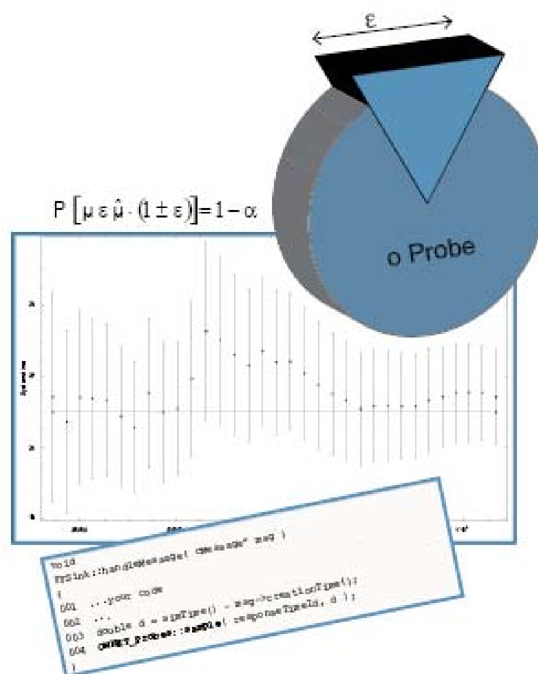
oProbe - an OMNeT++ Extension Module

by Tore J. Berg

Last updated: November 27, 2007



Contents

	Summary	**3**
1	**Introduction**	**3**
1.1	Terminology	5
1.2	Acknowledgement	5
2	**Getting Started**	**6**
2.1	Using *oProbe*	9
2.2	The effect of correlated samples	16
3	**The Software Architecture**	**21**
3.1	The OMNET-package	24
4	**The Probe Module**	**24**
5	**Interface Levels**	**27**
5.1	Level 1 (no GUI)	27
5.2	Level 2	27
5.3	Full functionality	27
6	**Tips and Tricks**	**28**
6.1	Checking input files	28
6.2	Checking the run-time length	28
6.3	Performance issues	29
7	**Installation**	**30**
7.1	Basic Build	30
7.2	Build with HippoDraw	32
7.3	KDevelop users	32
7.4	FAQ	32
7.4.1	Release/debug mode	32
7.4.2	Changing default paths	33
7.4.3	Installation on Mandriva	33
8	**References**	**35**

Summary

The simulation community has been accused for producing many simulation studies of bad quality [4], and the article stresses the importance of producing statistically sound results. The mission of the *oProbe* open source project is to provide an instrument that produces statistically sound results at known quality. Stochastic sampling from a network of queues demands both confidence and correlation control. An important design goal is to be compatible with the *OMNeT++* framework without introducing changes to this code, as well as making it easy for existing simulators to make use of *oProbe*. The *oProbe* project introduces a probe module, which is a new simple module in the context of *OMNeT++*. A probe is the instrument that applies a controlled stochastic sampling technique. The probe module may have any number of probes. The *oProbe* project supports different interface levels according to the functionality required. Based on the FIFO queue example included in the sample directory in *OMNeT++*, the *oProbe* project provides example code on how the software can be used by existing or upcoming *OMNeT++* based simulators.

1 Introduction

In December 2006 we started a project that used open source code published on the Internet to build a simulation environment suitable for *steady-state* performance analysis of wireless communication networks. To hide the disparities between the different open source projects from the user at the left of the figure below, we built a dashboard (or a wrapper) between the user and the imported open source code.

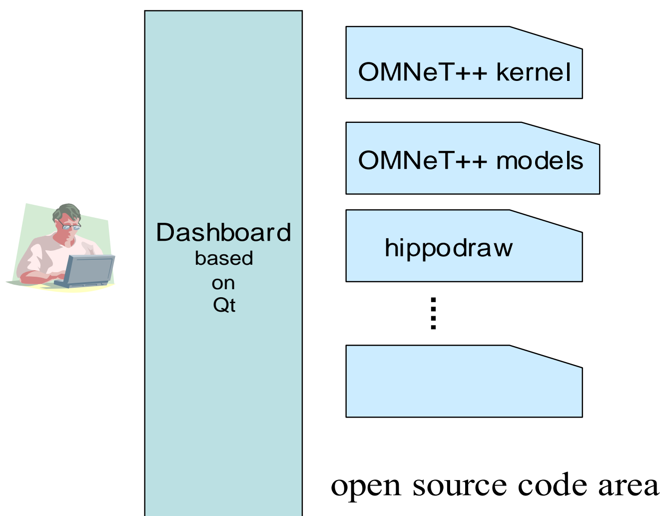


Figure 1.1 The dashboard provides a set of services to set up and run simulations.

