id=0) reception finished	
** Event #1974 t=163.213214962284 Aloha.host[16] (Host, id=19), on selfmsg `send/endTx` (cMessage, id=17) generating packet pk-19-#24	

**Figure 8.18. The log viewer showing module log**

You can filter the content of the window to only include messages from specific modules. Open the log window's context menu and select *Filter Window Contents*.



**Figure 8.19. The log filter dialog**

General logging behavior can be controlled in the *Preferences* dialog.

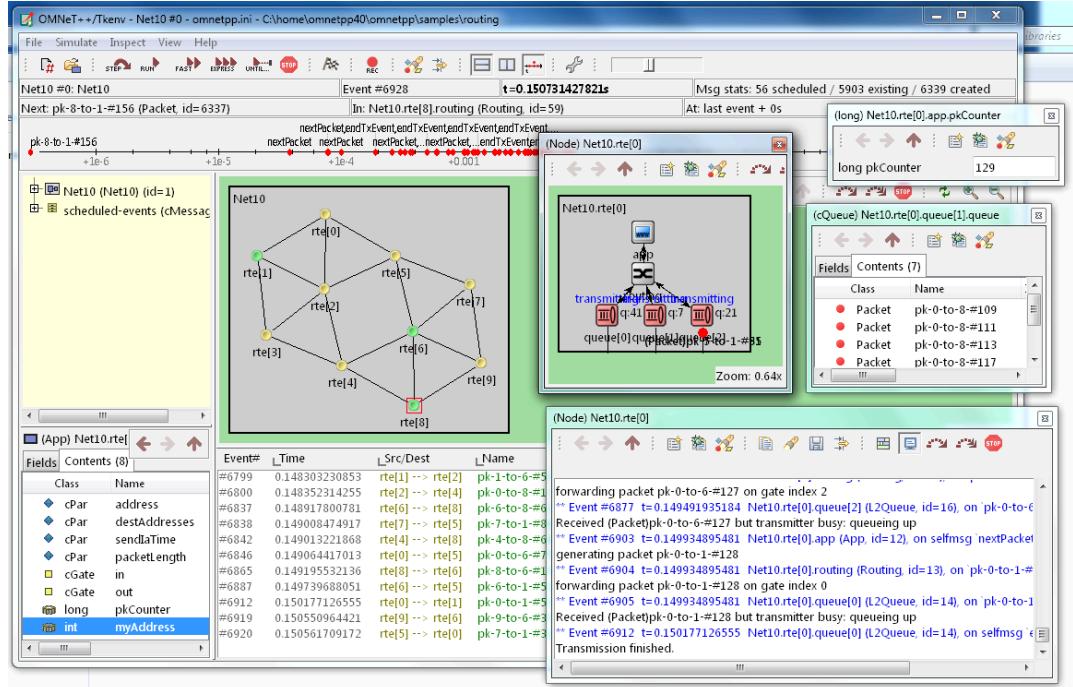
It is also possible to open separate log windows for individual modules. A log window for a compound module displays the log from all of its submodule tree. To open a log window, find the module in the module tree or the network display, right-click it and choose *Open Component Log* from the context menu.

## 8.6. Inspecting Objects

### 8.6.1. Object Inspectors

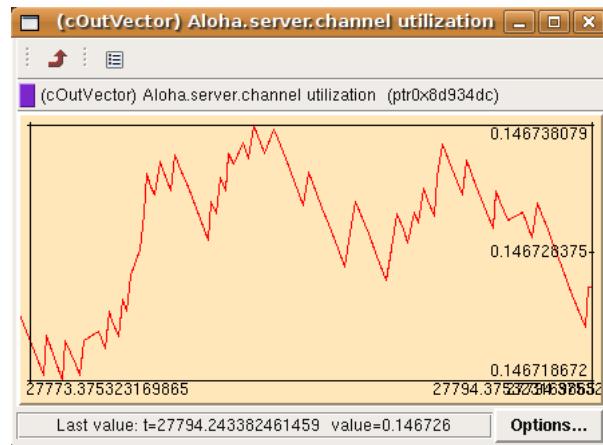
In addition to the inspectors embedded in the main window, Tkenv also lets you open floating inspector windows for objects. The screenshot below shows Tkenv with several inspectors open.

## The Tkenv Graphical Runtime Environment

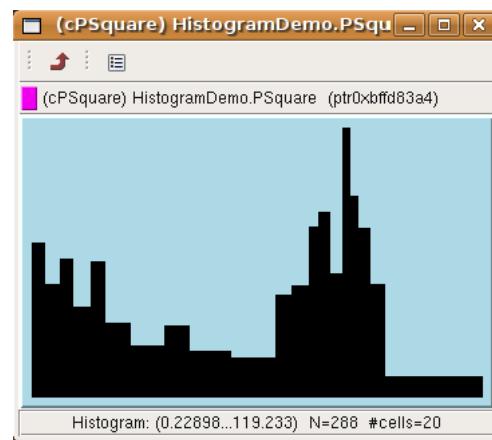


**Figure 8.20.** Tkenv with several floating inspectors open

The following screenshots show various inspectors available in Tkenv.



**Figure 8.21.** Graphical inspector for cOutVector object



**Figure 8.22.** Graphical for a histogram object

### 8.6.2. Browsing the Registered Components

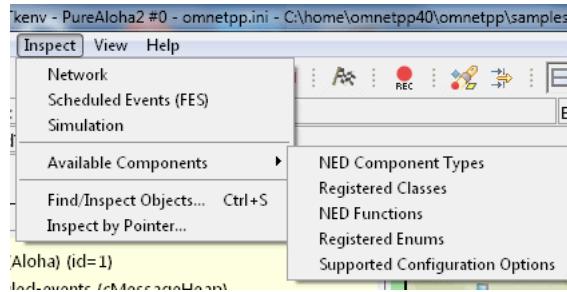


Figure 8.23. The Inspect menu

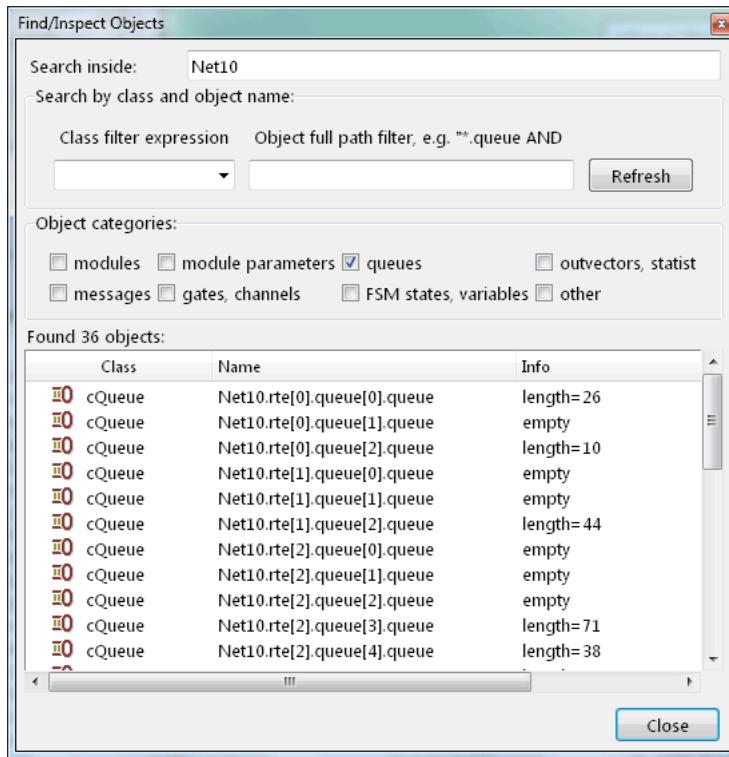
Registered components (NED Types, classes, functions, enums) can be displayed with the *Inspect | Available components* menu item. If an error message reports missing types or classes, you can check here whether the missing item is in fact available, i.e. registered correctly.

### 8.6.3. Querying Objects

The *Find/Inspect Objects* dialog lets you search the simulation for objects that meet certain criteria. The criteria may be the object name, class name, the value of a field of the object, or the combination of those. The results are presented in a table which you can sort by columns, and double-click items in it to inspect them.

Some possible use cases:

- Identifying bottlenecks in the network by looking at the list of all queues, and ordering them by length (i.e. have the result table sorted by the *Info* column)
- Finding nodes with the highest packet drop count. If the drop counts are watched variables (see `WATCH( )` macro), you can get a list of them.
- Finding modules that leak messages. If the live message count on the status bar keeps climbing up, you can issue a search for all message objects, and see where the leaked messages hide.
- Easy access for some data structures or objects, for example routing tables. You can search by name or class name, and use the result list as a collection of hotlinks, sparing you manual navigation in the simulation's object tree.



**Figure 8.24. Using the Find/Inspect Objects dialog for finding long queues**

The dialog lets you specify the search root, and the name and class name of the objects to find. The latter two accept wildcard patterns.

The checkboxes in the dialog can be used to select the object category that interests you. If you select a category, all objects with that type (and any type derived from it) will be included in the search. Alternatively, if you specify object class as a class filter expression, the search dialog will try to match the object's class name with the given string, meaning that objects of derived types will not be included.

You can also provide a generic filter expression, which matches the object's full path by default. Wildcards ("?", "\*") are allowed. "{a-exz}" matches any character in the range "a" .. "e" plus "x" and "z". You can match numbers: "\* . job{128..191}" will match objects named "job128", "job129", ..., "job191". "job{128..}" and "job{..191}" are also understood. You can combine patterns with AND, OR and NOT and parentheses (lowercase and, or, not are also accepted). You can match other object fields such as queue length, message kind, etc., with the syntax "fieldname(pattern)". If the pattern contains parentheses or space, you need to enclose in quotes. (HINT: You will want to start the pattern with "\* ." in most cases to match objects anywhere in the network!).

#### Examples:

\* . subnet2 . \* . destAddr "destAddr" "subnet2"

- \* . destAddr : Matches all objects with the name "destAddr" (likely module parameters).
- \* . node[8..10] . \* : Matches anything inside module node[8], node[9] and node[10] .
- className(cQueue) and not length(0) : Matches non-empty queue objects.
- className(cQueue) and length({10..}) : Matches queue objects with length >= 10.

- `kind(3)` or `kind({7..9})` : Matches messages with message kind equal to 3, 7, 8 or 9 (only messages have a "kind" attribute).
- `className(IP*)` and `*.data-*` : Matches objects whose class name begins with "IP" and name begins with "data-".
- `not className(cMessage)` and `byteLength({1500..})` : Matches messages whose class is not cMessage and byteLength is at least 1500 (only messages have a "byteLength" attribute).
- `"*(*) or "*msg(ACK)"` : Quotation marks needed when pattern is a reserved word or contains parentheses (note: `*.msg(ACK)` without parentheses would be interpreted as some object having a ".msg" attribute with the value "ACK"!).



Tkenv uses the `cObject::forEachChild` method to collect all objects from a tree recursively. If you have your own objects derived from `cObject`, you should redefine the `cObject::forEachChild` to function correctly with an object search.

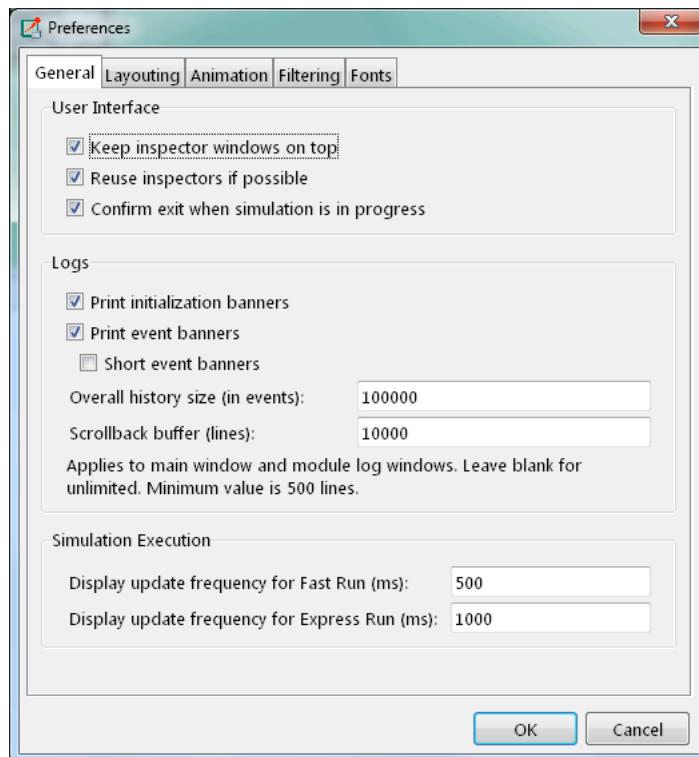


If you are debugging the simulation with a source level debugger, you may also use the *Inspect by pointer* menu item. Let the debugger display the address of the object to be inspected, and paste it into the dialog. Please note that entering an invalid pointer will crash the simulation.

## 8.7. The Preferences Dialog

Select *File | Preferences...* from the menu to display the runtime environment's configuration dialog. The dialog lets you adjust various display, network layouting and animation options.

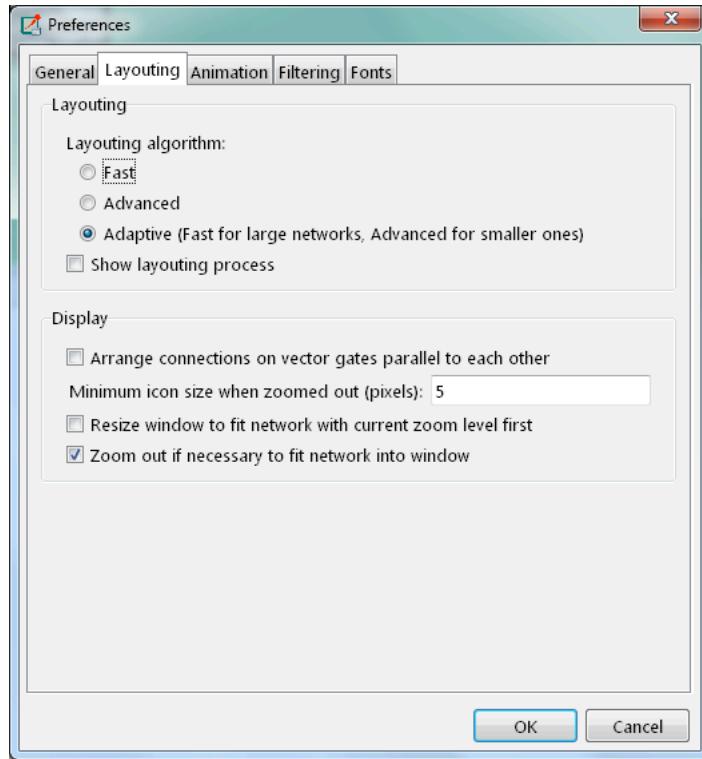
### 8.7.1. General



**Figure 8.25. General settings**

The *General* tab can be used to set the default display and logging behavior. It is possible to set how often the user interface will be updated during the simulation run.

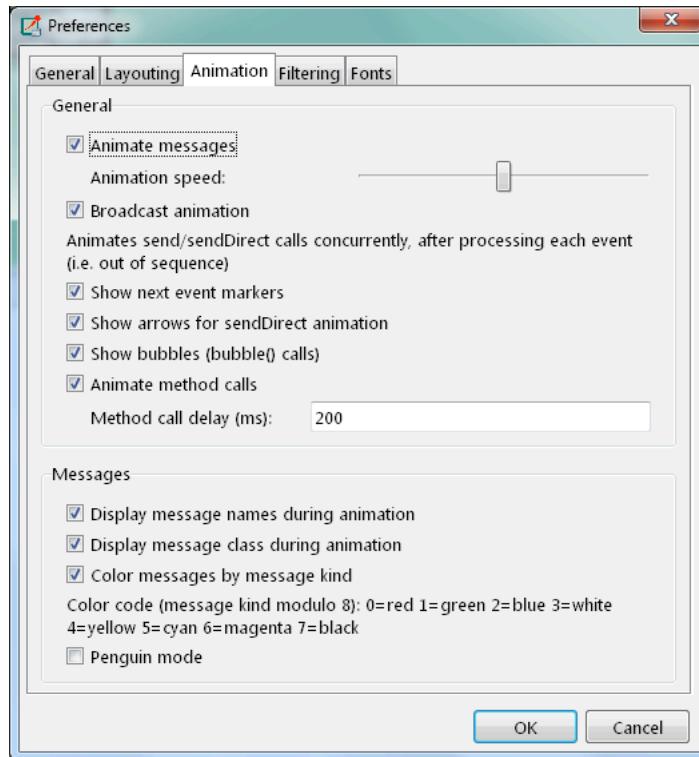
### 8.7.2. Configuring the Layouting Algorithm



**Figure 8.26. Layouting settings**

Tkenv provides automatic layouting for submodules that do not have their locations specified in the NED files. The layouting algorithm can be fine-tuned on the *Layouting* page of this dialog.

### 8.7.3. Configuring Animation

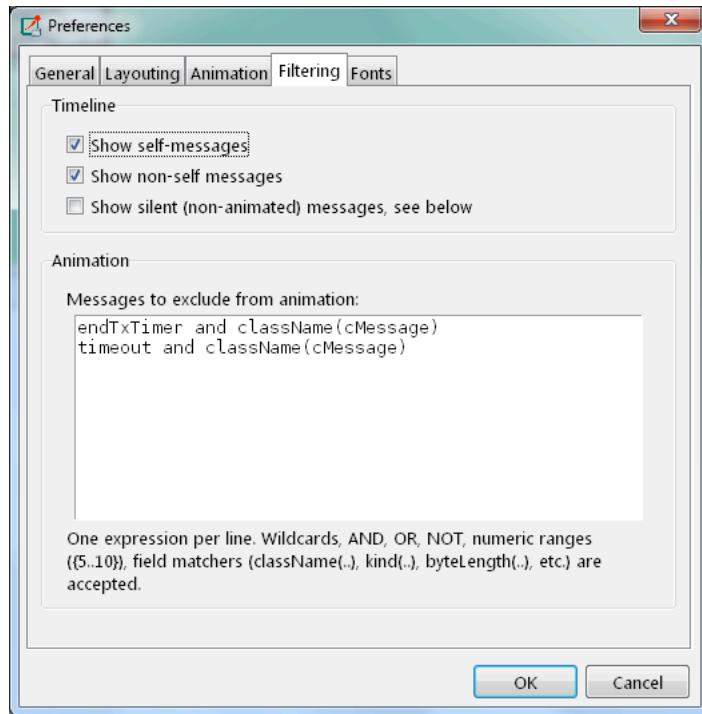


**Figure 8.27. Animation settings**

Tkenv provides automatic animation when you run the simulation. You can fine-tune the animation settings using the *Animation* page of the settings dialog. If you do not need all visual feedback Tkenv provides, you can selectively turn off some of the features:

- Animate messages: Turns on/off the visualization of messages passing between modules.
- Broadcast animation: Handles message broadcasts in a special way (messages sent within the same event will be animated concurrently).
- Show next event marker: Highlights the module which will receive the next event.
- Show a dotted arrow when a `sendDirect()` method call is executed.
- Show a flashing arrow when a method call occurs from one module to another. The call is only animated if the called method contains the `Enter_Method()` macro.
- The display of message names and classes can also be turned off.

### 8.7.4. Timeline and Animation Filtering



**Figure 8.28. Filtering**

The *Filtering* page of the dialog serves two purposes. First, it lets you filter the contents of the *Timeline*. You can hide all self-messages (timers), or all non-self messages, and you can further reduce the number of messages shown on the timeline by also hiding the non-animated messages, explained below.

Second, you can suppress the animation of certain messages. For example, when your focus is routing protocol messages, you can suppress the animation of data traffic.

The text box lets you specify several filters, one per line. You can filter messages by name, class name, or by any other property that appears in the *Fields* page of the *Object Inspector* when you focus it on the given message object.

 When you select *Exclude from Animation* from the context menu of a message object somewhere in the UI, it will add a new filter on this dialog page.

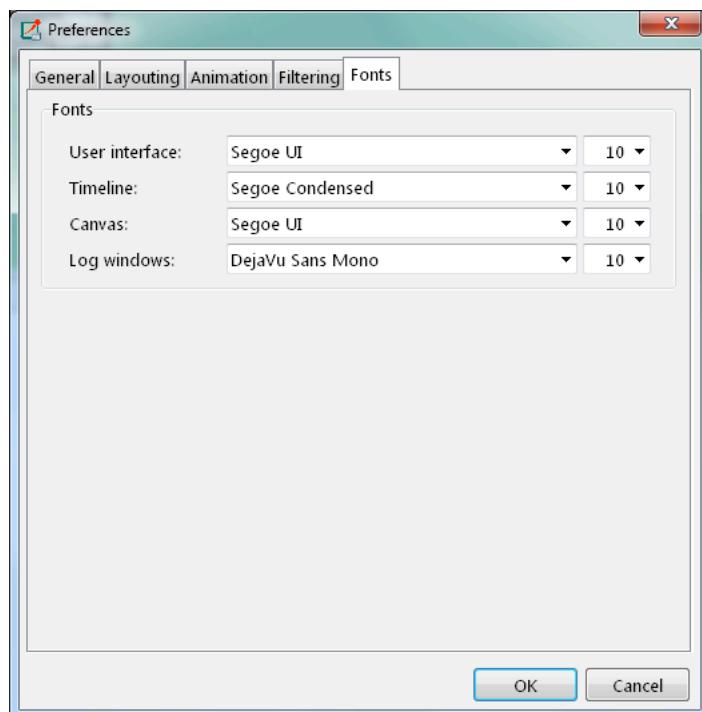
For object names, wildcards ("?", "\*") are allowed. "{a-exz}" matches any character in the range "a" .. "e" plus "x" and "z". You can match numbers: "job{128..191}" will match "job128", "job129", ..., "job191". "job{128..}" and "job{..191}" are also understood. You can combine patterns with AND, OR and NOT and parentheses (lowercase and, or, not are also acceptable). You can match against other object fields such as message length, message kind, etc. with the syntax "fieldname(pattern)". Put quotation marks around a pattern if it contains parentheses.