The dashboard should provide the following services:

- editors for setting up model parameters (pathloss, radio data, traffic, etc.)
- functions for controlled statistical sampling techniques to produce reliable results of known quality (analysis of data in run-time is required)

- editors to produce reports from the simulation results

Our research interest is modelling of wireless networks with focus on protocols [5]. We have tested a number of commercial simulation software, but have rejected usage of commercial software due to lack of source code. Without access to the source code, we are unable to make modification to protocols, and often need access to the source to get exact and detailed information about the protocol behaviour. By using *OMNeT++* [1] we have the source code that provides the basic functionality for building simulators and have found it more attractive to put resources on developing enhancement to *OMNeT++,* when required.

The simulation community has been accused for producing many simulation studies of bad quality, see [4]. This article stresses the importance of producing statistically sound results - a principle we are used to from earlier research. Despite the high quality of *OMNeT++,* the software did not meet our requirements with regard to statistical analysis, so we developed some enhancements for setting up experiments using controlled sampling techniques at run-time. Using *OMNeT++* version 3.3 and some commercial software, we developed a simulator of a proprietary wireless network and conducted a number experiments. We wanted to contribute back to the OMNeT++ open source project, so we stripped off the source code assumed to be of general interest and put together a tar ball with source code and documentation. The resulting software is described in this document and is named *oProbe* - *OMNeT++* extended with probes.

The target group for this document is mainly software developers with focus on *OMNeT++*. However, the software providing editor functions, batch means analysis and read/write handling of XML-file may be of interest to developers using other simulation framework since this part of the software is independent of *OMNeT++*.

Many of our *OMNeT++* extensions are based on Qt version 3 from *Trolltech*, see www.trolltech.com. Why Qt3? The project team members had long experience with the Qt3. The Qt3 software is stable, is well documented, runs on many platforms and is easy to use. The Qt3 and the Qt4 software are very popular within the open source community, so the decision to use Qt was easy. The current *oProbe* version uses Qt3, but shortly we will migrate to Qt version 4.

This document is organised as follows.

Chapter 2 “Getting Started” describes how this software is meant to be used and demonstrates its capabilities. We recommend that you read this chapter first. If you find the software interesting and want to test it, we provide an installation guide in chapter 7 “Installation”. The FAQ section in the same chapter gives some hints on problems that may occur. The *oProbe* rely on other open source distributions, so it is likely that the FAQ section will be useful.

This document has focus on the *oProbe* software, usage and installation. The mathematical analysis of the batch means method implemented is not explained in detail, but section 2.2 “The effect of correlated samples” gives an overview.

An automatic documentation system (*doxygen*, www.doxygen.org) has been applied in this project, so this document only gives an overview of the implementation. The starting point for a detailed description of the implementation is the *doc/html/index.html* file. However, before you start to brows around, please read chapter 3 “The Software Architecture”, which explains the relationship between packages and classes, and the naming convention used for assigning file names.

The *oProbe* project introduces a probe module, which is a new simple module in the context of *OMNeT++*. Chapter 4 describes this module.

The *oProbe* project supports three interface levels with increasing level of functionality. The lowest level does not provide a GUI, but only functions to do statistics on sampled data at run-time. The different interface levels are outlined in chapter 5.

1.1 Terminology

OMNeT++
Open source software providing a framework to build simulators [1], including INET.

HippoDraw
Open source software for making graphical plots of data [8].

oProbe
OMNeT++ extended with probes

$OMNET - the home directory for the OMNeT++ installation.
$OMNETI - the home directory for the INET installation
$HIPPO - the home directory the HippoDraw version 19.1 installation
$HOME - the login directory

1.2 Acknowledgement

Thanks to the “gosikt-“project at the Norwegian Research Establishment (www.ffi.no) for providing the necessary resources to produce this open source project.

2 Getting Started

This chapter gives an overview of the basic functionality of *oProbe* and explains the meaning of the term probe. The first section in this chapter takes you through a complete simulation session, starting with the configuration of simulator input data and ending up with an output graph that presents the results. The second section illustrates the effect of correlated samples.

Now we turn to the design of experiments. A typical simulation experiment for us is to estimate the expectations of stochastic variables as some input parameters are varied over a set of values. This is exemplified by Figure 2.1 where the estimate of interest is the average end-to-end delay as function of the offered traffic taken from the set $\{\Lambda_1, \Lambda_2, \Lambda_3, \Lambda_4, \Lambda_5\}$. The index *i* of these elements is referred to as *session run i*. They are all independent of each other, which means that the simulator is set to its initial state at the very beginning of each run[1]. The offered traffic Λ_i will at run-time lead to creation of one or more traffic generators. Normally, the offered traffic set is ordered after increasing load since this eases the process of producing nice output graphs.