Some examples:

- m\* : matches any object whose name begins with "m"
- m\* AND \*-{0..250} : matches any object whose name begins with "m" and ends with a dash and a number between 0 and 250

- not `*timer*` : matches any object whose name does not contain the substring "timer"
- not `(*timer* or *timeout*)` : matches any object whose name contains neither "timer" nor "timeout"
- `kind(3) or kind({7..9})` : matches messages with message kind equal to 3, 7, 8 or 9
- `className(IP*) and data-*` : matches objects whose class name begins with "IP" and name begins with "data-"
- `not className(cMessage) and byteLength({1500..})` : matches objects whose class is not cMessage and whose byteLength is at least 1500
- "or" or "and" or "not" or "`*(*)` or `"msg(ACK)"` : quotation marks needed when pattern is a reserved word or contains parentheses (note: msg(ACK) without parentheses would be interpreted as an object having an "msg" attribute with the value "ACK"!).

### 8.7.5. Configuring Fonts



**Figure 8.29. Font selection**

The *Fonts* page of the settings dialog lets you select the typeface and font size for various user interface elements.

### 8.7.6. The `.tkenvrc` File

Settings are stored in `.tkenvrc` files. There are two `.tkenvrc` files: one is stored in the current directory and contains project-specific settings like the list of open inspectors; the other is saved into the user's home directory and contains global settings.



Inspectors are identified by their object names. If you have several components that share the same name (this is especially common for messages), you may end up with a

lot of inspector windows when you start the simulation. In such cases, you may simply delete the `.tkenvrc` file.

 Error messages generated by the Tkenv runtime environment are logged to the `.tkenvlog` file.

## 8.8. Tkenv and C++

This section describes which C++ API functions various parts of Tkenv employ to display data and otherwise perform their functions. Most functions are member functions of the `cObject` class.

### 8.8.1. Inspectors

Inspectors display the hierarchical name (i.e. full path) and class name of the inspected object in the title using the `getFullPath()` and `getClassName()` `cObject` member functions. The *Go to parent* feature in inspectors uses the `getOwner()` method of `cObject`.

The *Object Navigator* displays the full name and class name of each object (`getFullName()` and `getClassName()`), and also the ID for classes that have one (`getId()` on `cMessage` and `cModule`). When you hover with the mouse, the tooltip displays the info string (`str()` method). The roots of the tree are the network module (`simulation.getSystemModule()`) and the FES (`simulation.getFES()`). Child objects are enumerated with the help of the `forEachChild()` method.

In the *Object Inspector*, the *Contents* page displays the full name, class name and info string (`getFullName()`, `getClassName()`, `str()`) of child objects enumerated using `forEachChild()`. `forEachChild()` can only enumerate objects that are subclassed from `cObject`. If you want your non-`cObject` variables (e.g. primitive types or STL containers) to appear in the *Contents* list, you need to wrap them into `cObject`. The `WATCH()` macro does exactly that: it creates an object wrapper that displays the variable's value via the wrapper's `str()` method. There are watch macros for STL containers as well, they present the wrapped object to Tkenv in a more structured way, via custom class descriptors (`cClassDescriptor`, see below).

One might ask how the `forEachChild()` method of modules can enumerate messages, queues, and other objects that are owned by the module. The answer is that the module class maintains a list of owned objects, and `cObject` automatically joins that list.

The *Object Inspector's Fields* page display an object's fields by making use of the class descriptor (`cClassDescriptor`) for that class. Class descriptors are automatically generated for new classes by the message compiler. Class descriptors for the OMNeT++ library classes are also generated by the message compiler, see `src/sim/sim_std.msg` in the source tree.

The *Network Display* uses `cSubmoduleIterator` to enumerate submodules, and its *Go to parent module* function uses `getParentModule()`. Background and submodule rendering is based on display strings (`getDisplayString()` method of `cComponent`).

The module log page of *Log Viewer* displays the output to EV streams from modules and channels.

The message/packet traffic page of *Log Viewer* shows information based on stored copies of sent messages (the copy is created using `dup()`), and stored sendhop information. The *Name* column displays the message name (`getFullName()`). However, the *Info* column does not display the string returned from `str()`, but rather, strings

produced by a `cMessagePrinter` object. Message printers can be dynamically registered.

### 8.8.2. During Simulation

Tkenv sets up a network by calling `simulation.setupNetwork()`, then immediately proceeds to invoke `callInitialize()` on the root module. During simulation, `simulation.takeNextEvent()` and `simulation.executeEvent()` are called iteratively. When the simulation ends, Tkenv invokes `callFinish()` on the root module; the same happens when you select the *Conclude Simulation* menu item. The purpose of `callFinish()` is to record summary statistics at the end of a successful simulation run, so it will be skipped if an error occurs during simulation. On exit, and before a new network is set up, `simulation.deleteNetwork()` is called.

The *Debug Next Event* menu item issues the `int3 x86` assembly instruction on Windows, and raises a `SIGTRAP` signal on other systems.

## 8.9. Reference

### 8.9.1. Command-Line Options

A simulation program built with Tkenv accepts the following command line switches:

- `-h`: The program prints a help message and exits.
- `-u Tkenv`: Causes the program to start with Tkenv. (This is the default, unless the program hasn't been linked with Tkenv, or has another, custom environment library with a higher priority than Tkenv.)
- `-f fileName`: Specifies 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, and a third one the module parameters you change frequently. The `-f` switch is optional and can be omitted.
- `-l fileName`: Loads a shared library (.so file on Unix, .dll on Windows, and .dylib on Mac OS X). Multiple `-l` switches are accepted. Shared libraries may contain simple modules and other, arbitrary code. File names may be specified without the file extension and the lib name prefix (i.e. `foo` instead of `libfoo.so`).
- `-n filePath`: When present, overrides the `NEDPATH` environment variable and sets the source locations for simulation NED files.
- `-c configName`: Selects an INI configuration for execution.
- `-r run-number`: It has the same effect as (but takes priority over) the `tkenv-default-run=` INI file configuration option.

### 8.9.2. Environment Variables

- `OMNETPP_TKENV_DIR`: In case of nonstandard installation, it may be necessary to set this environment variable to point to `<omnetpp>/src/tkenv` so that Tkenv can find its parts written in Tcl.
- `OMNETPP_IMAGE_PATH`: It controls where Tkenv will load images for network graphics (modules, background, etc.) from. The value should be a semicolon-separated list of directories, but on non-Windows systems, the colon is also accepted as separator. The default is `<omnetpp>/images; ./images`, that is, by default Tkenv looks into the `images` folder of your installation, and `images` folder in the working directory of the simulation. The directories will be scanned recursively, and subdirectory names

become part of the icon name; for example, if an `images/` directory is listed, the file `images/misc/foo.png` will be registered as icon `misc/foo`. PNG and GIF files are accepted.

- `OMNETPP_PLUGIN_PATH`: Certain aspects of Tkenv can be extended with plugins written in Tcl. The path where such plugins are looked for is taken from this environment variable. It is expected to contain a semicolon-separated list of directories, and defaults to `./plugins`. (Again, colon is also accepted as separator on non-Windows systems.)

### 8.9.3. Configuration Options

Tkenv accepts the following configuration options in the INI file.

- `tkenv-extra-stack`: 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).
- `tkenv-default-config`: Specifies which INI file configuration Tkenv should set up automatically after startup. If there is no such option, Tkenv will ask which configuration to set up.
- `tkenv-default-run`: Specifies which run of the selected configuration Tkenv should set up after startup. If there is no such option, Tkenv will ask.
- `tkenv-plugin-path`: Tkenv will append this value to the value of the `OMNETPP_PLUGIN_PATH` or its built-in default value to get the list of directories to be scanned for plugins written in Tcl.

All other Tkenv settings can be changed via the GUI, and are saved into the `.tkenvrc` file in the user's home directory or in the current directory.

# Chapter 9. Sequence Charts

## 9.1. Introduction

This chapter describes the Sequence Chart and the Eventlog Table tools. Both of them display an eventlog file recorded by the OMNeT++ simulation kernel.

An eventlog file contains a log of messages sent during the simulation and the details of events that prompted their sending or reception. This includes both messages sent between modules and self-messages (timers). The user can control the amount of data recorded from messages, start/stop time, which modules to include in the log, and so on. The file also contains the topology of the model (i.e. the modules and their interconnections).

 Please refer to the OMNeT++ Manual for further details on eventlog files and their exact format.

The Sequence Chart displays eventlog files in a graphical form, focusing on the causes and consequences of events and message sends. They help the user understand complex simulation models and help with the correct implementation of the desired component behaviors. The Eventlog Table displays an eventlog file in a more detailed and direct way. It is in a tabular format, so that it can show the exact data. Both tools can display filtered eventlogs created via the Eventlog Tool filter command as described in the OMNeT++ Manual, by a third party custom filter tool, or by the IDE's in-memory filtering.

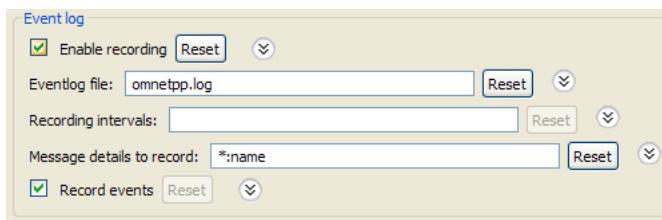
Using these tools, you will be able to easily examine every detail of your simulation back and forth in terms of simulation time or events. You will be able to focus on the behavior instead of the statistical results of your model.

## 9.2. Creating an Eventlog File

The INI File Editor in the OMNeT++ IDE provides a group of widgets in the *Output Files* section to configure automatic eventlog recording. To enable it, simply put a checkmark next to its checkbox, or insert the line

```
record-eventlog = true
```

into the INI file. Additionally, you can use the `--record-evenlog` command line option or just click the record button on the Tkenv toolbar before starting the simulation.



**Figure 9.1. INI file eventlog configuration**

By default, the recorded eventlog file will be put in the project's `results` directory, with the name  `${configname} - ${runnumber}.elog`.

 If you override the default file name, please make sure that the file extension is `.elog`, so that the OMNeT++ IDE tools will be able to recognize it automatically.

The 'recording intervals' and 'record events' configuration keys control which events will be recorded based on their simulation time and on the module where they occur. The 'message details' configuration key specifies what will be recorded from a message's content. Message content will be recorded each time a message gets sent.

The amount of data recorded will affect the eventlog file size, as well as the execution speed of the simulation. Therefore, it is often a good idea to tailor these settings to get a reasonable tradeoff between performance and details.



Please refer to the OMNeT++ Manual for a complete description of eventlog recording settings.

## 9.3. Sequence Chart

This section describes the Sequence Chart in detail, focusing on its features without a particular example.

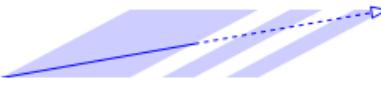
The Sequence Chart is divided into three parts: the top gutter, the bottom gutter and the main area. The gutters show the simulation time while the main area displays module axes, events and message sends. The chart grows horizontally with simulation time and vertically with the number of modules. Module axes can optionally display enumerated or numerical vector data.

There are various options, which control how and what the Sequence Chart displays. Some of these are available on the toolbar, while others are accessible only from the context menu.

### 9.3.1. Legend

Graphical elements on the Sequence Chart represent modules, events and messages, as listed in the following table.

	simple module axis
	compound module axis
	axis with attached vector data
	module full path as axis label
	initialization event
	self-message processing event
	message processing event
	event number
	self-message
	message send
	message reuse
	method call

	message send that goes far away; split arrow
(arrow with a dashed segment)	
	virtual message send; zigzag arrow
(arrow with zigzag)	
	transmission duration; reception at start
(blue parallelogram)	
	transmission duration; reception at end
(blue parallelogram)	
	split transmission duration; reception at start
(blue strips)	
	split transmission duration; reception at end
(blue strips)	
<code>pk-9-to-1-#58</code> (blue letters)	message name
<code>requestPacket()</code> (brown letters)	method name
	zero simulation time region
	simulation time hairline

### 9.3.2. Timeline

Simulation time may be mapped onto the horizontal axis in various ways; linear mapping is only one of the ways. The reason for having multiple mapping modes is that intervals between interesting events are often of different magnitudes (e.g. microsecond timings in a MAC protocol versus multi-second timeouts in higher layers), which is impossible to visualize using a linear scale.

The available timeline modes are:

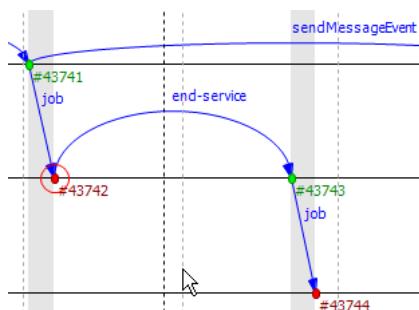
- Linear -- the simulation time is proportional to the distance measured in pixels.
- Event number -- the event number is proportional to the distance measured in pixels.
- Step -- the distance between subsequent events, even if they have non-subsequent event numbers, is the same.

- Nonlinear -- the distance between subsequent events is a nonlinear function of the simulation time between them. This makes the figure compact even if there are several magnitudes difference between simulation time intervals. On the other hand, it is still possible to decide which interval is longer and which one is shorter.
- Custom nonlinear -- like nonlinear. This is useful in those rare cases when the automatic nonlinear mode does not work well. The best practice is to switch to *Nonlinear* mode first and then to *Custom nonlinear*, so that the chart will continuously refresh as the parameters change. At the extreme, you can set the parameters so that the nonlinear mode becomes equivalent to linear mode or step mode.

You can switch between timeline modes using the  button on the toolbar or from the context menu.

### 9.3.3. Zero Simulation Time Regions

It is quite common in simulation models for multiple events to occur at the same simulation time, possibly in different modules. A region with a gray background indicates that the simulation time does not change along the horizontal axis within the area, thus all events inside it have the same simulation time associated with them.



**Figure 9.2. Nonlinear simulation time**

### 9.3.4. Module Axes

The Sequence Chart's vertical axis corresponds to modules in the simulation. By default, each simple module is displayed on a separate horizontal axis and events that occurred in that module are shown as circles on it. A compound module is represented with a double line and it will display events from all contained simple modules, except internal events and those that have their own axes displayed. An event is internal to a compound module if it only processes a message from, and sends out messages to, other modules inside.

It is not uncommon for some axes to not have any events at all. These axes would waste space by occupying some place on the screen, so by default they are omitted from the chart unless the *Show Axes Without Events* option is turned on. The discovery process is done lazily as you navigate through the chart, and it may add new axes dynamically as soon as it turns out that they actually have events.

Module axes can be reordered with the option *Axis Ordering Mode* . Ordering can be manual, or sorted by module name, by module id or by minimizing the total number of axes that arrows cross.

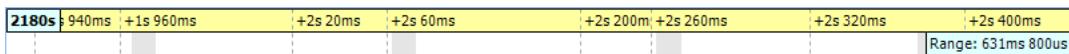
 The algorithm that minimizes crossings works by taking a random sample from the file and determines the order of axes from that (which means that the resulting order will only be an approximation). A more precise algorithm, which takes all arrows into account would not be practical because of the typically large size of eventlog files.

### 9.3.5. Gutter

The upper and lower edges of the Sequence Chart show a gutter that displays the simulation time. The left side of the top gutter displays a *time prefix* value, which should be added to each individual simulation time shown at the vertical hairlines. This reduces the number of characters on the gutter and allows easier recognition of simulation time changes in the significant digits. The right side of the figure displays the simulation time range that is currently visible within the window.



To see the simulation time at a specific point on the chart, move the mouse to the desired place and read the value in the blue box horizontally aligned with the mouse on the gutter.

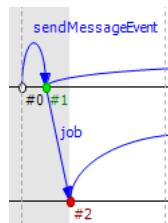


**Figure 9.3. Gutter and range**

### 9.3.6. Events

Events are displayed as filled circles along the module axes. A green circle represents the processing of a self-message, while a red circle is an event caused by receiving a message from another module. The event with event number zero represents the module initialization phase and may spread across multiple module axes because the simulation kernel calls each module during initialization. This event is displayed with a white background.

Event numbers are displayed below and to the right of their corresponding events and are prefixed with '#'. Their color changes according to their events' colors.



**Figure 9.4. Various event kinds**

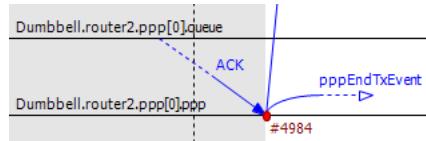
### 9.3.7. Messages

The Sequence Chart represents message sends with blue arrows. Vertically, the arrow starts at the module which sent the message and ends at the module which processed the message. Horizontally, the start and end points of the arrow correspond to the sender and receiver events. The message name is displayed near the middle of the arrow, but not exactly in the middle to avoid overlapping with other names between the same modules.

Sometimes, when a message arrives at a module, it simply stores it and later sends the very same message out. The events, where the message arrived, and where the message was actually sent, are in a so-called "message reuse" relationship. This is represented by a green dotted arrow between the two events. These arrows are not shown by default because timer self-messages are usually reused continuously. This would add unnecessary clutter to the chart and would make it hard to understand. To show and hide these arrows, use the *Show Reuse Messages* button on the toolbar.

Sometimes, depending on the zoom factor, a message send goes far away on the chart. In this case, the line is split into two smaller parts that are displayed at the two ends

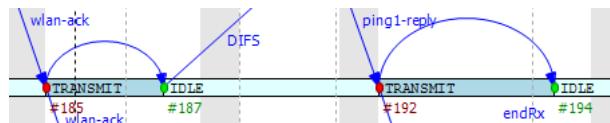
pointing towards each other, but without a continuous line connecting them. At one end of both arrow pieces is a dotted line while at the other end is a solid line. The one which is solid always points exactly to, or from, the event to which it is connected. The other one, which is dotted, either specifies the module where the arrow starts, or ends, or in the case of a self-message, it points toward the other arrow horizontally.



**Figure 9.5. Split arrows**

### 9.3.8. Displaying Module State on Axes

It is possible to display a module's state on an axis. The axis is then rendered as a colored strip that changes color every time the module state changes. The data are taken from an output vector in an *output vector file*, normally recorded by the simulation together with the eventlog file.



**Figure 9.6. Axis with state information displayed**

To attach an output vector to an axis, right-click the desired axis and select *Attach Vector to Axis* from the context menu. You will be prompted for an output vector file and for a vector in the file. If the vector is of type enum (that is, it has metadata attached that assigns symbolic names to values, e.g. *IDLE* for 0, *TRANSMIT* for 1, etc.), then the chart will display symbolic names inside the strip, otherwise it will display the values as numbers. The background coloring for the strip is automatic.



Recording output vectors is explained in the *OMNeT++ Simulation Manual*. It is recommended to turn on recording event numbers (`**.vector-record-eventnumbers = true` in file setting), because that allows the Sequence Chart tool to display state changes accurately even if there are multiple events at the same simulation time.

The format of output vector files is documented in an appendix of the Manual. To see whether a given output vector is suitable for the Sequence Chart, search for the vector declaration (`vector ...` line) in the file. When event numbers are enabled, the vector declaration will end in ETV (not TV). If a vector has an enum attached, there will be an attr enum line after the vector declaration. An example vector declaration with an enum:

```
vector 5 Net.host[2].radio state ETV
attr enum "IDLE=0,TRANSMIT=1,RECEIVE=2"
```

### 9.3.9. Zooming

To zoom in or out horizontally along the timeline, use the *Zoom In* and *Zoom Out* buttons on the toolbar. To decrease or increase the distance between the axes, use the *Increase/Decrease Spacing* commands.



When you zoom out, more events and messages become visible on the chart, making it slower. When you zoom in, message lines start breaking, making it less informative. Try to keep a reasonable zoom level.

### 9.3.10. Navigation

To scroll through the Sequence Chart, use either the scroll bars, drag with the left mouse button or scroll with the mouse wheel using the **Shift** modifier key for horizontal scroll.

There are also navigation options to go to the previous (**Shift+LEFT**) or next (**Shift+RIGHT**) event in the same module.

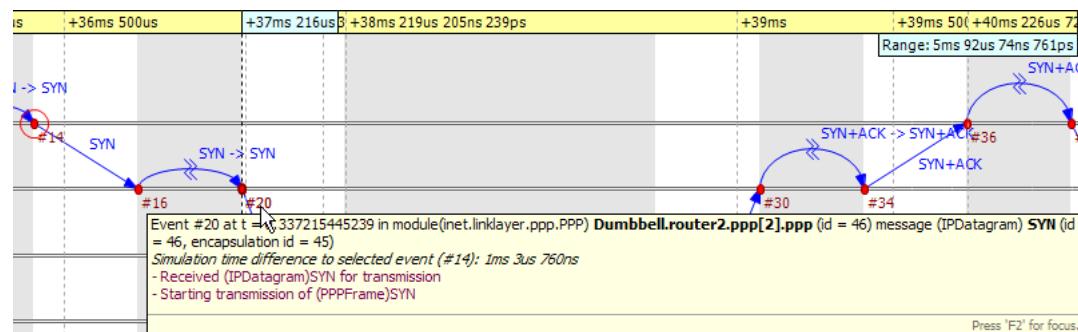
Similar to navigating in the Eventlog Table, to go to the cause event, press **Ctrl+LEFT**. To go to the arrival of a message send, press **Ctrl+RIGHT** while an event is selected.

### 9.3.11. Tooltips

The Sequence Chart displays tooltips for axes, events, message sends and reuses. When a tooltip is shown for any of the above, the chart will highlight the corresponding parts. Sometimes, when the chart is zoomed out it might show a complex tooltip immediately because there are multiple items under the mouse.



To measure the simulation time difference between two events, select one of them while staying at the other to display the tooltip.



**Figure 9.7. Event tooltip**

### 9.3.12. Bookmarks

Just like the Eventlog Table, the Sequence Chart also supports bookmarks to make navigation easier. Bookmarks are saved for the files rather than the various editors, therefore they are shared between them. The chart highlights bookmarked events with a circle around them similar to primary selection but with a different color.

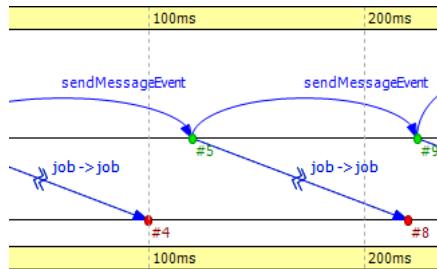
### 9.3.13. Associated Views

When you open an eventlog file in the Sequence Chart editor, it will automatically open the *Eventlog Table View* with the same file. If you select an event on the Sequence Chart editor, then the *Eventlog Table View* will jump to the same event and vice versa. This interconnection makes navigation easier and you can immediately see the details of the selected event's raw data.

### 9.3.14. Filtering

You can also filter the contents of the Sequence Chart. This actually means that some of the events are not displayed on the chart so that the user can focus on the relevant parts. When filtering is turned on (displayed in the status line), some of the message arrows might have a filter sign (a double zigzag crossing the arrow line's center). Such a message arrow means that there is a message going out from the source module, which after processing in some other filtered out modules, reaches the target module. The

message name of the arrow in this case corresponds to the first and the last message in the chain that was filtered out.



**Figure 9.8. Zigzag arrows**

When a module filter is used, it will determine which modules will have axes. If the events that occurred in a module are completely filtered out, then the Sequence Chart will not display the superfluous axis belonging to that module. This reduces the number of axes and makes it easier to understand the figure.

Events may not have subsequent event numbers, which means that the events in between have been filtered out. At the extreme, the chart may even be empty, meaning that there are no matching events at all.

To filter the Sequence Chart, open the *Filter Dialog* using the filter button on the toolbar. You can also filter from the context menu using the shortcuts provided for events and message sends currently under the mouse.

## 9.4. Eventlog Table

This section describes the Eventlog Table in details focusing on its features without a particular example.

The Eventlog Table has one row per line in the eventlog file. It has three columns. The first two are called event number and simulation time respectively. They show the values corresponding to the simulation event where the line was recorded. The third column, called details, contains the actual data, which varies for each line kind. The different kinds of lines can be easily recognized by their icons. Some lines, such as sending a message through a sequence of gates, relate to each other and are indented so that the user can recognize them more easily.

There are various options, which control how and what the Eventlog Table displays. Some of these are available on the toolbar, while others are accessible only from the context menu.

### 9.4.1. Display Mode

The eventlog file content may be displayed in two different notations. The *Raw* data notation shows exactly what is present in the file.

Event #	Time	Details
#6212	0.115848515392s	● E # 6212 t 0.115848515392 m 683 ce 5997 msg 1891
#6212	0.115848515392s	↳ MB sm 683 tm 668 m fireChangeNotification(RX-END.)
#6212	0.115848515392s	↳ ME
#6212	0.115848515392s	✖ DM id 1891 pe 6212
#6212	0.115848515392s	↳ BS id 1799 tid 1799 eid 1798 etid 1798 c IPDatagram n tcpseg(l=1024,0msg) pe 5997 k 0 p 0 l 8512 er 0 d n
#6212	0.115848515392s	↳ SH sm 683 sg 3 pd 0 td 0
#6212	0.115848515392s	↳ SH sm 676 sg 3 pd 0 td 0
#6212	0.115848515392s	↳ SH sm 675 sg 1048576 pd 0 td 0
#6212	0.115848515392s	↳ ES t 0.115848515392
#6213	0.115848515392s	● E # 6213 t 0.115848515392 m 677 ce 6212 msg 1799
#6213	0.115848515392s	↳ MB sm 677 tm 670 m localDelivery(1024,0,21) /m

**Figure 9.9. Raw notation**

The *Descriptive* notation, after some preprocessing, displays the log file in a readable format. It also resolves references and types, so that less navigation is required to understand what is going on. To switch between the two, use the *Display Mode*  button on the toolbar or the context menu.

Event #	Time	Details
#6212	0.115848515392s	● Event in module (PPP) <b>Dumbbell.sink[8].ppp[0].ppp</b> on arrival of message (PPPFrame) <b>tcpseg(l=1024,0msg)</b>
#6212	0.115848515392s	↳ Begin calling <b>fireChangeNotification(RX-END, )</b> in module (NotificationBoard) <b>Dumbbell.sink[8].notificationBo</b>
#6212	0.115848515392s	↳ End calling module
#6212	0.115848515392s	✖ Deleting message (PPPFrame) <b>tcpseg(l=1024,0msg)</b>
#6212	0.115848515392s	↳ Sending message (IPDatagram) <b>tcpseg(l=1024,0msg)</b> arriving at 0.115848515392s, now + 0s kind = 0 lengt
#6212	0.115848515392s	→ Sending through module (PPP) <b>ppp</b> gate <b>netwOut</b>
#6212	0.115848515392s	↳ Sending through module (PPPInterface) <b>Dumbbell.sink[8].ppp[0]</b> gate <b>netwOut</b>
#6212	0.115848515392s	↳ Sending through module (NetworkLayer) <b>Dumbbell.sink[8].networkLayer</b> gate <b>ifIn[0]</b>
#6212	0.115848515392s	↳ Arrival at 0.115848515392s, now + 0s
#6213	0.115848515392s	● Event in module (IP) <b>Dumbbell.sink[8].networkLayer.ip</b> on arrival of message (IPDatagram) <b>tcpseg(l=1024,</b>
#6213	0.115848515392s	↳ Begin calling <b>localDelivery(102 168 0 31)win</b> in module (RoutingTable) <b>Dumbbell.sink[8].routingTable</b>

**Figure 9.10. Descriptive notation**

#### 9.4.2. Name Mode

There are three different ways to display names in the Eventlog Table; it is configurable with the *Name Mode*  option. Full path and full name shows what you would expect. The smart mode uses the context of the line to decide whether a full path or a full name should be displayed. For each event line, this mode always displays the full path. For all other lines, if the name is the same as the enclosing event's module name, then it shows the full name only. This choice makes lines shorter and allows for faster reading.

#### 9.4.3. Type Mode

The option called *Type Mode* can be used to switch between displaying the C++ class name or the NED type name in parenthesis before module names. This is rarely used, so it is only available from the context menu.

#### 9.4.4. Line Filter

The Eventlog Table may be filtered by using the *Line Filter*  button on the toolbar. This option allows filtering for lines with specific kinds. There are some predefined filters.

You can also provide a custom filter pattern, referring to fields present in *Raw* mode, using a match expression. The following example is a custom filter, which will show message sends where the message's class is AirFrame.

BS and c(AirFrame)

Please refer to the OMNeT++ Manual for more details on match expressions.



To avoid confusion, event lines marked with green circles  are always shown in the Eventlog Table and are independent of the line filter.

#### 9.4.5. Navigation

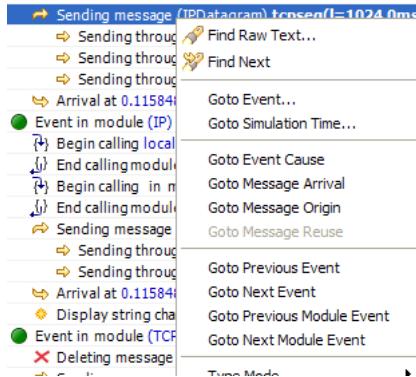
You can navigate using your keyboard and mouse just like in any other table. There are a couple of non-standard navigation options in the context menu, which can also be used with the keyboard.

The simplest are the *Goto Event* and the *Goto Simulation Time*, both of which simply jump to the designated location.

There are navigation options for going to the previous (**Alt+UP**) or next (**Alt+DOWN**) event in general, and to go to the previous (**Shift+UP**) or next (**Shift+DOWN**) event in the same module.

Some of the navigation options focus on the causes of events and consequences of message sends. To go to the cause event, press **Ctrl+UP**. To go to the arrival of a message send, press **Ctrl+DOWN**, after selecting the message being sent.

Finally, there are navigation options for message reuse relationships. You can go to the original event of a message from the line where it was being reused. In the other direction, you can go to the reused event of a message from the event where it was received. These options are enabled only if they actually make sense for the current selection.



**Figure 9.11. Navigation context menu**

#### 9.4.6. Selection

The Eventlog Table uses multiple selection even though most of the user commands require single selection.

#### 9.4.7. Searching

For performance reasons, the search function works directly on the eventlog file and not the text displayed in the Eventlog Table. It means that some static text present in *Descriptive* mode cannot be found. Usually, it is easier to figure out what to search for in *Raw* mode, where the eventlog file's content is directly displayed. The search can work in both directions, starting from the current selection, and may be case insensitive. To repeat the last search, use the *Find Next* command.

#### 9.4.8. Bookmarks

For easier navigation, the Eventlog Table supports navigation history. This is accessible from the standard IDE toolbar just like for other kinds of editors. It works by remembering each position where the user stayed more than 3 seconds. The navigation history is temporary and thus it is not saved when the file is closed.

Persistent bookmarks are also supported and they can be added from the context menu. A Bookmarked event is highlighted with a different background color.

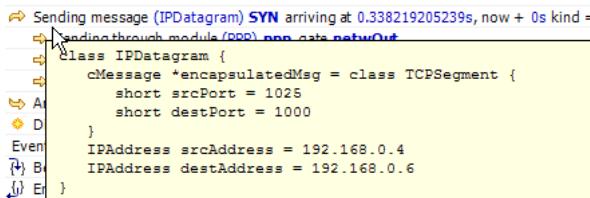
#6212	0.115848515392s	Arrival at 0.115848515392s, now + 0s
#6213	0.115848515392s	Event in module (IP) Dumbbellsink[8].networkLayer.ip on arrival of message (IPDatagram) tcpseg(l=1024,0msg)
#6213	0.115848515392s	Begin calling localDeliver(192.168.0.31)y/n in module (RoutingTable) Dumbbell.sink[8].routingTable
#6213	0.115848515392s	End calling module
#6213	0.115848515392s	Begin calling in module (InterfaceTable) Dumbbell.sink[8].interfaceTable
#6213	0.115848515392s	End calling module
#6213	0.115848515392s	Sending message (TCPSegment) tcpseg(l=1024,0msg) arriving at 0.115848515392s, now + 0s kind = 0 length = 0
#6213	0.115848515392s	Sending through module (IP) ip gate transportOut[0]
#6213	0.115848515392s	Sending through module (NetworkLayer) Dumbbell.sink[8].networkLayer gate TCPOut
#6213	0.115848515392s	Arrival at 0.115848515392s, now + 0s
#6213	0.115848515392s	Display string changed to t=fwd:121 up:123 ;q=queue;p=85,95;i=block/routing
#6214	0.115848515392s	Event in module (TCP) Dumbbellsink[8].tcp on arrival of message (TCPSegment) tcpseg(l=1024,0msg) from

**Figure 9.12. A bookmark**

To jump to a bookmark, use the standard Bookmark View (this is possible even after restarting the IDE).

### 9.4.9. Tooltips

Currently, only the message send lines have tooltips. If message detail recording was configured for the simulation, then a tooltip will show the recorded content of a message send over the corresponding line.



**Figure 9.13. A message send tooltip**

### 9.4.10. Associated Views

When you open an eventlog file in the Eventlog Table editor, it will automatically open the *Sequence Chart View* with the same file. If you select an event on the Eventlog Table editor, then the *Sequence Chart View* will jump to the same event and vice versa. This interconnection makes navigation easier, and you can immediately see the cause and effect relationships of the selected event.

### 9.4.11. Filtering

If the Eventlog Table displays a filtered eventlog, then subsequent events may not have subsequent event numbers. This means that the events in between have been filtered out. At the extreme, the table may even be empty, which means that there are no matching events at all.

## 9.5. Filter Dialog

The content of an eventlog can be filtered within the OMNeT++ IDE. This is on-the-fly filtering as opposed to the file content filtering provided by the Eventlog tool. To use on the fly filtering, open the filter configuration dialog with the button on the toolbar, enable some of the range, module, message, or trace filters, set the various filter parameters, and apply the settings. The result is another eventlog, resident in memory, that filters out some events.

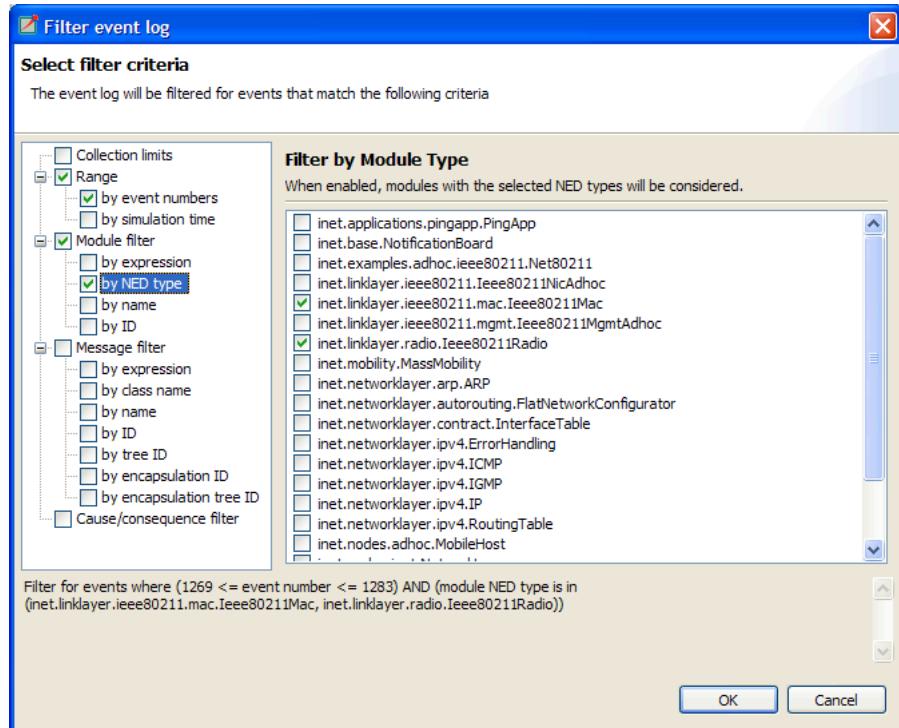


Similar to the command line Eventlog tool described in the OMNeT++ Manual, the in-memory filtering can only filter out whole events.

In-memory, on-the-fly filtering means that the filter's result is not saved into an eventlog file, but it is computed and stored within memory. This allows rapid switching between different views of the same eventlog within both the Sequence Chart and the Eventlog Table.

The filter configuration dialog shown in Figure 9.14, “Filter Dialog” has many options. They are organized into a tree with each part restricting the eventlog’s content. The individual filter components may be turned on and off independent of each other. This allows remembering the filter settings even if some of them are temporarily unused.

The combination of various filter options might be complicated and hard to understand. To make it easier, the *Filter Dialog* automatically displays the current filter in a human readable form at the bottom of the dialog.

**Figure 9.14. Filter Dialog**

### 9.5.1. Range Filter

This is the simplest filter, which filters out events from the beginning and end of the eventlog. It might help to reduce the computation time dramatically when defining filters, which otherwise would be very expensive to compute for the whole eventlog file.

### 9.5.2. Module Filter

With this kind of filter, you can filter out events that did not occur in any of the specified modules. The modules which will be included in the result can be selected by their NED type, full path, module id, or by a match expression. The expression may refer to the raw data present in the lines marked with 'MC' in the eventlog file.

### 9.5.3. Message Filter

This filter is the most complicated one. It allows filtering for events, which either process or send specific messages. The messages can be selected based on their C++ class name, message name, various message ids, and a match expression. The expression may refer to the raw data present in the lines marked with 'BS' in the eventlog file.

There are four different message ids to filter, each with different characteristics. The most basic one is the id, which is unique for each constructed message independent of how it was created. The tree id is special because it gets copied over when a message is created by copying (duplicating) another. The encapsulation id is different in that it gives the id of the innermost encapsulated message. Finally, the encapsulation tree id combines the two by providing the innermost encapsulated message's tree id.

### 9.5.4. Tracing Causes/Consequences

The trace filter allows filtering for causes and consequence of a particular event specified by its event number. The cause/consequence relation between two events means that there is a message send/reuse path from the cause event to the consequence event.

If there was a message reuse in the path, then the whole path is considered to be a message reuse itself.



Since computing the causes and consequences in an eventlog file that is far away from the traced event might be a time consuming task, the user can set extra range limits around the traced event. These limits are separate from the range filter due to being relative to the traced event. This means that if you change the traced event, there is no need to change the range parameters. It is strongly recommended that users provide these limits when tracing events to avoid long running operations.

### 9.5.5. Collection Limits

When an in-memory filter is applied to an eventlog, it does not only filter out events, but it also provides automatic discovery for virtual message sends. It means that two events far away, and not directly related to each other, might have a virtual message send (or reuse) between them. Recall that there is a virtual message send (or reuse) between two events if and only if there is a path of message sends (or reuses) connecting the two.

The process of collecting these virtual message dependencies is time consuming and thus has to be limited. There are two options. The first one limits the number of virtual message sends collected per event. The other one limits the depth of cause/consequence chains during collection.

### 9.5.6. Long-Running Operations

Sometimes, computing the filter's result takes a lot of time, especially when tracing causes/consequences without specifying proper range limits in terms of event numbers or simulation times. If you cancel a long running operation, you can go back to the *Filter Dialog* to modify the filter parameters, or simply turn the filter off. To restart drawing, use the refresh button on the toolbar.



Providing a proper range filter is always a good idea to speed up computing the filter's result.

## 9.6. Other Features

Both the Sequence Chart and the Eventlog Table tools can be used as an editor and also as a view. The difference between an editor or a view is quite important because there is only at most one instance of a view of the same kind. It means that even if multiple eventlog files are open in Sequence Chart editors, there is no more than one *Eventlog Table* view shared between them. This single view will automatically display the eventlog file of the active editor. It will also remember its position and state when it switches among editors. For more details on editors and views, and their differences, please refer to the Eclipse documentation.



Despite the name "editor", which is a concept of the Eclipse platform, neither the Sequence Chart, nor the Eventlog Table can be used to actually change the contents of an eventlog file.

It is possible to open the same eventlog file in multiple editors and to navigate to different locations, or use different display modes or filters in a location. Once an eventlog is open in an editor, you can use the *Window | New Editor* to open it again.



Dragging one of the editors from the tabbed pane to the side of the editor's area allows you to interact with the two simultaneously.

### 9.6.1. Settings

There are various settings for both tools which affect the display, such as display modes, content position, filter parameters, etc. These user-specified settings are automatically saved for each file and they are reused whenever the file is revisited. The per file settings are stored under the OMNeT++ workspace, in the directory `.metadata\plugins\org.eclipse.core.resources\projects\<project-name>`.

### 9.6.2. Large File Support

Since an eventlog file might be several Gigabytes, both tools are designed in a way that allows for efficient displaying of such a file without requiring large amounts of physical memory to load it at once. As you navigate through the file, physical memory is filled up with the content lazily. Since it is difficult to reliably identify when the system is getting low on physical memory, it is up to the user to release the allocated memory when needed. This operation, although usually not required, is available from the context menu as *Release Memory*. It does not affect the user interface in any way.

The fact that the eventlog file is loaded lazily and optionally filtered also means that the exact number of lines and events it contains cannot be easily determined. This affects the way scrollbars work in the lazy directions: horizontal for the Sequence Chart and vertical for the Eventlog Table. These scrollbars act as a non-linear approximation in that direction. For the most, the user will be unaware of these approximations unless the file is really small.

### 9.6.3. Viewing a Running Simulation's Results

Even though the simulation kernel keeps the eventlog file open for writing while the simulation is running, it may be open in the OMNeT++ IDE simultaneously. Both tools can be guided by pressing the **END** key to follow the eventlog's end as new content is appended to it. If you pause the simulation in the runtime environment, then after a few seconds the tools will refresh their content and jump to the very end. This process makes it possible to follow the simulation step-by-step on the Sequence Chart.

### 9.6.4. Caveats

Sometimes, drawing the Sequence Chart may take a lot of time. Zooming out too much, for example, might result in slow response times. A dialog might pop up telling the user that a long running eventlog operation is in progress. You can safely cancel these operations at any time you like, or just wait until they finish. To restart the rendering process, simply press the refresh button  on the toolbar. Before refreshing, it is a good idea to revert to some defaults (e.g. default zoom level) or revert the last changes (e.g. navigate back, turn filter off, etc.).



An operation which runs for an unreasonably long time might be a sign of a problem that should be reported.

## 9.7. Examples

This section will guide you through the use of the Sequence Chart and Eventlog Table tools, using example simulations from OMNeT++ and the INET Framework. Before running any of the simulations, make sure that eventlog recording is enabled by adding the line

```
record-eventlog = true
```

in the `omnetpp.ini` file in the simulation's directory. To open the generated eventlog in the OMNeT++ IDE, go to the example's `results` directory in the *Resource Navigator* View, and double-click the log file. By default, the file will open in the Sequence Chart.

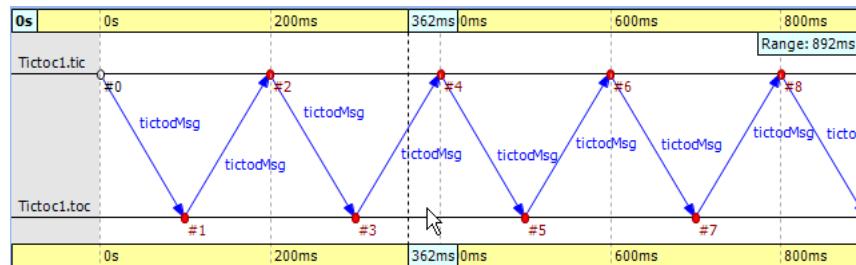


To open the file in the Eventlog Table as editor, right-click the file, and choose the corresponding item from the context menu's *Open With* submenu.