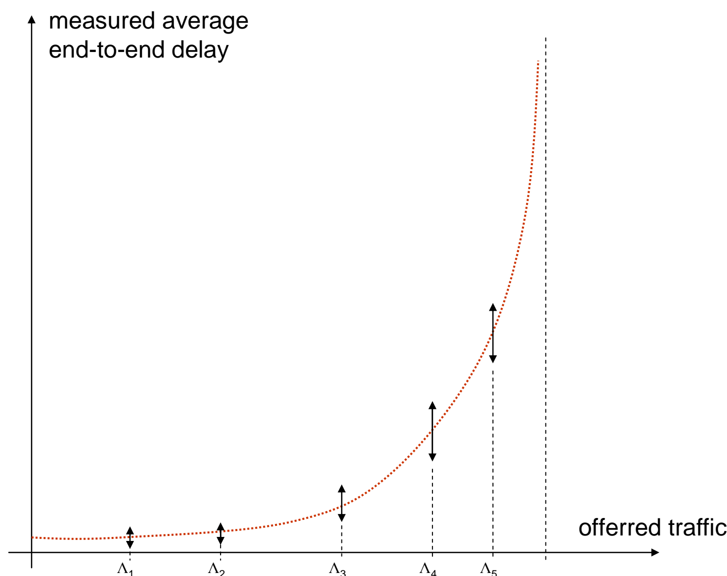


Figure 2.1 An example plot from an experiment.

To take measurements from a network you do need an instrument - we use the term **probe**. In practise, you insert a probe into your network by placing some specific statements in the simulator source code. The practical aspects are dealt with in chapter 4.

A typical simulation run will have many active probes taking samples from different probability distributions. Unfortunately, network simulations normally imply collecting time-variant and correlated samples. Probes must use controlled statistical sampling techniques to produce

[1] All queues are set to an empty state at time instance zero.

trustworthy results, and *oProbe* uses a classical batch means analysis technique to get control of the correlation. Section 2.2 gives a short description of the technique used. When the uncorrelated batch size is found, the probe starts to estimate the confidence interval for the first order moment (the expectation) of the underlying distribution.

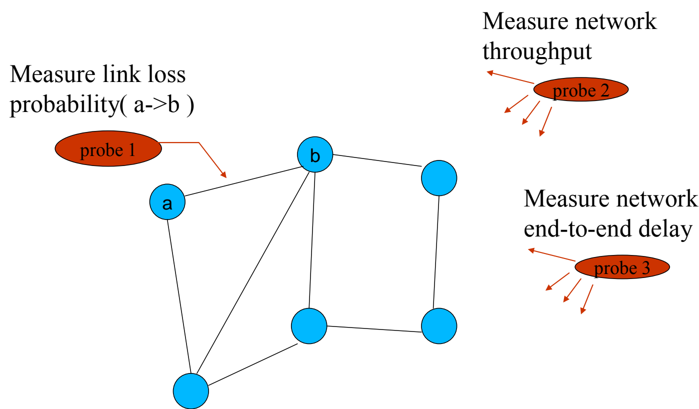


Figure 2.2 Probes are the objects that collect and process samples from probability distributions.

Sampling from probability distributions means that you must have control of the quality of the estimate. Figure 2.3 shows three hypothetical simulation runs under three different parameter sets. Given a specific confidence level, you can say set 1 gives less packet loss than set 3. However, you cannot state any difference between set 1 and set 2 since the confidence intervals overlap. You have to improve the accuracy under the same confidence level to state any difference between set 1 and set 2. In practise, this means that you must increase the sample size by running more simulations.

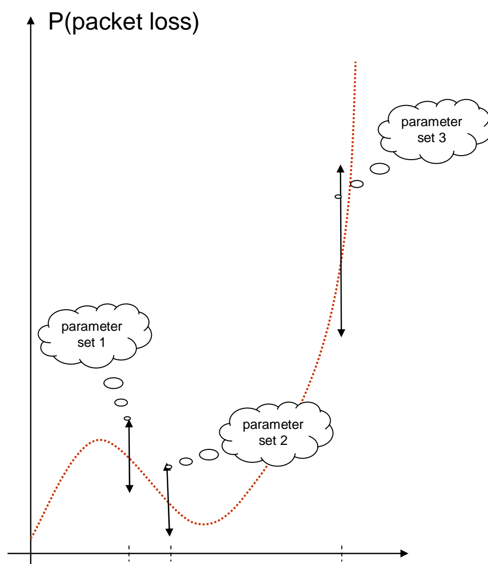


Figure 2.3 Illustration of confidence intervals (as vertical arrows).

The *oProbe* project defines the following three probe types:

Terminating: The probe shall produce results within a predefined statistical significance.

NonTerminating: The probe shall use a controlled sampling technique but shall not affect the simulation run length. Then it might be impossible to produce a probe report since the desired quality cannot be achieved from the sample size available.

SampleMean: Disable the controlled sampling technique but calculate the sample average.

All our simulation experiments use a set of probes by which we shall give some statements at a known precision, and the simulation process must run until the required accuracy is reached. These probes are activated as **terminating** probes and affect the simulation run length.

We must always be critical to the outcome of an experience due to the high probability of errors in the simulation input files and the possibility of software errors[2]. To detect these types of errors, we use a set of auxiliary probes - probes not directly of interest to the experiment but added to reveal suspicious network behaviour. For example, if I detect high packet loss in a network where I expect no loss then I once more forgot to tune down one of the traffic generators! These types of probes are activated as **sample mean** probes since they only calculate the sample average without using a controlled sampling technique. Moreover, we usually add some sample mean probes to get additional information to backup up our conclusions made from the terminating probe.