### 9.7.1. Tictoc

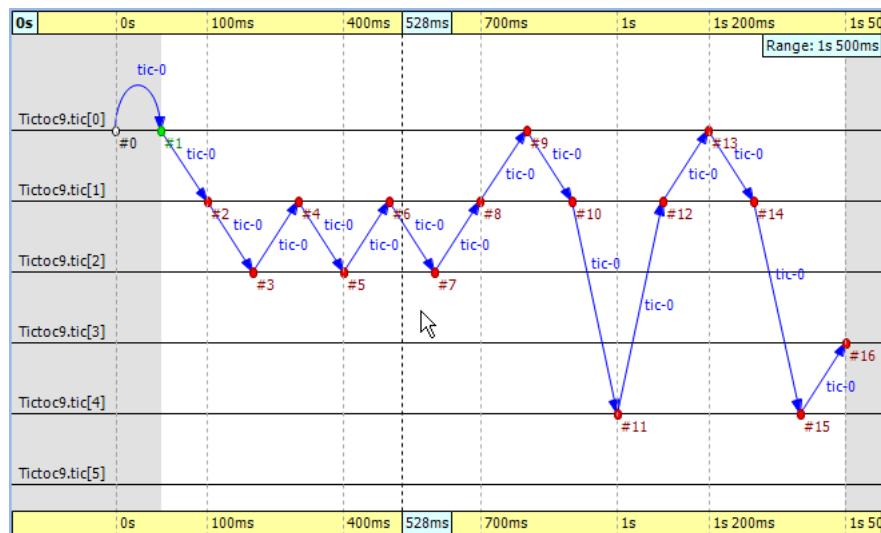
The Tictoc example is available in the OMNeT++ installation under the directory `samples/tictoc`. Tictoc is the most basic example in this chapter and it provides a quick overview on how to use and understand the Sequence Chart.

Start the simulation and choose the simplest configuration, 'Tictoc1', which specifies only two nodes called 'tic' and 'toc.' During initialization, one of the nodes will send a message to the other. From then on, every time a node receives the message, it will simply send it back. This process continues until you stop the simulation. In Figure 9.15, "Tictoc with two nodes" you can see how this is represented on a Sequence Chart. The two horizontal black lines correspond to the two nodes and are labeled 'tic' and 'toc.' The red circles represent events and the blue arrows represent message sends. It is easy to see that all message sends take 100 milliseconds and that the first sender is the node 'tic.'



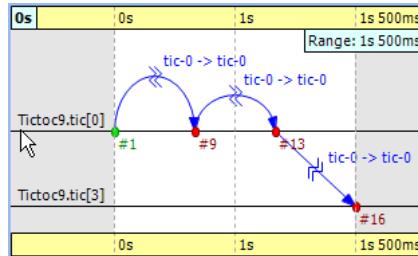
**Figure 9.15. Tictoc with two nodes**

In the next Tictoc example, there are six nodes tossing a message around until it reaches its destination. To generate the eventlog file, restart the simulation and choose the configuration 'Tictoc9'. In Figure 9.16, "Tictoc with six nodes" you can see how the message goes from one node to another, starting from node '0' and passing through it twice more, until it finally reaches its destination, node '3.' The chart also shows that this example, unlike the previous one, starts with a self-message instead of immediately sending a message from initialize to another node.



**Figure 9.16. Tictoc with six nodes**

Let us demonstrate with this simple example how filtering works with the Sequence Chart. Open the *Filter Dialog* with the toolbar button  and put a checkmark for node '0' and '3' on the *Module filter|by name* panel, and apply it. The chart now displays only two axes that correspond to the two selected nodes. Note that the arrows on this figure are decorated with zigzags, meaning that they represent a sequence of message sends. Such arrows will be called virtual message sends in the rest of this chapter. The first two arrows show the message returning to node '0' at event #9 and event #13, and the third shows that it reaches the destination at event #16. The events where the message was in between are filtered out.



**Figure 9.17. Filtering for node '0' and '3'**

### 9.7.2. FIFO

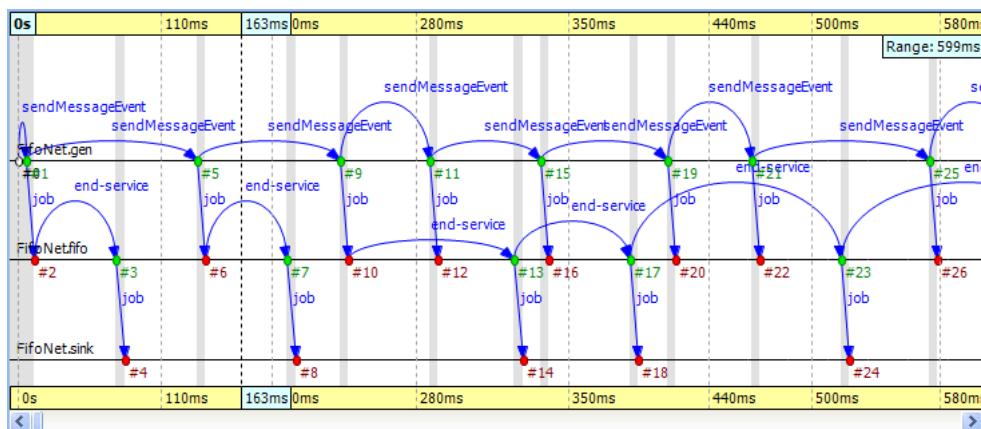
The FIFO example is available in the OMNeT++ installation under the directory `samples/fifo`. The FIFO is an important example because it uses a queue, which is an essential part of discrete event simulations and introduces the notion of message reuses.

When you start the simulation, choose the configuration 'low job arrival rate' and let it run for a while. In Figure 9.18, "The FIFO example" you can see three modules: a 'source', a 'queue', and a 'sink.' The simulation starts with a self-message and then the generator sends the first message to the queue at event #1. It is immediately obvious that the message stays in the queue for a certain period of time, between event #2 and event #3.



When you select one event and hover with the mouse above the other, the Sequence Chart will show the length of this time period in a tooltip.

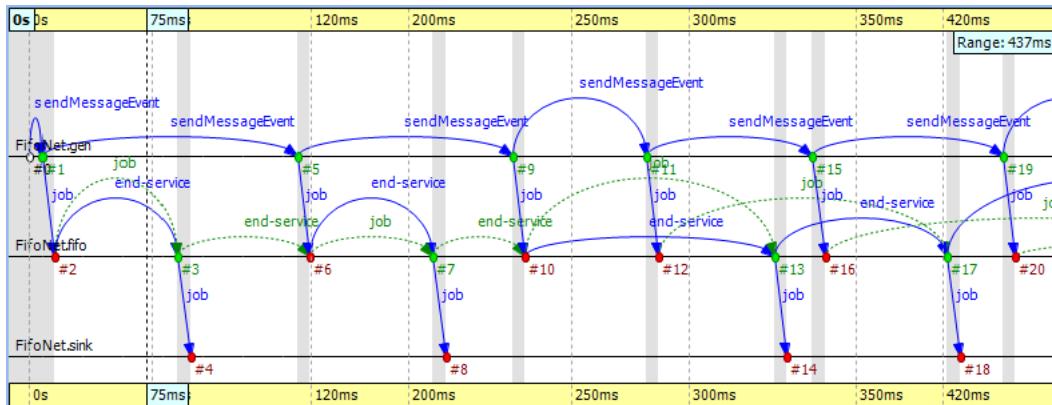
Finally, the message is sent to the 'sink' where it is deleted at event #4.



**Figure 9.18. The FIFO example**

Something interesting happens at event #12 where the incoming message suddenly disappears. It seems like the queue does not send the message out. Actually, what

happens is that the queue enqueues the job because it is busy serving the message received at event #10. Since this queue is a FIFO, it will send out the first message at event #13. To see how this happens, turn on *Show Reuse Messages* from the context menu; the result is shown in Figure 9.19, “Showing reuse messages”. It displays a couple of green dotted arrows, one of which starts at event #12 and arrives at event #17. This is a reuse arrow; it means that the message sent out from the queue at event #17 is the same as the one received and enqueued at event #12. Note that the service of this message actually begins at event #13, which is the moment that the queue becomes free after it completes the job received at event #10.

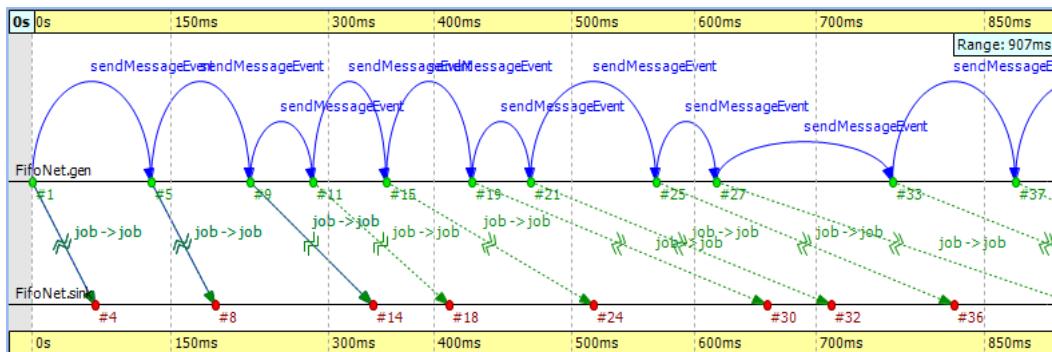


**Figure 9.19. Showing reuse messages**

Another type of message reuse is portrayed with the arrow from event #3 to event #6. The arrow shows that the queue reuses the same timer message instead of creating a new one each time.

 Whenever you see a reuse arrow, it means that the underlying implementation remembers the message between the two events. It might be stored in a pointer variable, a queue, or some other data structure.

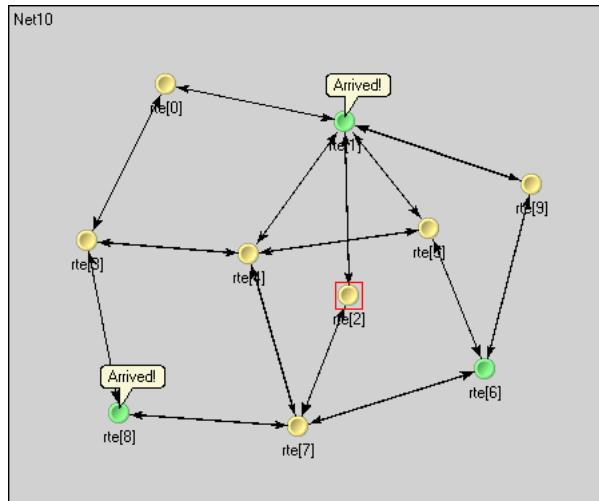
The last part of this example is about filtering out the queue from the chart. Open the *Filter Dialog*, select 'sink' and 'source' on the *Module filter|by NED type* panel, and apply the change in settings. If you look at the result in Figure 9.20, “Filtering the queue”, you will see zigzag arrows going from the 'source' to the 'sink.' These arrows show that a message is being sent through the queue from 'source' to 'sink.' The first two arrows do not overlap in simulation time, which means the queue did not have more than one message during that time. The third and fourth arrows do overlap because the fourth job reached the queue while it was busy with the third one. Scrolling forward you can find other places where the queue becomes empty and the arrows do not overlap.



**Figure 9.20. Filtering the queue**

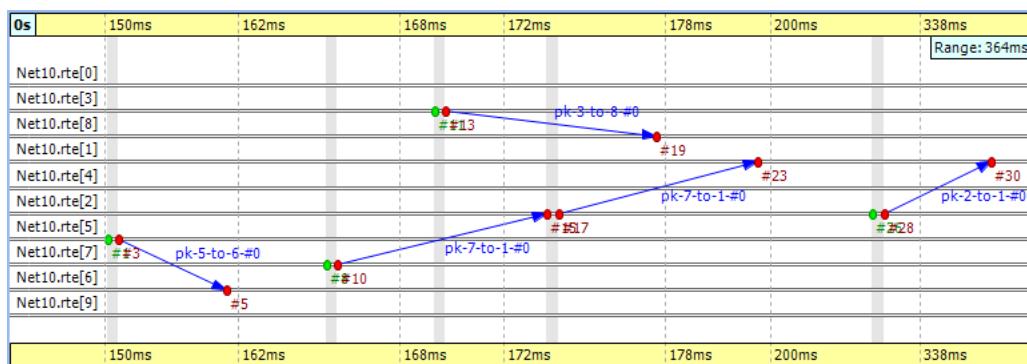
### 9.7.3. Routing

The Routing example is available in the OMNeT++ installation under the directory samples/routing. The predefined configuration called 'Net10' specifies a network with 10 nodes with each node having an application, a few queues and a routing module inside. Three preselected nodes, namely the node '1,' '6,' and '8' are destinations, while all nodes are message sources. The routing module uses the shortest path algorithm to find the route to the destination. The goal in this example is to create a sequence chart that shows messages which travel simultaneously from multiple sources to their destinations.



**Figure 9.21. Network with 10 nodes**

Since we do not care about the details regarding what happens within nodes, we can simply turn on filtering for the NED type 'node.Node.' The chart will have 10 axes with each axis drawn as two parallel solid black lines close to each other. These are the compound modules that represent the nodes in the network. So far events could be directly drawn on the simple module's axis where they occurred, but now they will be drawn on the compound module's axis of their ancestor.



**Figure 9.22. Filtering for nodes**

To reduce clutter, the chart will automatically omit events which are internal to a compound module. An event is internal to a compound module if it only processes a message from, and sends out messages to, other modules inside the compound module.

If you look at Figure 9.22, "Filtering for nodes" you will see a message going from node '7' at event #10 to node '1' at event #23. This message stays in node '2' between

event #15 and event #17. The gray background area between them means that zero simulation time has elapsed (i.e. the model does not account for processing time inside the network nodes).



This model contains both finite propagation delay and transmission time; arrows in the sequence chart correspond to the interval between the start of the transmission and the end of the reception.

This example also demonstrates message detail recording configured by

```
eventlog-message-detail-pattern = Packet:declaredOn(Packet)
```

in the INI file. The example in Figure 9.23, "Message detail tooltip" shows the tooltip presented for the second message send between event #17 and event #23.

```

Sending (Packet) pk-7-to-1-#0 (#17->#23) dt = 8ms 292us (id = 41)
class Packet {
    int srcAddr = 7
    int destAddr = 1
    int hopCount = 2
}

```

Press 'F2' for focus.

**Figure 9.23. Message detail tooltip**

It is very easy to find another message on the chart that goes through the network parallel in simulation time. The one sent from node '3' at event #13 to node '8' arriving at event #19 is such a message.

#### 9.7.4. Wireless

The Wireless example is available in the INET Framework under the directory examples/adhoc/ieee80211. The predefined configuration called 'Config1' specifies two mobile hosts moving around on the playground and communicating via the IEEE 802.11 wireless protocol. The network devices are configured for ad-hoc mode and the transmitter power is set so that hosts can move out of range. One of the hosts is continuously pinging the other.

In this section, we will explore the protocol's MAC layer, using two sequence charts. The first chart will show a successful ping message being sent through the wireless channel. The second chart will show ping messages getting lost and being continuously re-sent.

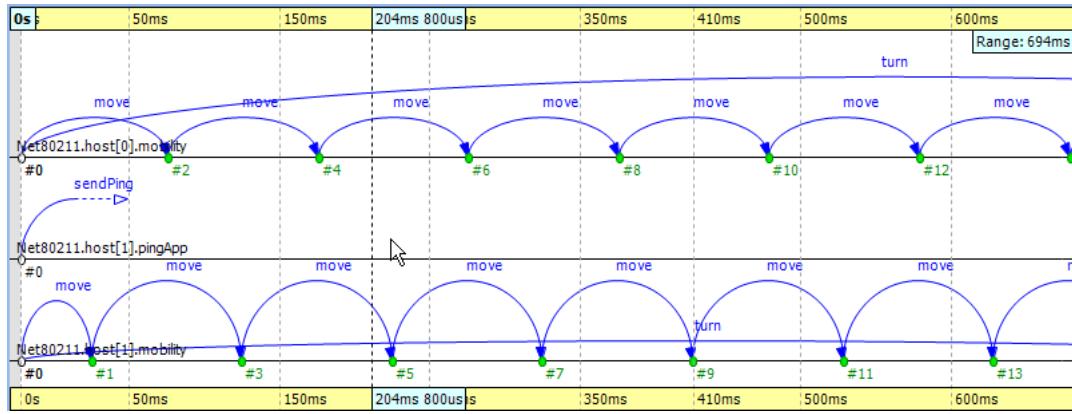
We also would like to record some message details during the simulation. To perform that function, comment out the following line from `omnetpp.ini`:

```
eventlog-message-detail-pattern = *:(not declaredOn(cMessage) and not
                                 declaredOn(cNamedObject) and not declaredOn(cObject))
```

To generate the eventlog file, start the simulation environment and choose the configuration 'host1 pinging host0.' Run the simulation in fast mode until about event #5000.

#### Preparing the Result

When you open the Sequence Chart, it will show a couple of self-messages named 'move' being scheduled regularly. These are self-messages that control the movement of the hosts on the playground. There is an axis labeled 'pingApp,' which starts with a 'sendPing' message that is processed in an event far away on the chart. This is indicated by a split arrow.

**Figure 9.24. The beginning**

You might notice that there are only three axes in Figure 9.24, “The beginning” even though the simulation model clearly contains more simple modules. This is because the Sequence Chart displays the first few events by default and in this scenario, they all happen to be within those modules. If you scroll forward or zoom out, new axes will be added automatically as needed.

For this example, ignore the 'move' messages and focus on the MAC layer instead. To begin with, open the *Filter Dialog*, select 'Ieee80211Mac' and 'Ieee80211Radio' on the *Module filter|by NED type* panel, and apply the selected changes. The chart will have four axes, two for the MAC and two for the radio simple modules.

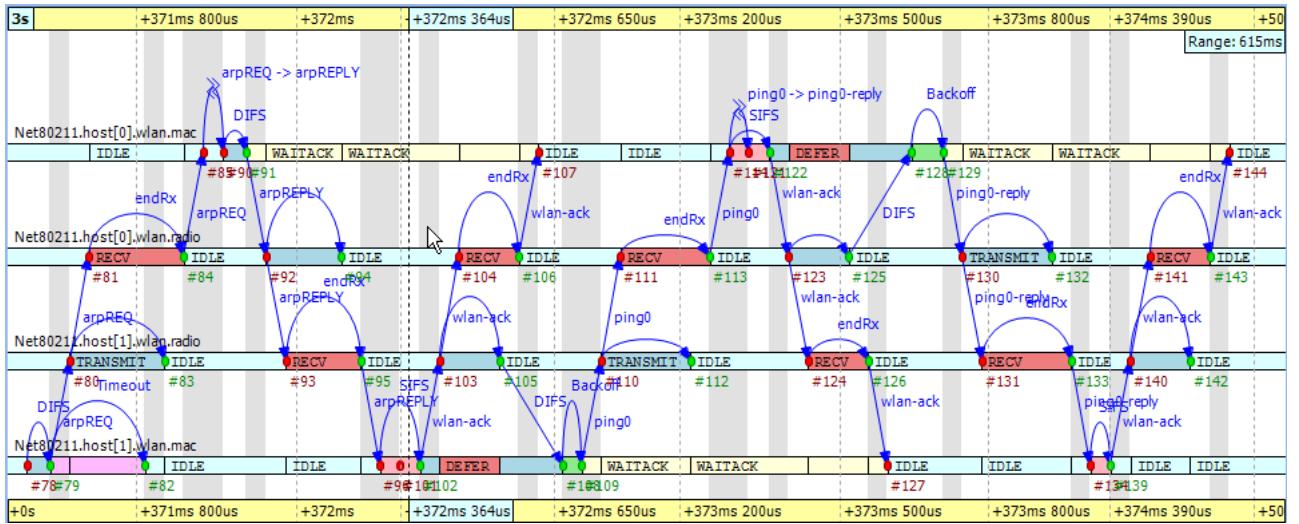
The next step is to attach vector data to these axes. Open the context menu for each axis by clicking on them one by one and select the *Attach Vector to Axis* submenu. Accept the vector file offered by default. Then, choose the vector 'mac:State' for the MAC modules and 'mac:RadioState' for the radio modules. You will have to edit the filter in the vector selection dialog (i.e. delete the last segment) for the radio modules because at the moment the radio state is recorded by the MAC module, so the default filter will not be right. When this step is completed, the chart should display four thick colored bars as module axes. The colors and labels on the bars specify the state of the corresponding state machine at the given simulation time.

To aid comprehension, you might want to manually reorder the axis, so that the radio modules are put next to each other. Use the button on the toolbar to switch to manual ordering. With a little zooming and scrolling, you should be able to fit the first message exchange between the two hosts into the window.

### Successful Ping

The first message sent by 'host1' is not a ping request but an ARP request. The processing of this message in 'host0' generates the corresponding ARP reply. This is shown by the zigzag arrow between event #85 and event #90. The reply goes back to 'host1,' which then sends a WLAN acknowledge in return. In this process, 'host1' discovers the MAC address of 'host0' based on its IP address.