Upon termination, the probe module writes statistics to two files - one using a human readable text format (*probeOutFile*) and one based on XML (*probeOutFile.xml*) suitable for automatic processing. The last file is used by the report generator included in this project. The following lines show the *probeOutFile* from an M/M/1 queue simulation:

[2] We often insert invariant checks in our source code to catch software errors, for example assert(invar1() == true). It is better to halt the simulation than produce bad results!

```
state is Accuracy reached
Sample size:                    61000
Mean:                           6.09886
Half Width:                     0.487882
Final Accuracy:                 0.0799955
No of batches:                  243
Batch size:                     250
* Schmeisers confidence interval *
state is Accuracy reached
Sample size:                    61000
Mean:                           6.10509
Half Width:                     0.608542
Final Accuracy:                 0.0996777
No of batches:                  20
Batch size:                     3000
* Autocorrelation confidence intervals *
Lag_1:                          0.178896 +-0.212849
Lag_2:                          -0.0468146 +-0.219555
Lag_3:                          0.00804297 +-0.220007
```

The autocorrelation statistics (Lag_1, Lag_2, Lag_3) is calculated for the topmost confidence interval (number of batches 243 and batch size 250). The Scheimser's confidence interval [2] is less affected by correlated data, and this confidence interval is the preferred choice when we shall analyse the simulation results.

2.1 Using *oProbe*

To illustrate the capabilities of the software, this section goes through example 1 included in the *oProbe* project. The instructions for building the exe-file are outlined in chapter 7.

Example 1 simulates an M/M/1-queue by using the fifo source code included with the OMNeT++ distribution (*$OMNET/samples/fifo*). In this section we measure the response time[3] as function of the offered traffic. We are afraid to produce bad results, so we include two probes. The first probe named *ResponseTime* is defined as terminating with 10% accuracy at 90% confidence, and measures the average response time. The second probe named *Throughput* is included to catch errors and measures the traffic out of the queue. An M/M/1 queue cannot drop packets per definition (since it has infinite queue size). If the throughput deviates from the offered load then we have one or more errors in the input files, or in the source code[4]. This second probe is set to the type sample mean because it is only used to check for errors.

The following table expresses the response time as function of the packet arrival rate (λ) when the service time ($1/\mu$) is kept fixed at 1.0 seconds.

[3] The queuing time plus the service time.
[4] Of course, in this case we should have implemented a probe that measured lost packets since a few lost packets will not be seen in the throughput. However, we probably detect wrong setting of the traffic level.

run N	λ [packets/s]	E[U]
1	0.250	1.33 s
2	0.333	1.50 s
3	0.500	2.00 s
4	0.667	3.00 s
5	0.800	5.00 s
6	0.830	6.00 s
7	0.910	11.0 s

Figure 2.4 Average theoretical response time E[U] as function of the packet arrival rate λ.

The simulator shall be set up with seven different runs with arrival rates as specified by this table. The following sections guide you through the necessary steps to configure the input parameters, start the simulation and finally, plot the simulation results. The steps are sequentially numbered and can be divided into the following main groups:

Starting the program:	step 1
Configuration of input data:	step 2 to 5
Execution:	step 6
Producing simulation report:	step 7 to 8

Step 1 Start the program

Open a shell and give the input:

```
$cd oprobe/example1
./example1
```

The widget shown in Figure 2.5 should appear. The blinking LED at the upper right corner is a watch dog timer telling about the status of Qt's event loop. A green colour indicates a healthy condition, in opposite to red - an error has been detected.

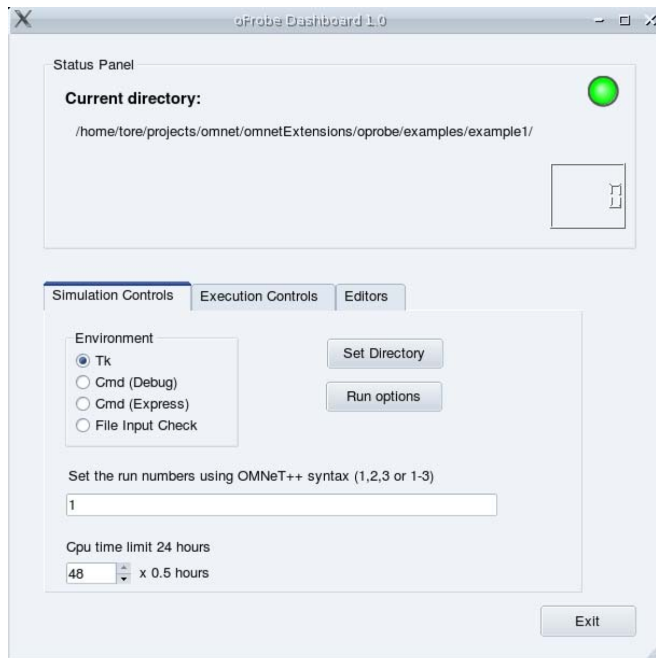


Figure 2.5 The main window for the oProbe application.