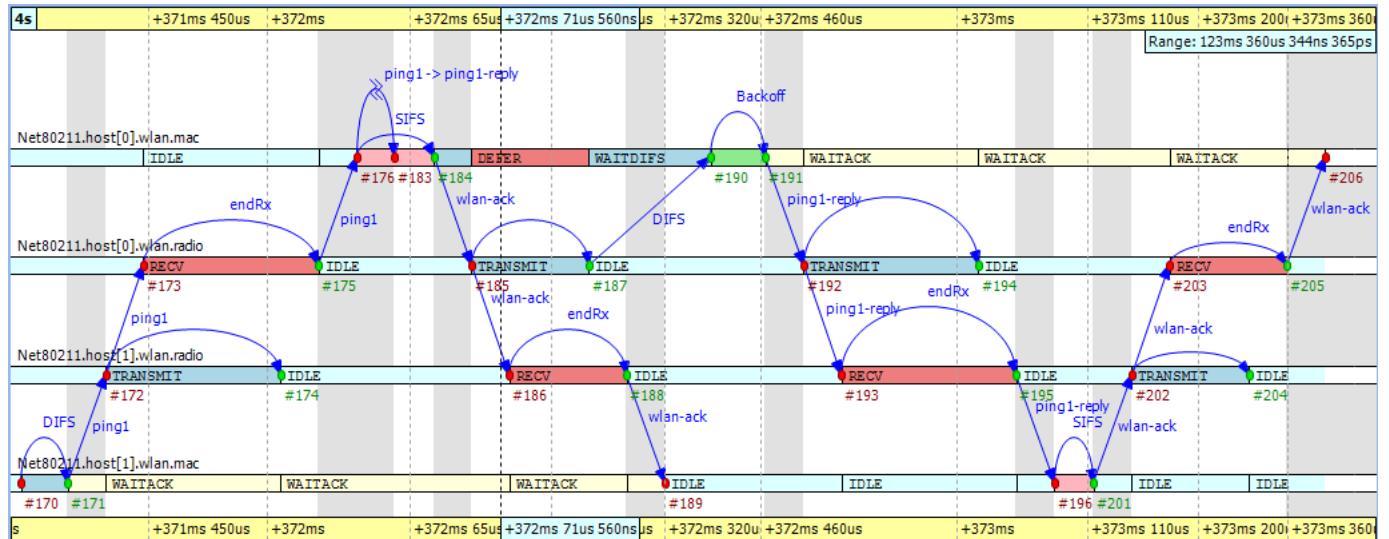
## Sequence Charts



**Figure 9.25. Discovering the MAC address**

The send procedure for the first ping message starts at event #105 in 'host1' and finishes by receiving the acknowledge at event #127. The ping reply send procedure starts at event #125 in 'host0' and finishes by receiving the WLAN acknowledge at event #144. If you scroll forward, you can see as in Figure 9.26, "The second ping procedure" the second complete successful ping procedure between event #170 and event #206. To focus on the second successful ping message exchange, open the *Filter Dialog* and enter these numbers in the range filter.

Timing is critical in a protocol implementation, so we will take a look at it using the Sequence Chart. The first self message represents the fact that the MAC module listens to the radio for a DIFS period before sending the message out. The message send from event #171 to event #172 occurs in zero simulation time as indicated by the gray background. It represents the moment when the MAC module decides to send the ping request down to its radio module. The back-off procedure was skipped for this message because there was no transmission during the DIFS period. If you look at event #172 and event #173, you will see how the message propagates through the air from 'radio1' to 'radio0.' This finite amount of time is calculated from the physical distance of the two modules and the speed of light. In addition, by looking at event #172 and event #174, you will notice that the transmission time is not zero. This time interval is calculated from the message's length and the radio module's bitrate.



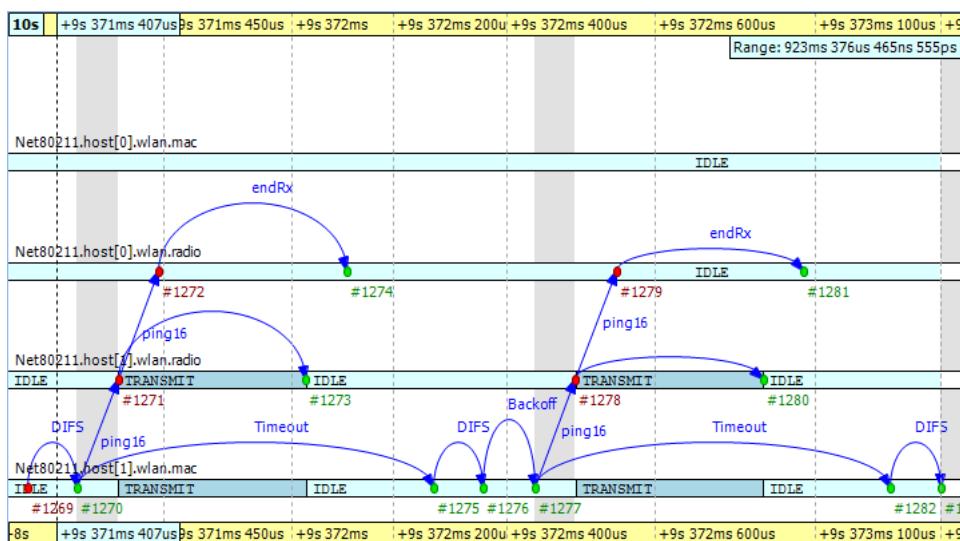
**Figure 9.26. The second ping procedure**

Another interesting fact seen in the figure is that the higher level protocol layers do not add delay for generating the ping reply message in 'host0' between event #176 and event #183. The MAC layer procedure ends with sending back a WLAN acknowledge after waiting a SIFS period.

Finally, you can get a quick overview of the relative timings of the IEEE 802.11 protocol by switching to linear timeline mode. Use the button  on the toolbar and notice how the figure changes dramatically. You might need to scroll and zoom in or out to see the details. This shows the usefulness of the nonlinear timeline mode.

### Unsuccessful Ping

To see how the chart looks when the ping messages get lost in the air, first turn off range filtering. Then, go to event #1269 by selecting the *Goto Event* option from the *Eventlog Table* View's context menu. In Figure 9.27, "Ping messages get lost" you can see how the receiver radio does not send up the incoming message to its MAC layer due to the signal level being too low. This actually happens at event #1274 in 'host0.' Shortly thereafter, the transmitter MAC layer in 'host1' receives the timeout message at event #1275, and starts the backoff procedure before resending the very same ping message. This process goes on with statistically increasing backoff time intervals until event #1317. Finally, the maximum number of retries is reached and the message is dropped.



**Figure 9.27. Ping messages get lost**

The chart also shows that during the unsuccessful ping period, there are no events occurring in the MAC layer of 'host0' and it is continuously in 'IDLE' state.

# Chapter 10. Analyzing the Results

## 10.1. Overview

Analyzing the simulation result is a lengthy and time consuming process. The result of the simulation is recorded as scalar values, vector values and histograms. The user then applies statistical methods to extract the relevant information and to draw a conclusion. This process may include several steps. Usually you need to filter and transform the data, and chart the result. Automation is very important here. You do not want to repeat the steps of recreating charts every time you rerun simulations.

In OMNeT++ 4.x, the statistical analysis tool is integrated into the Eclipse environment. Your settings (i.e. your recipe for finding results from the raw data) will be recorded in analysis files (.anf) and will become instantly reproducible. This means that all processing and charts are stored as datasets; for example, if simulations need to be rerun due to a model bug or misconfiguration, existing charts need not be recreated all over again. Simply replacing the old result files with the new ones will result in the charts being automatically displayed with the new data.

When creating an analysis, the user first selects the input of the analysis by specifying file names or file name patterns (e.g. "adhoc-\*.\*.vec"). Data of interest can be selected into datasets using additional pattern rules. The user can define datasets by adding various processing, filtering and charting steps; all using the GUI. Data in result files are tagged with meta information. Experiment, measurement and replication labels are added to the result files to make the filtering process easy. It is possible to create very sophisticated filtering rules (e.g. all 802.11 retry counts of host[5..10] in experiment X, averaged over replications). In addition, datasets can use other datasets as their input so datasets can build on each other.

## 10.2. Creating Analysis Files

To create a new analysis file, choose *File | New | Analysis File* from the menu. Select the folder for the new file and enter the file name. Press *Finish* to create and open an empty analysis file.



Figure 10.1. New Analysis File dialog

There is a quick way to create an analysis file for a result file. Just double-click on the result file in the *Project Explorer View* to open the *New Analysis File* dialog. The folder and file name is prefilled according to the location and name of the result file. Press *Finish* to create a new analysis file containing the vector and scalar files whose names correspond to the result file. If the name of the result file contains a numeric suffix (e.g. aloha-10.vec), then all files with the same prefix will be added to the analysis file (i.e. aloha-\* .vec and aloha-\* .sca).



If the analysis file already exists, double-clicking on the result file will open it.

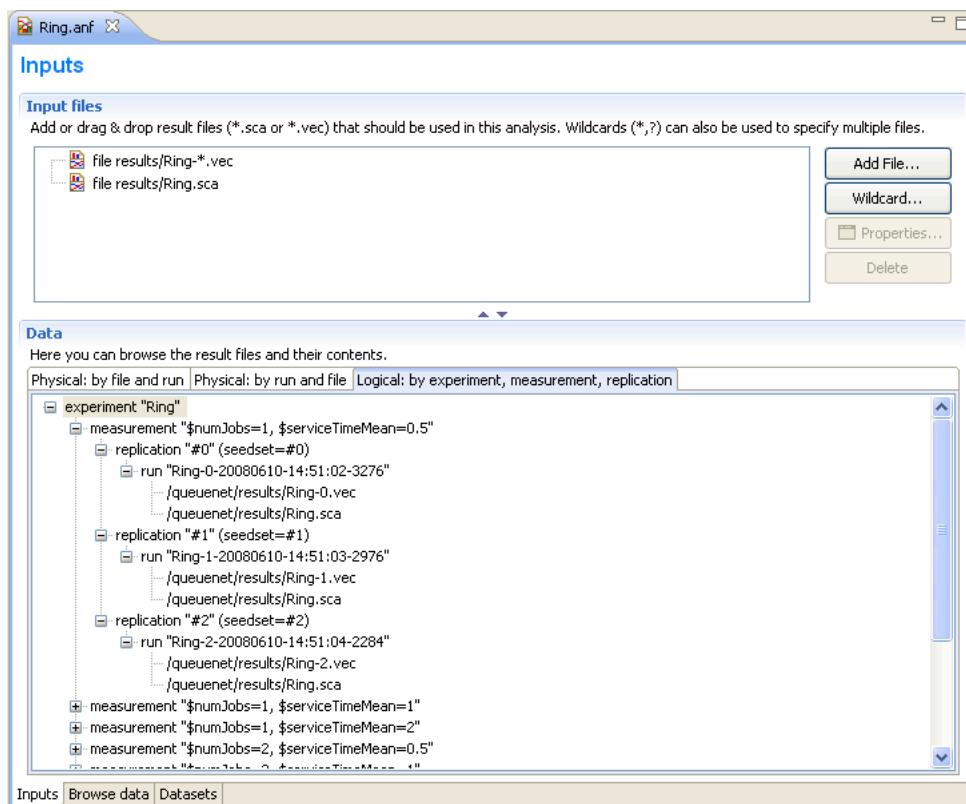
## 10.3. Using the Analysis Editor

The Analysis Editor is implemented as a multi-page editor. What the editor edits is the "recipe": what result files to take as inputs, what data to select from them, what (optional) processing steps to apply, and what kind of charts to create from them. The pages (tabs) of the editor roughly correspond to these steps.

### 10.3.1. Input Files

#### Selecting input files

The first page displays the result files that serve as input for the analysis. The upper half specifies what files to select using explicit filenames or wildcards. New input files can be added to the analysis by dragging vector and scalar files from the *Project Explorer View*, or by opening dialogs with the *Add File...* or *Wildcard...* buttons. If the file name starts with '/' it is interpreted relative to the workspace root; otherwise, it is relative to the folder of the analysis file.



**Figure 10.2. Specifying input files for data analysis**

The input files are loaded when the analysis file is opened. When the file changes on the disk, it is reloaded automatically when the workspace is refreshed (Eclipse refreshes the workspace automatically if the *General|Workspace|Refresh automatically* option is turned on in the Preferences). Vector files are not loaded directly; instead, an index file is created and the vector attributes (name, module, run, statistics, etc.) are loaded from the index file. The index files are generated during the simulation, but can be safely deleted without loss of information. If the index file is missing or the vector file was modified, the IDE rebuilds the index in the background.



The *Progress View* displays the progress of the indexing process.

The lower half shows what files matched the input specification and what runs they contain. Note that OMNeT++ 4.x result files contain a unique run ID and several metadata annotations in addition to the actual recorded data. The third tree organizes simulation runs according to their experiment-measurement-replication labels.

The underlying assumption is that users will organize their simulation-based research into various *experiments*. An experiment will consist of several *measurements*, which are typically (but not necessarily) simulations done with the same model but with different parameter settings (i.e. the user will explore the parameter space with several simulation runs). To gain statistical confidence in the results, each measurement may be repeated several times with different random number seeds. It is easy to set up such scenarios with the improved INI files of OMNeT++ 4.x. Then, the experiment-measurement-replication labels will be assigned automatically (Note: please refer to the chapter "Configuring and Running Simulations" in the manual for more discussion).

### Browsing input

The second page of the Analysis editor displays results (vectors, scalars and histograms) from all files in tables and lets the user browse them. Results can be sorted and filtered. Simple filtering is possible with combo boxes, or when that is not enough, the user can write arbitrarily complex filters using a generic pattern-matching expression language. Selected or filtered data can be immediately plotted, or remembered in named datasets for further processing.



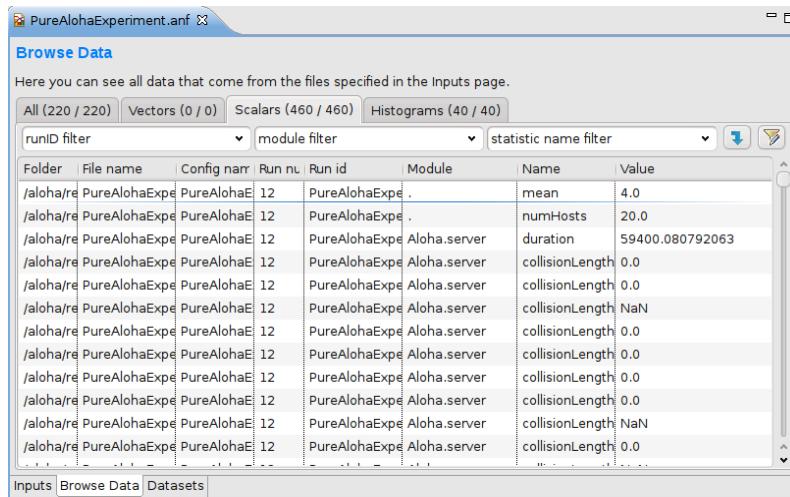
You can switch between the *All*, *Vectors*, *Scalars* and *Histograms* pages using the underlined keys (**Alt+KEY** combination) or the **Ctrl+PgUp** and **Ctrl+PgDown** keys.

Pressing the *Advanced* button switches to advanced filter mode. In the text field, you can enter a complex filter pattern.



You can easily display the data of a selected file, run, experiment, measurement or replication if you double-click on the required tree node in the lower part of the *Inputs* page. It sets the appropriate filter and shows the data on the *Browse Data* page.

If you right-click on a table cell and select the *Set filter: ...* action from the menu, you can set the value of that cell as the filter expression.



**Figure 10.3. Browsing vector and scalar data generated by the simulation**

To hide or show table columns, open *Choose table columns...* from the context menu and select the columns to be displayed. The settings are persistent and applied in each subsequently opened editor. The table rows can be sorted by clicking on the column name.

You can display the selected data items on a chart. To open the chart, choose *Plot* from the context menu (double-click also works for single data lines). The opened chart is not added automatically to the analysis file, so you can explore the data by opening the chart this way and closing the chart page without making the editor "dirty."

The selected vector's data can also be displayed in a table. Make sure that the *Output Vector View* is opened. If it is not open, you can open it from the context menu (*Show Output Vector View*). If you select a vector in the editor, the view will display its content.

You can create a dataset from the selected result items. Select *Add Filter Expression to Dataset...* if you want to add all items displayed in the table. Select *Add Filter Selected Data to Dataset...* if you want to add the selected items only. You can add the items to an existing dataset, or you can create a new dataset in the opening dialog.



You can switch between the *Inputs*, *Browse Data* and *Dataset* pages using the **Alt+PgUp** and **Alt+PgDown** keys.

### Filter expressions

A filter expression is composed of atomic patterns with the AND, OR, NOT operators. An atomic pattern filters for the value of a field of the result item and has the form `<field_name>(<pattern>)`. The following table shows the valid field names. You can omit the name field and simply use the name pattern as a filter expression. It must be quoted if it contains whitespace or parentheses.

Field	Description
name	the name of the scalar, vector or histogram
module	the name of the module
file	the file of the result item
run	the run identifier
attr: <i>name</i>	the value of the run attribute named <i>name</i> , e.g. attr:experiment
param: <i>name</i>	the value of the module parameter named <i>name</i>

In the pattern specifying the field value, you can use the following shortcuts:

Pattern	Description
?	matches any character except '!'
*	matches zero or more characters except '.'
**	matches zero or more characters (any character)
{a-z}	matches a character in range a-z
{ ^ a-z}	matches a character not in range a-z
{32..255}	any number (i.e. sequence of digits) in range 32..255 (e.g. "99")
[32..255]	any number in square brackets in range 32..255 (e.g. "[99]")
\	takes away the special meaning of the subsequent character



Content Assist is available in the text fields where you can enter a filter expression. Press **Ctrl+Space** to get a list of appropriate suggestions related to the expression at the cursor position.

## Examples

```
"queuing time"
```

filters for result items named *queuing time*.

```
module(**.sink) AND (name("queuing time") OR  
name("transmission time"))
```

results in the *queuing times* and *transmission times* that are written by modules named *sink*.

### 10.3.2. Datasets

#### Overview

The third page displays the datasets and charts created during the analysis. Datasets describe a set of input data, the processing applied to them and the charts. The dataset is displayed as a tree of processing steps and charts. There are nodes for adding and discarding data, applying processing to vectors and scalars, selecting the operands of the operations and content of charts, and for creating charts.

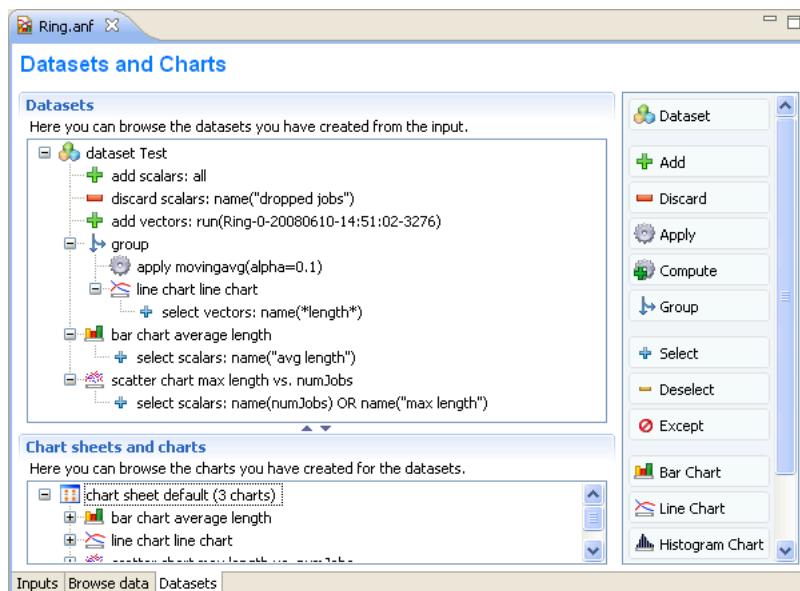


Figure 10.4. Defining datasets to be analyzed



You can browse the dataset's content in the *Dataset View*. Open the view by selecting *Show Dataset View* from the context menu. Select a chart to display its content or another node to display the content of the dataset after processing is applied.

## Editing datasets

The usual tree editing operations work on the Dataset tree. New elements can be added by dragging elements from the palette on the right to an appropriate place on the tree. Alternatively, you can select the parent node and press the button on the toolbar. An existing element can be edited by selecting the element and editing its properties on the property sheet, or by opening an item-specific edit dialog by choosing *Properties...* from the context menu.



Datasets can be opened on a separate page by double-clicking on them. It is easier to edit the tree on this page. Double-clicking a chart node will display the chart.

Computations can be applied to the data by adding *Apply to Vectors/Compute Vectors/Compute Scalars* nodes to the dataset. The input of the computations can be selected by adding *Select/Deselect children* to the processing node. By default, the computation input is the whole content of the dataset at the processing node. Details of the computations are described in the next sections.

Processing steps within a *Group* node only affect the group. This way, you can create branches in the dataset. To group a range of sibling nodes, select them and choose *Group* from the context menu. To remove the grouping, select the *Group* node and choose *Ungroup*.

Charts can be inserted to display data. The data items to be displayed can be selected by adding *Select/Deselect children* to the chart node. By default, the chart displays all data in the dataset at its position. You can modify the content of the chart by adding *Select and Deselect children* to it. Charts can be fully customized including setting titles, colors, adding legends, grid lines, etc. See the *Charts* section for details.

## Computing Vectors

Both *Compute Vectors* and *Apply to Vectors* nodes compute new vectors from other vectors. The difference between them is that *Apply to Vectors* will remove its input from the dataset, while *Compute* keeps the original data, too.

Table 10.1, “Processing operations” contains the list of available operations on vectors.

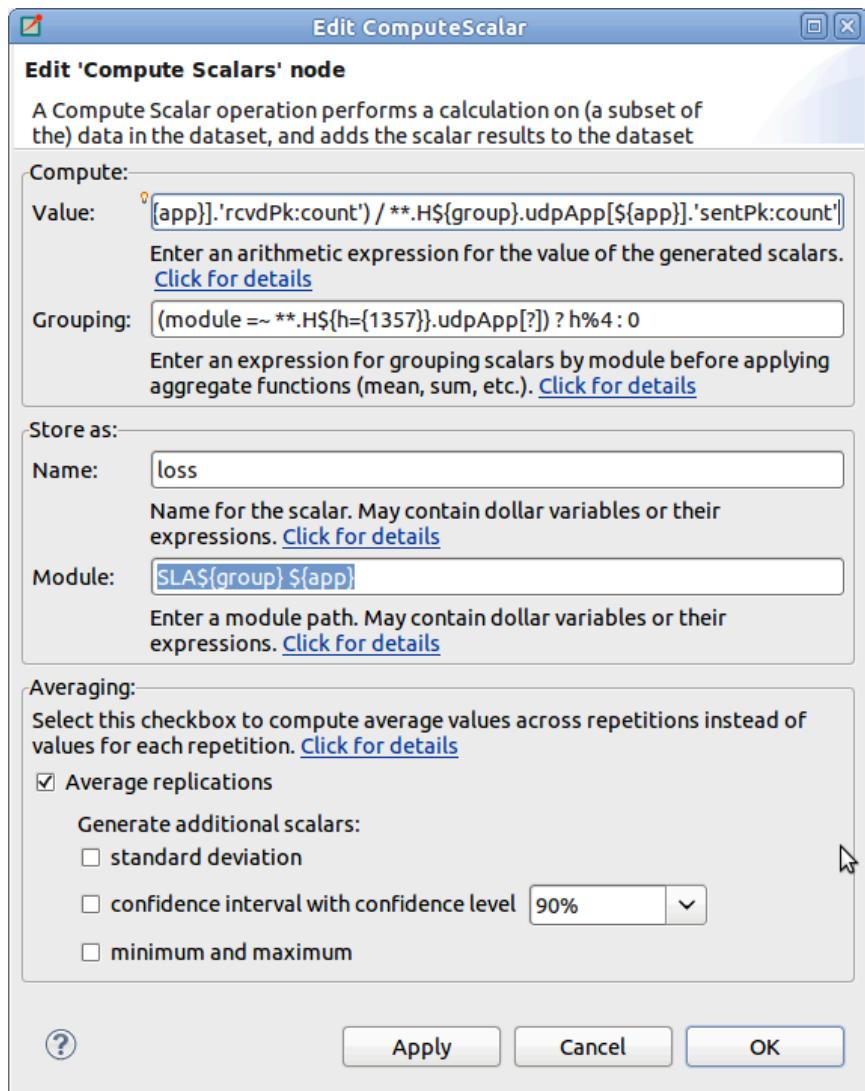
Name	Description
scatter	Create scatter plot dataset. The first two arguments identifies the scalar selected for the X axis. Additional arguments identify the iso attributes; they are (module, scalar) pairs, or names of run attributes.
add	Adds a constant to the input: $y^{out}_k = y_k + c$
compare	Compares value against a threshold, and optionally replaces it with a constant
crop	Discards values outside the [t1, t2] interval
difference	Substracts the previous value from every value: $y^{out}_k = y_k - y_{k-1}$
diffquot	Calculates the difference quotient of every value and the subsequent one: $y^{out}_k = (y_{k+1} - y_k) / (t_{k+1} - t_k)$

Name	Description
divide-by	Divides input by a constant: $y^{out}_k = y_k / a$
divtime	Divides input by the current time: $y^{out}_k = y_k / t_k$
expression	Evaluates an arbitrary expression. Use t for time, y for value, and tprev, yprev for the previous values.
integrate	Integrates the input as a step function (sample-hold or backward-sample-hold) or with linear interpolation
linertrend	Adds linear component to input series: $y^{out}_k = y_k + a * t_k$
mean	Calculates mean on (0,t)
modulo	Computes input modulo a constant: $y^{out}_k = y_k \% a$
movingavg	Applies the exponentially weighted moving average filter: $y^{out}_k = y^{out}_{k-1} + alpha * (y_k - y^{out}_{k-1})$
multiply-by	Multiplies input by a constant: $y^{out}_k = a * y_k$
nop	Does nothing
removerepeats	Removes repeated y values
slidingwinavg	Replaces every value with the mean of values in the window: $y^{out}_k = \text{SUM}(y_i, i=k-winsize+1..k)/winsize$
subtractfirstval	Subtract the first value from every subsequent values: $y^{out}_k = y_k - y[0]$
sum	Sums up values: $y^{out}_k = \text{SUM}(y_i, i=0..k)$
timeavg	Calculates the time average of the input (integral divided by time)
timediff	Returns the difference in time between this and the previous value: $y^{out}_k = t_k - t_{k-1}$
timeshift	Shifts the input series in time by a constant: $t^{out}_k = t_k + dt$
timetoserial	Replaces time values with their index: $t^{out}_k = k$
timewinavg	Calculates time average: replaces input values in every `winSize' interval with their mean. $t^{out}_k = k * winSize$ $y^{out}_k = \text{average of } y \text{ values in the } [(k-1)*winSize, k*winSize] \text{ interval}$
winavg	Calculates batched average: replaces every `winsize' input values with their mean. Time is the time of the first value in the batch.

**Table 10.1. Processing operations**

### Computing Scalars

The Compute Scalars dataset node adds new scalars to the dataset whose values are computed from other statistics in the dataset. The input of the computation can be restricted by adding Select/Deselect nodes under it.



**Figure 10.5. Edit 'Compute Scalars' dialog**

In the *Properties* dialog of the *Compute Scalars* node, you can set the name and module of the generated scalars, and the expression that computes their values. You can also enter a grouping expression, and set flags to average the values across replications. Content Assist (**Ctrl+SPACE**) is available for the *Value* field, it can propose statistic and function names.

The content of the dialog is validated after each keystroke, and errors are displayed as small icons next to the edit field. Hovering over the error icon shows the error message. However, not all errors can be detected statically, in the dialog. If an error occurs while performing the computation, then an error marker is added to the analysis file and to the corresponding dataset node in the Analysis editor. You can view the error in the *Problems View*, and double-clicking the problem item navigates back to the *Compute Scalars* node.

When a *Compute Scalars* node in the dataset is evaluated, computations will use the fields roughly in the order they appear in the dialog. First, grouping takes place (by simulation run, and then by the optional grouping expression); then values are computed; then values are stored by the given name and module; and finally, averaging across simulation runs takes place.

Explanation the dialog fields:

**Value.** This is an arithmetic expression for the value of the generated scalar(s). You can use the values of existing scalars, various properties of existing vectors and histograms (mean, min, max, etc), normal and aggregate functions (mean, min, max, etc), and the usual set of operators.

To refer to the value of a scalar, simply use the scalar name (e.g. `pkLoss`), or enclose it with apostrophes if the name of the scalar contains special characters (e.g. `'rcvdPk:count'`.) If there are several scalars with that name in the input dataset, usually recorded by different modules or in different runs, then the computation will be performed on each. The scalar name cannot contain wildcards or dollar variables (see later.)

When necessary, you can qualify the scalar name with a module name pattern that will be matched against the full paths of modules. The same pattern syntax is used as in ini files and in other parts of the Analysis Tool (Quick reminder: use `*` and `**` for wildcards, `{5..10}` for numeric range, `[5..10]` for module index range). If multiple scalars match the qualified name, the expression will be computed for each. If there are several such patterns in the expression, then the computation will be performed on their Cartesian product.

If the expression mentions several unqualified scalars (i.e. without module pattern), they are expected to come from the same module. For example, if your expression is `foo+bar` but the `foo` and `bar` scalars have been recorded by different modules, the result will be empty.

The iteration can be restricted by binding some part of the module name to variables, and use those variables in other patterns. The  `${x=<pattern>}`  syntax in a module name pattern binds the part of the module name matched to *pattern* to a variable named *x*. These variables can be referred as  `${x}`  in other patterns. The  `${...}`  syntax also allows you to embed expressions (e.g.  `${x+1}` ) into the pattern.

To make use of vectors and histograms in the computation, you can use the `count()`, `mean()`, `min()`, `max()`, `stddev()` and `variance()` functions to access their properties (e.g. `count(**.mac.pkDrop)` or `max('end-to-end delay')`).

The following functions can be applied to scalars or to an expression that yields a scalar value: `count()`, `mean()`, `min()`, `max()`, `stddev()`, `variance()`. These aggregate functions produce one value for a group of scalars instead of one for each scalar. By default, each scalar belongs to the same group, but it is possible to group them by module name (see Grouping). Aggregate functions cannot cross simulation run boundaries, e.g. they cannot be used to compute average over all replications of the same configuration; use the *Average replications* checkbox for that.

**Grouping.** Scalars can be grouped by module or value before the value computation, and you can enter a grouping expression here. Grouping is the most useful when you want to use aggregate functions (`count()`, `mean()`, etc.) in the value expression.

The grouping expression is evaluated for each statistic in the input dataset, and the resulting value denotes the statistic's group. For example, if the expression produces 0 for some statistics and 1 for others, there will be two groups. Aggregate functions (`count()`, `mean()`, etc.) are performed on each group independently.

In the grouping expression, you can refer to the name, module and values of the statistic (module, name and value; the latter is only meaningful on scalars, and produces *NaN* for histograms and vectors), and to attributes of the simulation run (run, configname, runnumber, experiment, measurement, replication, iteration variables of the ini file, etc.) However, note that run attributes are not as useful as they would appear, because grouping only takes place within simulation runs, you cannot join data from several simulation runs into one group this way.

Often you want derive the group identifier from some part of the module name. A useful tool for that is the pattern matching operator (`=~`) combined with conditionals (`? :`). The expression `<str> =~ <pat>` matches the string `str` with the pattern `pat`. If there is no match, the value of the expression is `false`, otherwise the input string `str` (which counts as true). The useful bit is that you can bind parts of the matching string to variables with the  `${x=<pattern>}`  syntax in the pattern (see above), and you can use those variables later in the grouping expression, and also in the *Value*, *Name* and *Module* fields.

**Name.** This is the name for the generated scalars. You can enter a literal string here. You can also use dollar variables bound in the *Value* and *Grouping* fields (e.g.  `${x}` ), and their expressions (e.g.  `${x+y+1}` ).

**Module.** This is the place where you can enter the module name for the newly computed scalars. If the value expression contains unqualified scalars (those without module name patterns) that are not subject to aggregate functions, and you agree to place the new scalars into the same modules as theirs, then this field can be left empty. Otherwise, enter the module name. You can use dollar variables bound in the *Value* and *Grouping* fields (e.g.  `${x}` ), and their expressions (e.g.  `${x+y+1}` ). Note that the module name does not need to be an existing module name; you can "make up" new modules by entering arbitrary names here.

**Average replications checkbox.** Check to compute only the average value across repetitions.

The computation is performed in each run independently by default. If some run is a replication of the same measurement with different seeds, you may want to average the results. If the *Average replications* checkbox is selected, then only one scalar is added to the dataset for each measurement.

A new run generated for the scalar which represents the set of replications which it is computed from. The attributes of this run are those that have the same value in all replications.

**Other checkboxes.** In addition to mean, you can also add other statistical properties of the computed scalar to the dataset by selecting the corresponding checkboxes in the dialog. The names of these new scalars will be formed by adding the `:stddev`, `:confint`, `:min`, or `:max` suffix to the base name of the scalar.



A more formal description of the *Compute Scalars* feature's operation, together with the list of available functions and other details, can be found in the Appendix (see Appendix A.*Specification of the 'Compute Scalars' operation*).

**Examples.** Let us illustrate the usage of the computations with some examples.

1. Assume that you have several source modules in the network that generate CBR traffic, parameterized with packet length (in bytes) and send interval (seconds). Both parameters are saved as scalars by each module (`pkLen`, `sendInterval`), but you want to use the bit rate for further computations or charts. Adding a *Compute Scalar* node with the following content will create an additional `bitrate` scalar for each source module:

**Value:** `pkLen*8/sendInterval`

**Name:** `bitrate`

2. Assume that several sink modules record `rcvdByteCount` scalars, and the simulation duration is saved globally as the `duration` scalar of the toplevel module. We are interested in the throughput at each sink module. In the value expression we can re-

fer to the duration scalar by its qualified name, i.e. prefix it with the full name of its module. `rcvdByteCount` can be left unqualified, and then the *Module* field doesn't need to be filled out because it will be inferred from the `rcvdByteCount` statistic.

**Value:** `8*rcvdByteCount / Network.duration`  
**Name:** throughput

3. If you are interested in the total number of bytes received in the network, you can use the `sum()` function. In this example we store the result as a new scalar of the toplevel module, `Network`.

**Value:** `sum(rcvdByteCount)`  
**Name:** totalRcvdBytes  
**Module:** Network

4. If several modules record scalars named `rcvdByteCount` but you are only interested in the ones recorded from network hosts, you can qualify the scalar name with a pattern:

**Value:** `sum(**.host*.**.rcvdByteCount)`  
**Name:** totalHostRcvdBytes  
**Module:** Network

 An alternative solution would be to restrict the input of the Compute Scalars node by adding a Select child under it.

5. If several modules record vectors named `end-to-end delay` and you are interested in the average of the peak end-to-end delays experienced by each module, you can use the `max()` function on the vectors to get the peak, then `mean()` to obtain their averages. Note that the vector name needs to be quoted with apostrophes because it contains spaces.

**Value:** `mean(max('end-to-end delay'))`  
**Name:** avgPeakDelay  
**Module:** Network

6. Let's assume there are 3 clients (`cli0`, `cli1`, `cli2`) and 3 servers (`srv0`, `srv1`, `srv2`) in the network, and each client sends datagrams to the corresponding server. The packet loss per client-server pair can be computed from the number of sent and received packets. We use the `i` variable to match the corresponding clients and servers. (Without the `i` variable, i.e. by writing just `Net.cli*.pkSent - Net.srv*.pkRcvd`, the result would be Cartesian product.)

**Value:** `Net.cli${i=0..2}.pkSent - Net.srv${i}.pkRcvd`  
**Name:** pkLoss  
**Module:** Net.srv\${i}

7. A similar example is when you want to compute the total number of transport packets (the sum of the TCP and UDP packet counts) for each host. Since the input scalars are recorded by different modules, we need the host variable to match TCP and UDP modules under the same host.

**Value:**  `${host=**}.udp.pkCount + ${host}.tcp.pkCount`  
**Name:** transportPkCount  
**Module:** \${host}

8. This example is a slight modification of the previous example. Assume that the TCP module writes an output vector (named `pkSent`) containing the length of each packet sent, and the UDP module writes a histogram of the sent packet lengths. As in the

previous example, we want to compute the number of sent packets for each host; we use `count()` to extract the number of values from histograms and output vectors.

**Value:** `count(${host=**}.udp.pkSent) + count(${host}.tcp.pkSent)`  
**Name:** `pkCount`  
**Module:**  `${host}`

9. Now we are computing the average number of data packets sent by the hosts. We will use the `mean()` function in the *Value* expression: `mean(${host=**}.udp.pkCount + ${host}.tcp.pkCount)`. The mean function computes one value from a set of values. Because this value can not be associated with one host, the `host` variable will be undefined outside the `mean()` function call. Therefore you can not enter  `${host}` into the *Module*, but an appropriate module name should be chosen. Important: the `mean()` function cannot be used to compute the average of values that come from different runs, as the *Value* expression is always evaluated with input statistics that come from the same run; you have to use the *Average replications* checkbox for that instead.

**Value:** `mean(${host=**}.udp.pkCount + ${host}.tcp.pkCount))`  
**Name:** `avgPkCount`  
**Module:** `Network`

10 Again, we are interested in the average number of sent packets, but we want to compute the average for each subnet. In SQL you would use the GROUP BY clause to generate that report, here you can use the *Grouping* expression. Let us assume that the subnets are at the second level of the module hierarchy, so they can be identified by the second component of the full names of modules. Giving `(module =~ *.${subnet=*}.**) ? subnet : "n/a"` as the *Grouping* expression, the group identifier will be the subnet of the modules (and n/a for the network). Now the same *Value* expression as in the previous example computes the average for each subnet. The group variable now contains the name of the subnet, so you can use the  `${group}` expression as the *Module*.

**Value:** `mean(${host=**}.udp.pkCount + ${host}.tcp.pkCount))`  
**Grouping:** `(module =~ *.${subnet=*}.**) ? subnet : "n/a"`  
**Name:** `avgPkCount`  
**Module:**  `${group}`

11 It is also possible to group the scalars by their values.

**Value:** `count(responseTime)`  
**Grouping:** `value > 1.0 ? "Large" : "Normal"`  
**Name:** `num${group}ResponseTimes`  
`or: ${"num" ++ group ++ "ResponseTimes"}`  
**Module:** `Network`

 Note the use of `++` for string concatenation in the name expression. The normal `+` operator does numeric addition, and it would attempt to convert both operands to numbers beforehand.

12 Assume that you run the simulation with the same parameter settings, but different seeds, i.e. you have runs that are replications of the same measurement. Then computations are repeated for each run, therefore they produce values for each replications. If you are not interested in the individual results in the replications, you can generate only their average by turning on the *Average replications* checkbox. The average will be saved with the name entered into the *Name* field. The minimum/maximum value, the standard deviation of the distribution, and the confidence interval of the mean can also be generated. Their name will contain a `:min`, `:max`, `:stddev`, `:confint` suffix.

13To add a constant value as a scalar, enter the value into *Value* field, its name into the *Name*, and the module name into the *Module* field. Note that you can use any name for the target module, not just names of existing modules. As a result of the computation, one scalar will be added in every run that is present in the input dataset.

**Value:** 299792458  
**Name:** speedOfLight  
**Module:** Network.channelControl

14If you want to add the constant for each module in your dataset, then enter *module* into the *Grouping* field, and  `${group}` into the *Module* field. In this case the statistics of the input dataset are grouped not only by their run, but by their module name too. The expression entered into the *Grouping* field is a predefined variable, that refers to the module name of the statistic which the group identifier is generated for. We refer to the value of the group identifier (i.e. the module name) in the *Module* field.

**Value:** 3.1415927  
**Grouping:** module  
**Name:** pi  
**Module:** \${group}

15Sometimes multiple steps are needed to compute what you need. Assume you have a network where various modules record ping round-trip delays (RTT), and you want to count the modules with large RTT values (where the average RTT is more than twice the global average in the network). The following examples achieves that. Step 1 and 2 could be merged, but we left them separate for better readability.

Step 1:

**Value:** mean('rtt:vector')  
**Name:** average

Step 2:

**Value:** average / mean(\*\*.average)  
**Name:** relativeAverage

Step 3:

**Value:** count(relativeAverage)  
**Grouping:** value > 2.0 ? "Above" : "Normal"  
**Name:** num\${group}  
**Module:** Net

16In this example, we have 100 routers (`Net.rte[0]..Net.rte[99]`) that all record the number of dropped packets (`drops` scalar), and we want to know how many routers dropped 0..99 packets, 100..199 packets, 200..299 packets, and so on. For this we group the scalars by their values, and count the scalars in each group.

**Value:** count(drops)  
**Grouping:** floor(value / 100)  
**Name:** values in the \${group}00- group  
**Module:** Net

17Assume we have the same 100 routers with drop count scalars as in the previous example. We want to group the routers in batches of 10 by index, and compute the average drop count in each batch.

---

**Value:** mean(drops)  
**Grouping:** (module=~ \*.rte[\$10index=\*]) ? floor(index/10) : "n/a"

**Name:** avgDropsInGroup\${group}  
**Module:** Net

This is a variation of the previous example: if the batches are not of equal size, we can use the `locate(x,a1,a2,a3,...an)` function to group them. `locate()` returns the index of the first element of  $a1..an$  that is less or equal than  $x$  (or 0 if  $x < a1$ ).

**Value:** mean(drops)  
**Grouping:**  
`(module=~ *.rte[$index=*)] ? locate(index,5,15,35,85) : "n/a"`  
**Name:** avgDropsInGroup\${group}  
**Module:** Net

## Export

You can export the content of the dataset into text files. Three formats are supported: comma separated values (CSV), Octave text files and Matlab script. Right-click on the processing node or chart, and select the format from the *Export to File* submenu. The file will contain the data of the chart, or the dataset after the selected processing is applied. Enter the name of the file and press *Finish*.

Vectors are always written as two columns into the CSV files, but the shape of the scalars table can be changed by selecting the grouping attributes in the dialog. For example, assume we have two scalars (named s1 and s2) written by two modules (m1 and m2) in two runs (r1 and r2), resulting in a total of 8 scalar values. If none of the checkboxes is selected in the *Scalars grouped by* group, then the data is written as:

r1 m1 s1	r1 m1 s2	r1 m2 s1	r1 m2 s2	r2 m1 s1	r2 m1 s2	r2 m2 s1	r2 m2 s2
1	2	3	4	5	6	7	8

Grouping the scalars by module name and scalar name would have the following result:

Module	Name	r1	r2
m1	s1	1	5
m1	s2	2	6
m2	s1	3	7
m2	s2	4	8

The settings specific to the file format are:

**CSV.** You can select the separator, line ends and quoting character. The default setting corresponds to RFC4180. The precision of the numeric values can also be set. The CSV file contains an optional header followed by the vector's data or groups of scalars. If multiple vectors are exported, each vector is written into a separate file.

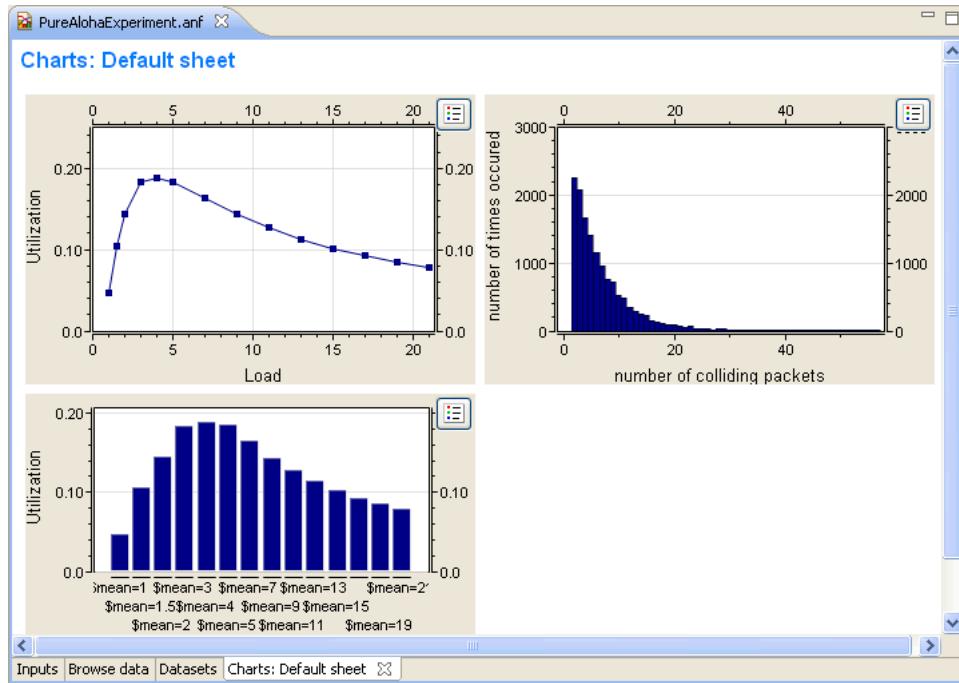
**Octave.** The data is exported as an Octave text file. This format can be loaded into the R [<http://www.r-project.org>] statistical data analysis tool, as well. The tables are saved as structures containing an array for each column.