Step 2 Set the working path

The simulator needs some setup files and the working path must be set to the directory where these files are located. For example 1 the correct location is *oprobe/example1/omnetFiles*. Press the ”Set Directory”-button, use the widget that pops up to select this directory and then answer ”yes” to save this setting. Your selection is shown in the status panel and is also saved in the file *$HOME/.qt/omnetextensionsrc*. This becomes the default work path the next time the program is started.

The *OMNeT++* core demands a simulation setup file named *omnettpp.ini* and *NED*-files (cfr the *OMNeT++* manual), and these files reside under the directory named *omnetFiles*/*setup*.

Step 3 Activate probes

Example 1 includes a default probe specification file (*setup*/*probeInFile.xml*) which activates the probes needed. You may therefore skip this step, but you should activate the probe editor to check the consistency of your installation.

The editor for the probe module appears when you click on the Probe-button in the Editors-tab widget, see Figure 2.6. The probe table window displays the probes activated by the user and is initially empty. The “Insert Probe”-button brings up the complete list of probe objects and by selecting one of them, the selected item will be inserted into the probe table.

The normal probe editing sequence is as follows:

1. Insert a probe into the probe table: Click on the “Insert Probe”-button
2. Set probe attributes: Click on the probe name in the probe table
3. Save the probe data: Click on the Save-button

Step 3 creates/updates the file *probeInFile.xml*, which is parsed by the probe module when the *OMNeT++* kernel starts.

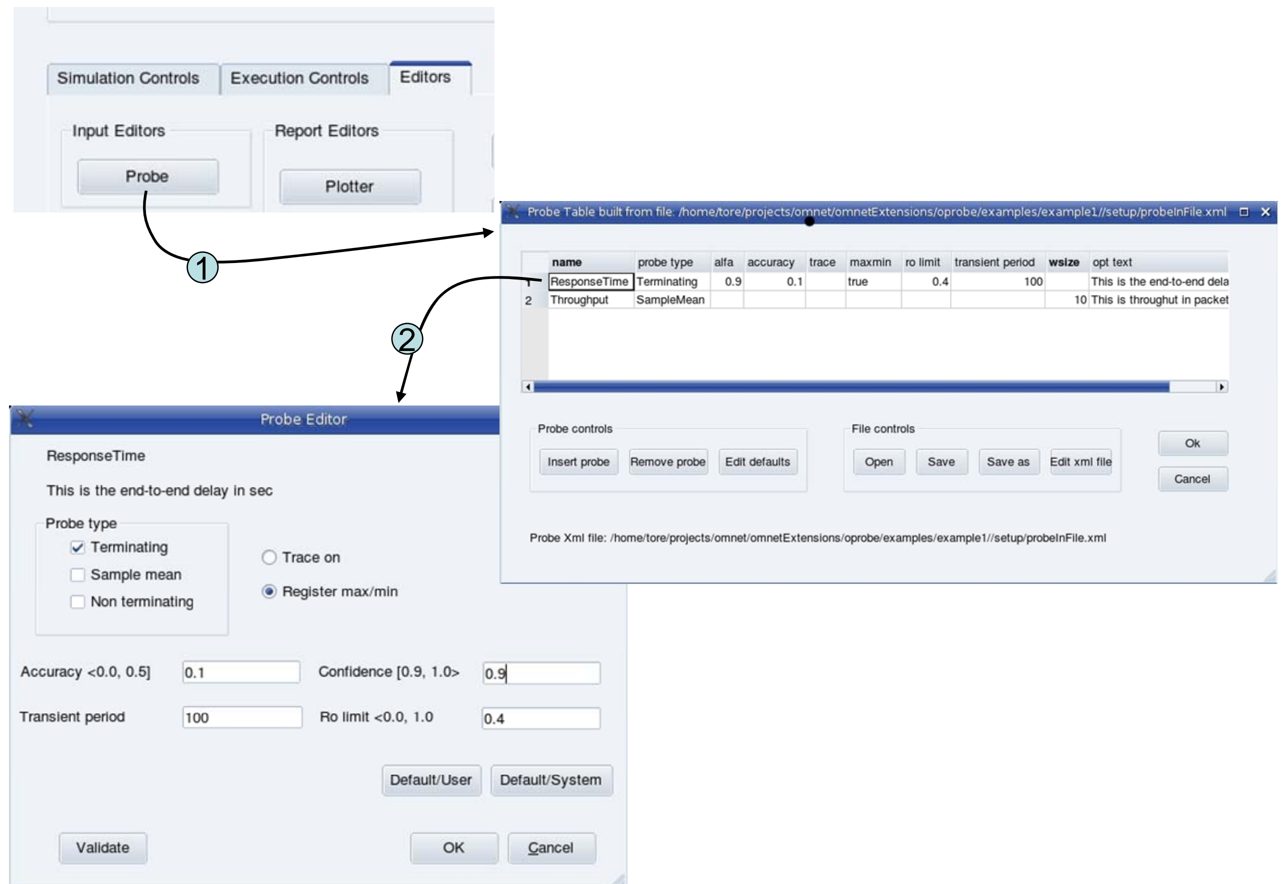


Figure 2.6 The probe table is activated from the main window (1) and shows the probes that are enabled. By clicking on a probe name (2), the probe editor appears and you may change the probe attributes.