**Matlab.** The data is exported as a Matlab script file. It can be loaded into Matlab/Octave with the `source()` function.

## Chart sheets

Sometimes, it is useful to display several charts on one page. When you create a chart, it is automatically added to a default chart sheet. Chart sheets and their charts are

displayed on the lower pane of the *Datasets* page. To create a new chart sheet, use the *Chart Sheet* button on the palette. You can add charts to it either by using the opening dialog or by dragging charts. To move charts between chart sheets, use drag and drop or Cut/Paste. You can display the charts by double-clicking on the chart sheet node.



**Figure 10.6. Chart Sheet page with three charts**

### 10.3.3. Charts

#### Overview

You typically want to display the recorded data in charts. In OMNeT++ 4.x, you can open charts for scalar, vector or histogram data with one click. Charts can be saved into the analysis file, too. The Analysis Editor supports bar charts, line charts, histogram charts and scatter charts. Charts are interactive; users can zoom, scroll, and access tooltips that give information about the data items.

Charts can be customized. Some of the customizable options include titles, fonts, legends, grid lines, colors, line styles, and symbols.

#### Creating charts

To create a chart, use the palette on the *Dataset* page. Drag the chart button and drop it to the dataset at the position you want it to appear. If you press the chart button then it opens a dialog where you can edit the properties of the new chart. In this case the new chart will be added at the end of the selected dataset or after the selected dataset item.

Temporary charts can be created on the *Browse Data* page for quick view. Select the scalars, vectors or histograms and choose *Plot* from the context menu. If you want to save such a temporary chart in the analysis, then choose *Convert to dataset...* from the context menu of the chart or from the toolbar.

#### Editing charts

You can open a dialog for editing charts from the context menu. The dialog is divided into several pages. The pages can be opened directly from the context menu. When

you select a line and choose *Lines...* from the menu, you can edit the properties of the selected line.

You can also use the *Properties View* to edit the chart. It is recommended that users display the properties grouped according to their category ( on the toolbar of the *Properties View*).

Main	
antialias	Enables antialiasing.
caching	Enables caching. Caching makes scrolling faster, but sometimes the plot might not be correct.
background color	Background color of the chart.
Titles	
graph title	Main title of the chart.
graph title font	Font used to draw the title.
x axis title	Title of the horizontal axis.
y axis title	Title of the vertical axis.
axis title font	Font used to draw the axes titles.
labels font	Font used to draw the tick labels.
x labels rotated by	Rotates the tick labels of the horizontal axis by the given angle (in degrees).
Axes	
y axis min	Crops the input below this y value.
y axis max	Crops the input above this y value.
y axis logarithmic	Applies a logarithmic transformation to the y values.
grid	Add grid lines to the plot.
Legend	
display	Displays the legend.
border	Add border around the legend.
font	Font used to draw the legend items.
position	Position of the legend.
anchor point	Anchor point of the legend.

**Table 10.2. Common chart properties**

### Zooming and panning

Charts have two mouse modes. In Pan mode, you can scroll with the mouse wheel and drag the chart. In Zoom mode, the user can zoom in on the chart by left-clicking and zoom out by doing a **Shift**+left click, or using the mouse wheel. Dragging selects a rectangular area for zooming. The toolbar icons  and  switch between Pan and Zoom modes. You can also find toolbar buttons to zoom in , zoom out  and zoom to fit .

### Tooltip

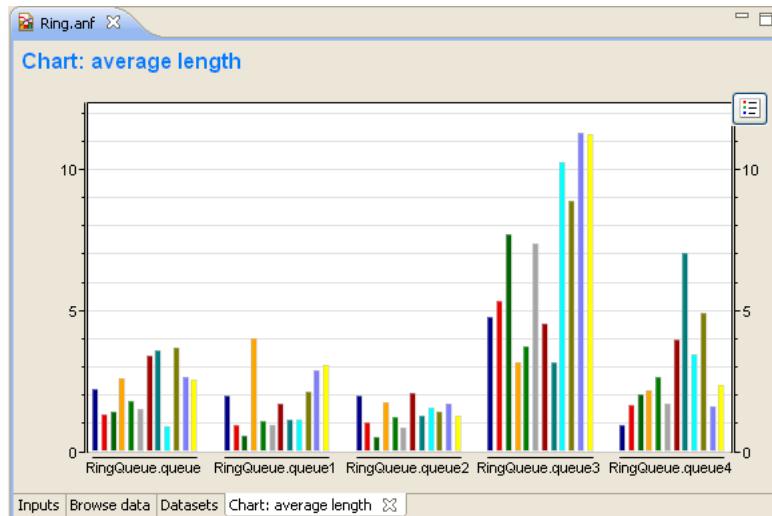
When the user hovers the mouse over a data point, the appearing tooltip shows line labels and the values of the points close to the cursor. The names of all lines can be displayed by hovering over the  button at the top right corner of the chart.

### Copy to clipboard

You can copy the chart to the clipboard by selecting *Copy to Clipboard* from the context menu. The chart is copied as a bitmap image and is the same size as the chart on the screen.

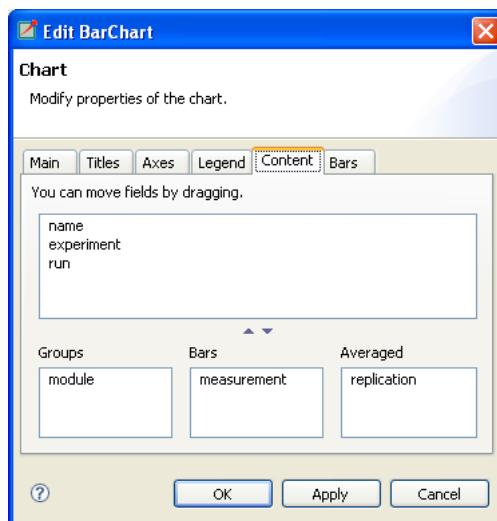
### Bar charts

Bar charts display scalars as groups of vertical bars. The bars can be positioned within a group next to, above or in front of each other. The baseline of the bars can be changed. Optionally, a logarithmic transformation can be applied to the values.



**Figure 10.7. Bar chart**

The scalar chart's content can be specified on the *Content* tab of their *Properties* dialog. Attributes in the "Groups" list box determine the groups so that within a group each attribute has the same value. Attributes in the "Bars" list box determine the bars; the bar height is the average of scalars that have the same values as the "Bar" attributes. You can classify the attributes by dragging them from the upper list boxes to the lower list boxes. You will normally want to group the scalars by modules and label the bars with the scalar name. This is the default setting, if you leave each attribute in the upper list box.



**Figure 10.8. Dialog page for bar chart content**

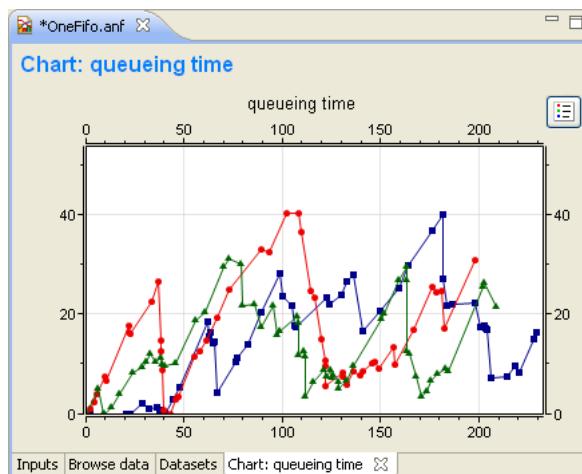
In addition to the common chart properties, the properties of bar charts include:

Titles	
wrap labels	If true labels are wrapped, otherwise aligned vertically.
Plot	
bar baseline	Baseline of the bars.
bar placement	Arrangement of the bars within a group.
Bars	
color	Color of the bar. Color name or #RRGGBB. Press Ctrl+Space for a list of color names.

**Table 10.3. Bar chart properties**

### Line charts

Line charts can be used to display output vectors. Each vector in the dataset gives a line on the chart. You can specify the symbols drawn at the data points (cross, diamond, dot, plus, square triangle or none), how the points are connected (linear, step-wise, pins or none) and the color of the lines. Individual lines can be hidden.



**Figure 10.9. Line chart**

Line names identify lines on the legend, property sheets and edit dialogs. They are formed automatically from attributes of the vector (like file, run, module, vector name, etc.). If you want to name the lines yourself, you can enter a name pattern in the *Line names* field of the *Properties* dialog (*Main* tab). You can use "{file}", "{run}", "{module}", or "{name}" to refer to an attribute value. Press **Ctrl+Space** for the complete list.

Processing operations can be applied to the dataset of the chart by selecting *Apply* or *Compute* from the context menu. If you want to remove an existing operation, you can do it from the context menu, too.

Line charts are synchronized with *Output Vector* and *Dataset* views. Select a data point and you will see that the data point and the vector are selected in the Output Vector and *Dataset View*, as well.

Axes	
x axis min	Crops the input below this x value.

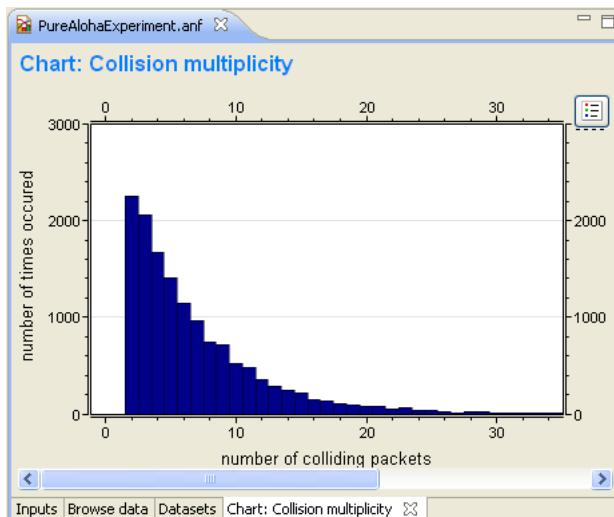
x axis max	Crops the input above this x value.
<b>Lines</b>	
display name	Display name of the line.
display line	Displays the line.
symbol type	The symbol drawn at the data points.
symbol size	The size of the symbol drawn at the data points.
line type	Line drawing method. One of Linear, Pins, Dots, Points, Sample-Hold or Backward Sample-Hold.
line color	Color of the line. Color name or #RRGGBB. Press Ctrl+Space for a list of color names.

**Table 10.4. Line chart properties**

### Histogram charts

Histogram charts can display data of histograms. They support three view modes:

Count	The chart shows the recorded counts of data points in each cell.
Probability density	The chart shows the probability density function computed from the histogram data.
Cumulative density	The chart shows the cumulative density function computed from the histogram data.



**Figure 10.10. Histogram chart**



When drawing several histograms on one chart, set the "Bar type" property to Outline. This way the histograms will not cover each other.

Plot	
bar type	Histogram drawing method.
bar baseline	Baseline of the bars.
histogram data type	Histogram data. Counts, probability density and cumulative density can be displayed.

show overflow cell	Show over/underflow cells.
Histograms	
hist color	Color of the bar. Color name or #RRGGBB. Press Ctrl+Space for a list of color names.

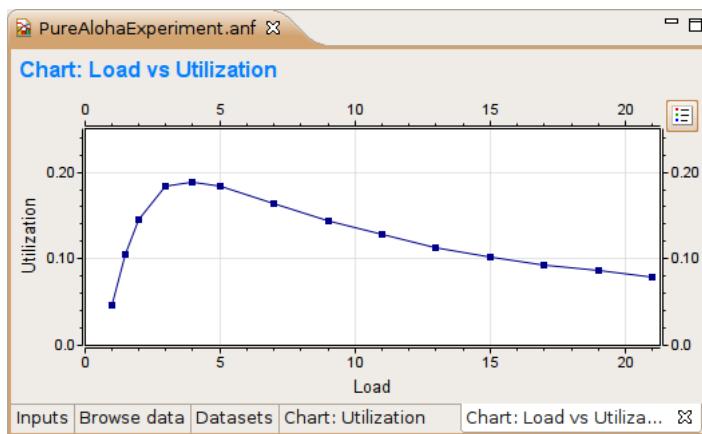
**Table 10.5. Histogram chart properties**

### Scatter charts

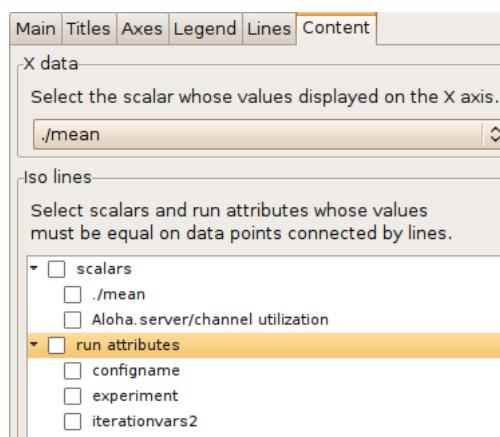
Scatter charts can be created from both scalar and vector data. You have to select one statistic for the x coordinates; other data items give the y coordinates. How the x and y values are paired differs for scalars and vectors.

**Scalars.** For each value of the x scalar, the y values are selected from scalars in the same run.

**Vectors.** For each value of the x vector, the y values are selected from the same run and with the same simulation time.


**Figure 10.11. A scatter chart**

By default, each data point that comes from the same y scalar belongs to the same line. This is not always what you want because these values may have been generated in runs having different parameter settings; therefore, they are not homogenous. You can specify scalars to determine the "iso" lines of the scatter chart. Only those points that have the same values of these "iso" attributes are connected by lines.


**Figure 10.12. A scatter chart**



If you want to use a module parameter as an iso attribute, you can record it as a scalar by setting "<module>.<parameter\_name>.param-record-as-scalar=true" in the INI file.

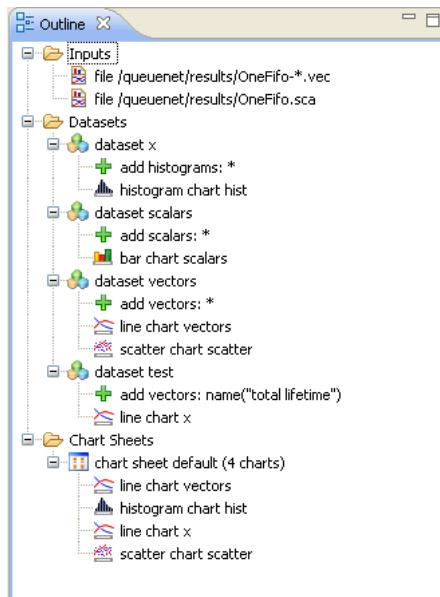
Axes	
x axis min	Crops the input below this x value.
x axis max	Crops the input above this x value.
Lines	
display name	Display name of the line.
display line	Displays the line.
symbol type	The symbol drawn at the data points.
symbol size	The size of the symbol drawn at the data points.
line type	Line drawing method. One of Linear, Pins, Dots, Points, Sample-Hold or Backward Sample-Hold.
line color	Color of the line. Color name or #RRGGBB. Press Ctrl+Space for a list of color names.

**Table 10.6. Scatter chart properties**

## 10.4. Associated Views

### 10.4.1. Outline View

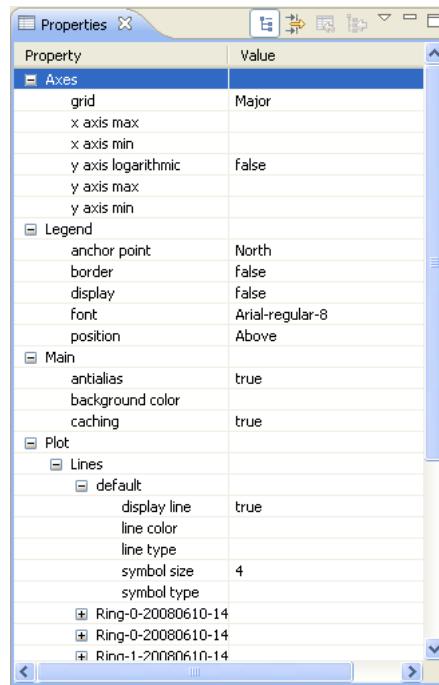
The *Outline View* shows an overview of the current analysis. Clicking on an element will select the corresponding element in the current editor. Tree editing operations also work in this view.



**Figure 10.13. Outline View of the analysis**

### 10.4.2. Properties View

The Properties View displays the properties of the selected dataset, processing node and chart. Font and color properties can be edited as text or by opening dialogs. Text fields that have a bulb on the left side have a content assist; press **Ctrl+Space** to activate it.



**Figure 10.14. Properties View showing chart properties**

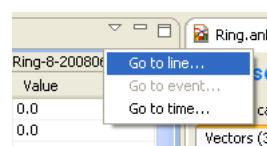
#### 10.4.3. Output Vector View

The *Output Vector View* shows the content of the selected vector. It displays the serial number, simulation time and value of the data points. When event numbers are recorded during the simulation, they are also displayed. Large output files are handled efficiently; only the visible part of the vector is read from the disk. Vectors that are the result of a computation are saved in temporary files.

Item#	Time	Value
0	9.333606143617654	0.0
1	16.09325237278197	0.0
2	20	0.0
3	24.51532566751914	0.0
4	25.98997613616606	0.29797143009415
5	28.94849378051141	0.0
6	30.16836441439083	0.0
7	30.84666725716397	0.0
8	33.16243090199714	0.0
9	38.54651808887529	0.0
10	41.00754548454184	0.0
11	44.40733447963808	1.7936577895403
12	46.27199732455145	2.9988566001251
13	48.39774482763042	3.4836809392653
14	50.983682772064	5.8503127999962
15	51.81434791674422	5.3484650457847
16	54.09515619215544	7.5985372321639
17	58.44160017779978	10.53616200814
18	59.45113901802828	0.4556614090837

**Figure 10.15. Output Vector View**

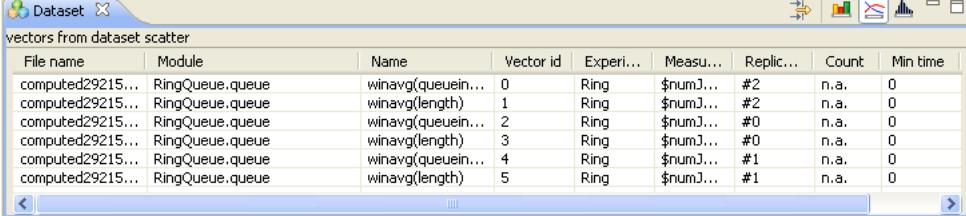
To navigate to a specific line, use the scroll bar or the menu of the view:



**Figure 10.16.**

#### 10.4.4. Dataset View

The *Dataset View* displays the dataset's content of the selected dataset item. It is divided into three tables, similar to the ones on the *Browse Data* page. The tables can be selected by the icons. There is also a tool button to show or hide the filter.



File name	Module	Name	Vector id	Experi...	Measu...	Replic...	Count	Min time
computed29215...	RingQueue.queue	winavg(queuein...	0	Ring	\$numJ... #2	n.a.	0	
computed29215...	RingQueue.queue	winavg(length)	1	Ring	\$numJ... #2	n.a.	0	
computed29215...	RingQueue.queue	winavg(queuein...	2	Ring	\$numJ... #0	n.a.	0	
computed29215...	RingQueue.queue	winavg(length)	3	Ring	\$numJ... #0	n.a.	0	
computed29215...	RingQueue.queue	winavg(queuein...	4	Ring	\$numJ... #1	n.a.	0	
computed29215...	RingQueue.queue	winavg(length)	5	Ring	\$numJ... #1	n.a.	0	

**Figure 10.17. Dataset View**

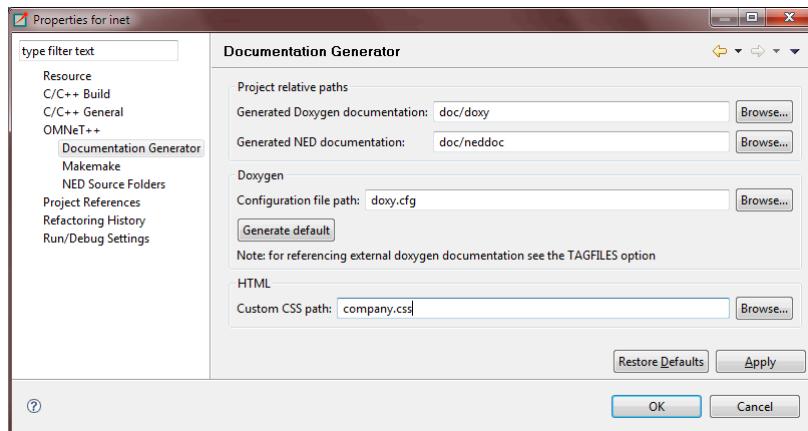
# Chapter 11. NED Documentation Generator

## 11.1. Overview

This chapter describes how to use the NED Documentation Generator from the IDE.

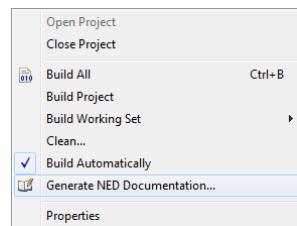
Please refer to the OMNeT++ Manual for a complete description of the documentation generation features and the available syntax in NED and MSG file comments.

The generator has several project-specific settings that can be set from the project context menu through the Properties menu item. The output folders for both NED documentation and C++ documentation can be set separately. The doxygen-specific configuration is read from the text file doxy.cfg by default. The IDE provides a sensible default configuration for doxygen in case you do not want to go through all the available options. The generated HTML uses CSS to make its style customizable. You can provide your own style sheet if the default does not meet your needs. In general, all project-specific settings have good defaults that work well with the documentation generator.