Step 4 Specify the execution environment

Execution controls provide functions to set up the run-time environment for debugging, or for production. The following four run-time environments supported are: Tk-environment, Cmd (Express), Cmd(Debug) and File Input Check.

The Tk-environment is a graphical user interface provided by the OMNeT++ framework, see Figure 2.7. You get an overview of the model and may inspect different parts of the simulator.

When you run the simulator, animation of events is activated, and the text window shows a lot of trace messages. We suggest that you use this environment first to test example 1.

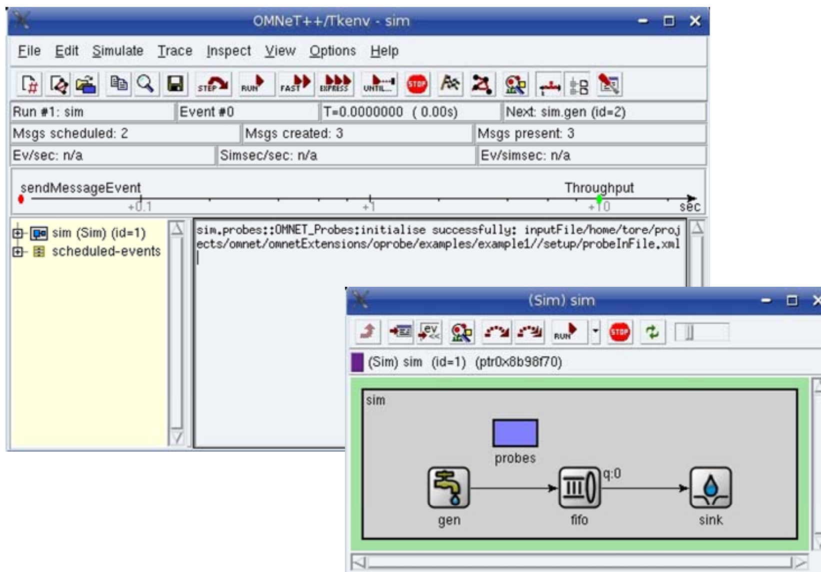
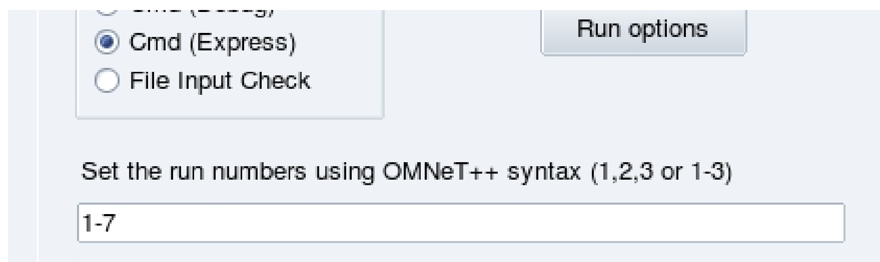


Figure 2.7 The M/M/1 queue running in the Tkenv environment. Press "Run" and the animation starts.

The Cmd (Express) is the run-time environment to use during the production phase - this is when you have validated that all the input data is correct and consistent, and that the simulation process will reach a steady-state (cfr section 6.2). Terminate the program by clicking on the Exit-button. Restart the program, select the Cmd (Express) environment and go to the next step. (The usage of the last two run-time environments (Cmd(Debug), File Input Check) is explained later.)

Step 5 Specify the session run numbers

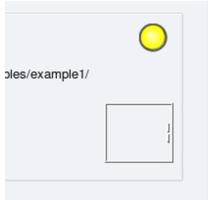
The *oProbe* project does not provide a traffic editor and the traffic generators have been specified in the input file *example1/omnetFiles/setup/simsetup.ini*. Just select the run numbers by typing "1-7" as shown below:



The dashboard provides only a window to setup the probe module. The other settings must be done through the text based files (omnetpp.ini and *.ned) as explained in [1].

Step 6 Running

The execution starts when you click on the Start-button in the ”Execution Control”-tab. The LED changes colour from green to yellow and the current run number is shown after some delay:



When the simulation session has ended, the LED becomes green again.

Step 7 Plotting

The dashboard provides a report editor supporting graphical plot of data and saving of plots to files. As explained before, the probe module produces two different output files. The report editor operates on an XML based file (*probeOutFile.xml*). Before you start to build a report, you should carefully scan through the output file *output/probeOutFile* looking for suspicious entries. Restart the program (step 1) and activate the report editor from the main window as shown in Figure 2.8.

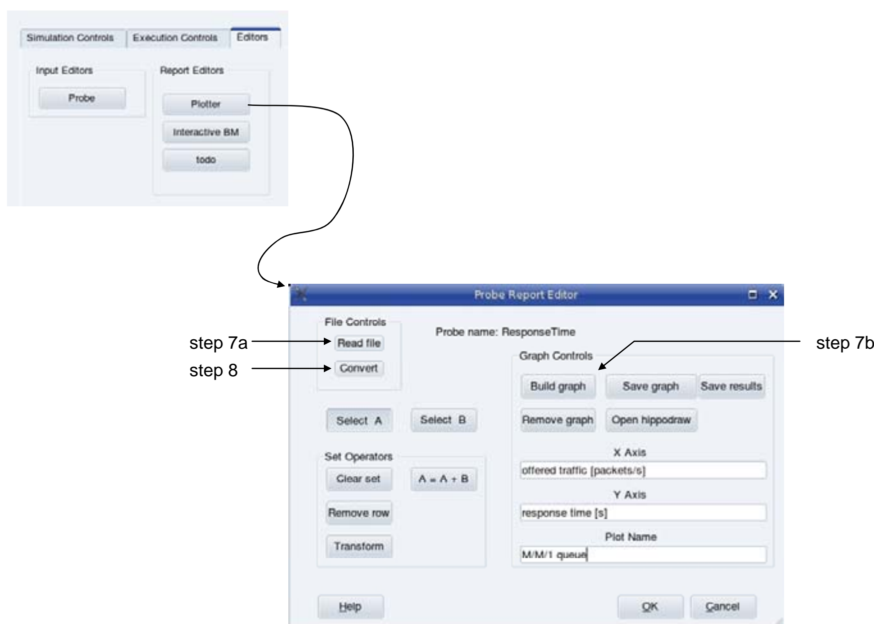
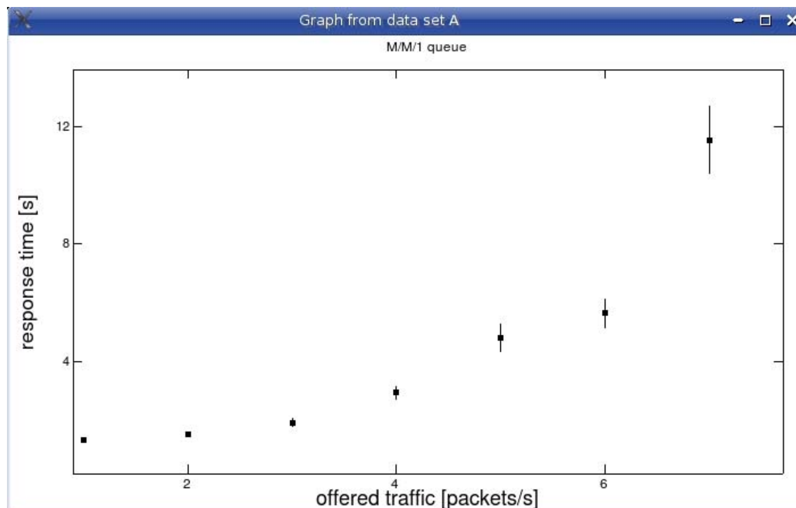


Figure 2.8 The Report Editor is activated from the main window.

Then use “File Controls”->”Read file” to open the probe report file (*example1/output/probeOutFile.xml*). Select the probe named *ResponseTime* and decorate the graph with text by filling in the lines within the “*Graph Controls*”-panel. If the program has been installed correctly with usage of *HippoDraw*, the following should appear on your screen:



Step 8 Exporting probe data

In our research we use *Mathematica*, www.wolfram.com, and have added functions to convert probe output data to a syntax accepted by this program. When we produce simulation reports, the reports combine results from different simulations and often we also include curves based on theoretical results. Select "File Controls"->Convert in the "Report Editor"-widget and open the probe report file (*probeOutFile.xml*). We want to plot the delay as function of the packet arrival rate, and the mapping between the run set and X-axis set for the M/M/1 queue is:

{1,2,3,4,5,6,7} -> {0.25, 0.333, 0.500, 0.667, 0.800, 0.830, 0.910}.

Answer "yes" to the question in the pop up window and fill in the transformation set:

Press Ok and select a file name (e.g., math.txt) and all the probes in the file *probeOutFile.xml* are now saved to this file. Here is an example:

```
responseTime:={
{{0.25,1.36698},ErrorBar[0.0676571]},
{{0.333,1.54679},ErrorBar[0.0672866]},
{{0.500,1.93905},ErrorBar[0.130872]},
{{0.667,2.95668},ErrorBar[0.21679]},
{{0.80,4.81411},ErrorBar[0.459481]},
{{0.83,5.64865},ErrorBar[0.498509]},
{{0.91,11.5382},ErrorBar[1.12785]}
};
```

The file *mathematica.txt* found under *example1* contains the complete code to produce the response time plot for the M/M/1. The plot is shown in the figure below.