**Figure 11.1. Configuring project-specific settings**

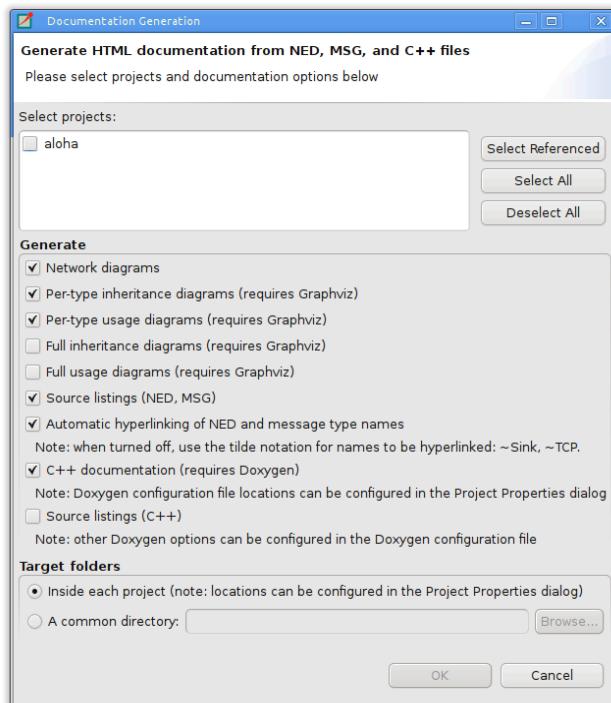
To generate NED documentation, you need to select one or more projects. Then, either go to the main Project menu or to the project context menu and select the Generate NED Documentation menu item. This will bring up the configuration dialog where you can set various settings for the current generation before starting it.



**Figure 11.2. Opening the NED documentation generator**

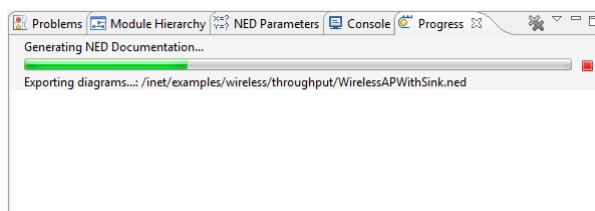
The IDE can generate documentation for multiple projects at the same time. Other options control the content of the documentation including what kind of diagrams will be generated and whether NED sources should be included. You can enable doxygen to

generate C++ documentation that will be cross-linked from the NED documentation. The tool can generate the output into each project as configured in the project-specific settings, or into a separate directory. The latter is useful for exporting standalone documentation for several complex projects at once.



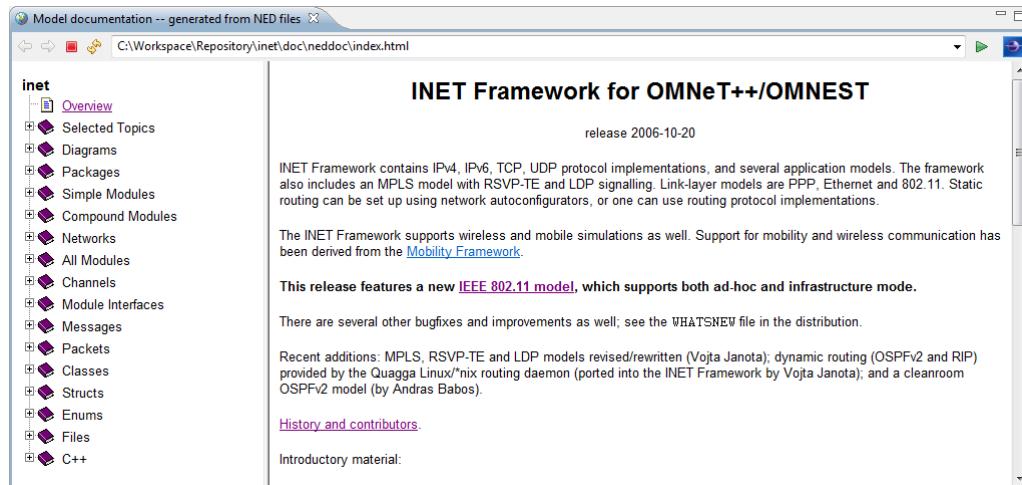
**Figure 11.3. Configuring the NED documentation generator**

The NED generation process might take a while for big projects, so please be patient. For example, the INET project's complete documentation including the C++ doxygen documentation takes a few minutes to build. You can follow the process in the IDE's progress monitor.



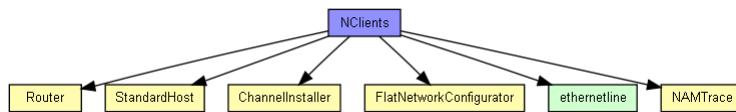
**Figure 11.4. Generating NED documentation in progress**

The result is a number of cross-linked HTML pages that can be opened by double-clicking the generated `index.html`. On the left side, you will see a navigation tree, while on the right side there will be an overview of the project. If you have not yet added a `@titlepage` directive into your NED comments, then the overview page will display a default content.



**Figure 11.5.** The resulting NED documentation

The documentation contains various inheritance and usage diagrams that make it easier to understand complex models. The diagrams are also cross-linked, so that when you click on a box, the corresponding model element's documentation will be opened. The NED model elements are also exported graphically from the NED Editor. These static images provide cross-referencing navigation for submodules.



**Figure 11.6.** NED usage diagram

There are also a number of tables that summarize various aspects of modules, networks, messages, packets, etc. The most interesting is the list of assignable parameters. It shows all parameters from all submodules down the hierarchy that do not have fixed values. These can be set either through inheritance, encapsulation or from the INI file as experiments.

Assignable submodule or channel parameters:			
Name	Type	Default value	Description
<code>configurator.networkAddress</code>	string	"192.168.0.0"	network part of the address (see netmask parameter)
<code>configurator.netmask</code>	string	"255.255.0.0"	host part of addresses are autoconfigured
<code>nam.logfile</code>	string	"trace.nam"	
<code>nam.prolog</code>	string	""	
<code>r1.routingFile</code>	string	""	
<code>r1.network_layer.proxyARP</code>	bool	true	
<code>r1.network_layer_ip.procDelay</code>	double	0s	
<code>r1.network_layer_arp.retryTimeout</code>	double	1s	number seconds ARP waits between retries to resolve an IP address
<code>r1.network_layer_arp.retryCount</code>	int	3	number of times ARP will attempt to resolve an IP address
<code>r1.network_layer_arp.cacheTimeout</code>	double	120s	number seconds unused entries in the cache will time out
<code>r1.ppp[*].queueType</code>	string	"DropTailQueue"	
<code>r1.ppp[*].ppp.mtu</code>	int	4470	
<code>r1.eth[*].queueType</code>	string	"DropTailQueue"	
<code>r1.eth[*].mac.promiscuous</code>	bool	false	if true, all packets are received, otherwise only the ones with matching destination MAC address
<code>r1.eth[*].mac.address</code>	string	"auto"	MAC address as hex string (12 hex digits), or "auto". "auto" values will be replaced by a generated MAC address in init_stage_0

**Figure 11.7.** NED assignable parameters

There are other tables that list parameters, properties, gates, using modules or networks, and various other data along with the corresponding descriptions. In general, all text might contain cross-links to other modules, messages, classes, etc. to make navigation easier.

# Chapter 12. Extending the IDE

There are several ways to extend the functionality of the OMNeT++ IDE. The Simulation IDE is based on the Eclipse platform, but extends it with new editors, views, wizards, and other functionality.

## 12.1. Installing New Features

Because the IDE is based on the Eclipse platform, it is possible to add additional features that are available for Eclipse. The installation procedure is exactly the same as with a standard Eclipse distribution. Choose the *Help | Install New Software...* menu item and select an existing Update Site to work with or add a new Site (using the site URL) to the Available Software Sites. After the selection, you can browse and install the packages the site offers.

To read about installing new software into your IDE, please visit the *Updating and installing software* topic in the *Workbench User Guide*. You can find the online help system in the *Help | Help Contents* menu.



There are thousands of useful components and extensions for Eclipse. The best places to start looking for extensions are the Eclipse Marketplace (<http://marketplace.eclipse.org/>) and the Eclipse Plugins info site (<http://www.eclipse-plugins.info>).

## 12.2. Adding New Wizards

The Simulation IDE makes it possible to contribute new wizards into the wizard dialogs under the *File | New* menu without writing Java code or requiring any knowledge of Eclipse internals. Wizards can create new simulation projects, new simulations, new NED files or other files by using templates, or perform export/import functions. Wizard code is placed under the *templates* folder of an OMNeT++ project, which makes it easy to distribute wizards with the model. When the user imports and opens a project which contains wizards, the wizards will automatically become available.



The way to create wizards is documented in the *IDE Customization Guide*.

## 12.3. Project-Specific Extensions

It is possible to install an Eclipse plug-in by creating a *plugins* folder in an OMNeT++ project and copying the plug-in JAR file to that folder (this mechanism is implemented as part of the Simulation IDE and does not work in generic Eclipse installations or with non-OMNeT++ projects). This extension mechanism allows the distribution of model-specific IDE extensions together with a simulation project without requiring the end user to do extra deployment steps to install the plug-in. Plugins and wizards that are distributed with a project are automatically activated when the host project is opened.

Eclipse plug-in JAR files can be created using the *Plug-in Development Environment*. The OMNeT++ IDE does not contain the *PDE* by default; however, it can be easily installed, if necessary.



Read the *OMNeT++ IDE Development Guide* for more information about how to install the PDE and how to develop plug-in extensions for the IDE.

---

# Appendix A. Specification of the 'Compute Scalars' operation

This appendix describes the *Compute Scalars* operation in details. Some fields of the operation can contain expressions, so we first describe the expressions. The *Compute Scalars* operation is described in the following section.

## A.1. Expressions

An expression is composed of constants, scalar values, vector or histogram fields, variables, operators, and functions.

### Constants

Numeric constants are accepted in the standard format.

Examples: -1, 3.14159, 4.2e1

String constants must be enclosed with double quotes. Double quotes within the string must be preceded by a backslash character.

Examples: "xyz", "This string contains a \" character"

### Scalars

The values of scalars in the input dataset can be accessed in two ways:

- **Simple names.** Using the name of a scalar in the expression refers to all scalar with that name. If the name of the scalar contains non-alphanumeric characters, it must be enclosed with apostrophies.
- **Qualified names.** Qualified names are in the form <module>. <scalar>. Here <module> is a pattern (see ...) that matched to the full name of the module of the scalar; <scalar> is the name of the scalar (quoted if needed). As a special case, <module> can also be the full name of the module.

Note that the scalar references may produce multiple values. Consequently the expression that contains them may also produce multiple values by iterating on the values of the scalars. The iteration is defined differently for the two types of scalars references:

- Scalars referenced by their simple names are restricted to come from the same module. For example, if we have four scalars recorded by two modules, m1.a, m1.b, m2.a, m2.b, then a+b produces two values: m1.a+m1.b and m2.a+m2.b.
- Scalars referenced by their qualified names are iterated independently on their modules. This means that if there are several such pattern in the expression, than the computation is performed on their cartesian product. With the input of the previous example, \*.a+\*.b produces four values: m1.a+m1.b, m1.a+m2.b, m2.a+m1.b, and m2.a+m2.b.

The iteration can be restricted by binding some part of the module name to variables, and use those variables in other patterns. The \${x=<pattern>} syntax in a module name pattern binds the part of the module name matched to pattern to a variable named x. These variables can be referred as \${x} in other patterns. The \${...} syntax allows to write any expression in the pattern (like \${x+1}).

Examples: iaTime, 'sentPk:count', Aloha.server.duration, cli \${i={0..2}}.pkSent, \*\*.host[\$i+2].'end-to-end-delay'



Simple scalars references can be viewed as a shortcut for qualified references, where the module part is  `${m=**}` in the first reference, and  `${m}` in the subsequent references (here `m` is a fresh variable name). E.g. if `a`, and `b` are scalars, then `a*b` is equivalent to  `${m=**}.a * ${m}.b`.

## Vectors and Histograms

To refer to a field of a vector or histogram, use `count(<name>)`, `mean(<name>)`, `min(<name>)`, `max(<name>)`, `stddev(<name>)`, and `variance(<name>)`. Here `<name>` is the simple or qualified name of the vector or histogram.

The same rules apply to qualified names and iterations over modules, as in the case of scalars.

## Patterns

Patterns can be used to specify the module name of an input statistic, or as the right-hand-side of the pattern matching operator (`=~`). Characters of the patterns are matched literally, except the following:

Pattern	Description
<code>?</code>	matches any character
<code>*</code>	matches zero or more characters except <code>'</code>
<code>**</code>	matches zero or more characters (any character)
<code>{a-z}</code>	matches a character in range <code>a-z</code>
<code>{^a-z}</code>	matches a character not in range <code>a-z</code>
<code>{32..255}</code>	any number (i.e. sequence of digits) in range <code>32..255</code> (e.g. "99")
<code>[32..255]</code>	any number in square brackets in range <code>32..255</code> (e.g. "[99]")
<code>\${x=**}</code>	matches a pattern, and binds the matched substring to <code>x</code>
<code>\${x+1}</code>	evaluates an expression, and matches the characters of the result
<code>\</code>	takes away the special meaning of the subsequent character

## Variables

Variables can be defined by using the  `${<var>}=...` notation in patterns. The scope of the variable is to the right of its definition. There can also be some predefined variables, and variables defined in one expression can be accessible in another.

### Predefined variables

in <i>Grouping expression</i>	module, name, run, run attributes of the current statistics, value if the current statistics is a scalar
in <i>Value expression</i>	group is the value of the grouping expression
in <i>Target module</i>	group is the value of the grouping expression

## Operators

The following operators are interpreted as usual:

Arithmetic	<code>+ - * / ^ %</code>
Bitwise	<code>~   &amp; # &lt;&lt; &gt;&gt;</code>
Concatenation	<code>++</code>
Comparision	<code>== != &lt; &gt; &lt;= &gt;=</code>

Boolean	<code>!     &amp;&amp;</code>
Conditional	<code>? :</code>
Pattern matching	<code>=~</code>

Arithmetic, bitwise, concatenation, boolean, and comparision operators always evaluate their arguments; binary operators evaluate their left arguments first. Arithmetic, bitwise, and comparision operators implicitly convert their arguments to numeric values, the concatenation operator converts them to strings, the boolean operators converts them to boolean values. A runtime error occurs if the conversion fails.

The conditional operator (`cond ? a : b`) is special, because it does not evaluate all operands. If the condition is true, then the second, otherwise the third operand is evaluated.

The pattern matching operator `=~` expects a string expression as the left, and a pattern as the right operand. If the pattern matches with the string, then the result is the string, otherwise false.

## Functions

The following functions compute an aggregated value from a set of values:

Function	Description
<code>count(&lt;expr&gt;)</code>	count of the values produced by <code>&lt;expr&gt;</code>
<code>sum(&lt;expr&gt;)</code>	sum of the values produced by <code>&lt;expr&gt;</code>
<code>min(&lt;expr&gt;)</code>	minimum of the values produced by <code>&lt;expr&gt;</code>
<code>max(&lt;expr&gt;)</code>	maximum of the values produced by <code>&lt;expr&gt;</code>
<code>mean(&lt;expr&gt;)</code>	mean of the values produced by <code>&lt;expr&gt;</code>
<code>stddev(&lt;expr&gt;)</code>	standard deviation of the values produced by <code>&lt;expr&gt;</code>
<code>variance(&lt;expr&gt;)</code>	variance of the values produced by <code>&lt;expr&gt;</code>

The following functions are not aggregating functions; if their arguments has multiple values, then the applying the function also produces multiple values by iterating on them.

Function	Description
<i>Math functions</i>	
<code>sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(x,y), exp(x), log(x), log10(x), sqrt(x), cbrt(x), hypot(x,y), sinh(x), cosh(x), tanh(x), ceil(x), floor(x), round(x), signum(x), min(x,y), max(x,y)</code>	These functions produce the same value as the similarly named methods of the <code>java.lang.Math</code> class in Java.
<code>deg(x), rad(x)</code>	Conversion from radians to degrees, and from degrees to radians
<code>fabs(x)</code>	Absolute value of <code>x</code>
<code>rem(x,y)</code>	IEEE floating point remainder of <code>x</code> and <code>y</code>
<i>String functions</i>	
<code>length(s)</code>	Returns the length of the string.
<code>contains(s, substr)</code>	Returns true if string <code>s</code> contains <code>substr</code> as substring.

Function	Description
substring(s, pos, len?)	Return the substring of s starting at the given position, either to the end of the string or maximum len characters.
substringBefore(s, substr)	Returns the substring of s before the first occurrence of substr, or the empty string if s does not contain substr.
substringAfter(s, substr)	Returns the substring of s after the first occurrence of substr, or the empty string if s does not contain substr.
substringBeforeLast(s, substr)	Returns the substring of s before the last occurrence of substr, or the empty string if s does not contain substr.
substringAfterLast(s, substr)	Returns the substring of s after the last occurrence of substr, or the empty string if s does not contain substr.
startsWith(s, substr)	Returns true if s begins with the substring substr.
endsWith(s, substr)	Returns true if s ends with the substring substr.
tail(s, len)	Returns the last len character of s, or the full s if it is shorter than len characters.
replace(s, substr, repl, startPos?)	Replaces all occurrences of substr in s with the string repl. If startPos is given, search begins from position startPos in s.
replaceFirst(s, substr)	Replaces the first occurrence of substr in s with the string repl. If startPos is given, search begins from position startPos in s.
trim(s)	Discards whitespace from the start and end of s, and returns the result.
indexOf(s, substr)	Returns the position of the first occurrence of substring substr in s, or -1 if s does not contain substr.
choose(index, s)	Interprets s as a space-separated list, and returns the item at the given index. Negative and out-of-bounds indices cause an error.
toUpper(s)	Converts s to all uppercase, and returns the result.
toLower(s)	Converts s to all lowercase, and returns the result.
<i>Misc functions</i>	
select(index, ...)	Returns the indexth item from the rest of the argument list; numbering starts from 0.
locate(x, ...)	Returns the zero-based index of the first argument that is greater than or equal to x. If no such element, then it returns the number of elements (index of

Function	Description
	last element + 1). Example: <code>locate(42, 0, 10, 20, 50, 100) == 3</code>

## Implicit conversions

The value of an expression can be a boolean, a double, an integer, or a string. During the evaluation, values are converted to the expected types of functions and operators automatically. The rules of these conversions are:

- when a double is expected, then string values are parsed; boolean values are converted to 1 (`true`), or 0 (`false`)
- when an integer is expected, then values are converted to double and are rounded
- when a boolean is expected, then 0 converted to `false`, anything else to `true`. String values are first converted to numeric, then to boolean.
- when a string is expected, then numbers are converted to their decimal notation, booleans are converted to "0" or "1".

## Parsing ambiguities

Expressions like  $a.b*c.d$  can be parsed as a reference to the  $d$  statistic of modules whose names matches the  $a.b*c$  pattern, or as the product of  $a.b$  and  $c.d$ . The expression parser always prefers the first meaning, i.e. \* and ? characters are interpreted as part of the pattern. If you want to enter the product or conditional expression, you can add spaces around the operators. Patterns can not contain unquoted spaces, so the parse will be unambiguous.

There is another ambiguity that arises from the use of simple names. If you use e.g. `s` in an expression, it can refer a scalar or a variable. In this case the name is first tried to be resolved as a variable reference, and if it was unsuccessful, then as a statistic name. However quoted names (e.g. '`s`') always refer to statistics.

## A.2. Computing Scalars

The *Compute Scalars* operation is specified by the following attributes:

Attribute	Description
group	a string expression that generates a group identifier for an input statistic
value	a numeric expression that produces the values of the new scalars
name	the name of the new scalars, may contain expressions within <code>\${...}</code>
module	the module of the new scalars, may contain expressions within <code>\${...}</code>
averageReplications	a boolean value, if true, then the computed scalars are averaged for each measurement, and add only the averages to the dataset
computeStddev	a boolean value, if true, then the standard deviation of the computed scalars are also added to the dataset
computeMinMax	a boolean value, if true, then the minimum/maximum of computed scalars are also added to the dataset
computeConfInt	a boolean value, if true then the half length of the symmetric confidence interval is also added to the dataset

Attribute	Description
confIntLevel	a number between 0 and 1, the confidence level of the computed confidence interval

The generation of the scalars follows the following steps:

1. Group input statistics by their run. The computation of scalars is performed in each run independently. You can not combine the statistics of different runs, other than computing the averages described in step 4.
2. Within each run, group the input statistics further by their group identifier. The group identifier of a statistic is computed by evaluating the *group* expression. The following variables can be used in the expression to access the fields of the current statistic:
  - name: the name of the statistic
  - module: the module that generated the statistic
  - value: the value of the statistics if it is a scalar, otherwise NaN
  - run: the name of the run in which the statistics was generated
  - configname, datetime, experiment, measurement, and other attributes of the run

The group identifier will be available in Step 3 as the *group* variable. The *group* expression is optional, by default each scalar belongs to the same group.

3. Within each group, evaluate the *value* expression. The evaluation can produce multiple values. For each value generate a name and a module by evaluating the *name* and *module* expressions. The *value*, *name*, and *module* expressions can refer to the *group* variable, and to any variable that is defined in the *group* expression and is constant within a group. The *name* and *module* expressions can also refer to variables defined in the *value* expression.
4. If the *averageReplications* field is false, then the computation produces the scalars whose modules and values were computed in Step 3. Otherwise the values that belongs to the same module and to the same measurement, and has the same name are collected, and their mean is computed. If requested, then standard deviation, minimum, maximum, and confidence interval of the mean is also be computed. The name of these additional scalars will be scalar:stddev, scalar:min, scalar:max, scalar:confint, where scalar is the name of mean as specified in the *name* field of the computation.

A new run identifier also generated to refer to the set of runs from which the average was computed. This run identifier has the form runs-xxxxxx, where xxxx is a hash computed from the identifiers of the members. The new run will have the attributes that has the same value in the individual runs.