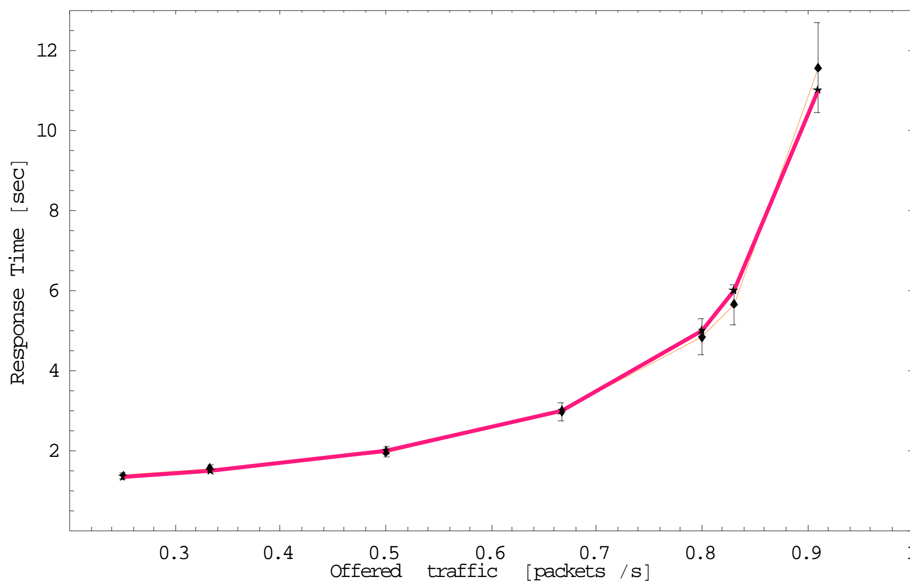


Figure 2.9 Simulated results exported to Mathematica and plotted together with the theoretical results (the red line). The file example1/mathematica.txt contains the complete code to produce this plot.

Reference [4] presents a survey of published papers on simulation studies and reports that 98% used plots to illustrate the simulations results. However, only 12% used confidence intervals on the plots. The study says nothing about correlation tests but we suspect only a fraction of the 12% even know about the correlation problem. The correlation problem is the subject of the next section.

2.2 The effect of correlated samples

This section is concerned with steady-state analysis of the average response time (queue time plus service time) of queues and the effect of correlated samples. Most communication networks have queues and we wish to give the reader an impression of the practical aspect of neglecting correlated samples.

The probe software uses a classical non overlapping batch means analysing technique starting with 100 batches. Each batch B_k is calculated as

$$B_k = \frac{1}{M_0} \sum_{j=1}^{M_0} s_{k,j}$$

where $s_{k,j}$ is the sampled value j in batch k. $B_1, B_2, \ldots B_n$ have unknown distribution, but if M_0 is large then the batches are approximately normal distributed.

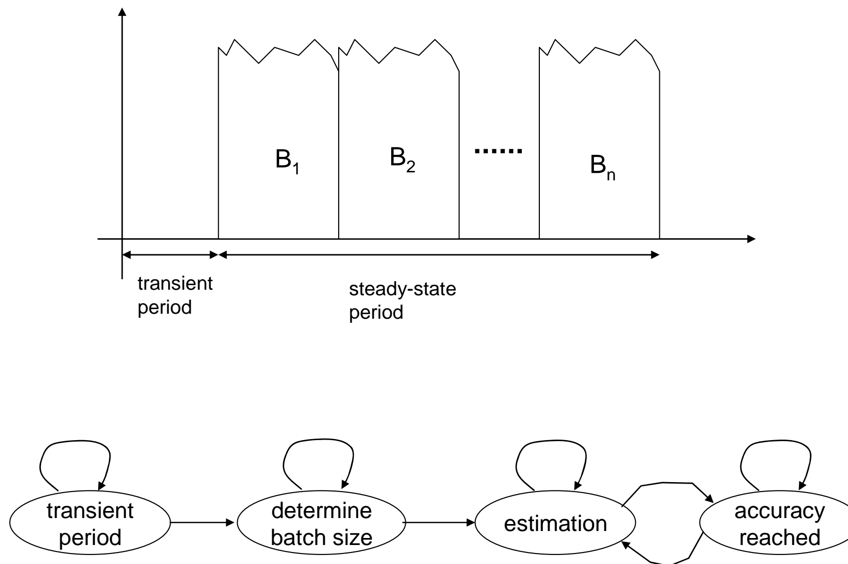


Figure 2.10 The principle of batch means analysis is illustrated at the top. The state diagram shown illustrates the states of a probe. The probe module prints out these states during execution when the user activates probe tracing from the probe editor widget.

A probe is set to an initial state when a simulation run starts and then changes state as the number of samples increases. The text below describes these states.

Transient period

All samples are discarded in this state.

Determine the uncorrelated batch size

Starting with 100 batches, each containing $M_0 = 50$ samples, the autocorrelation between the blocks is calculated. If the correlation is above the threshold *roLimit*, more samples are collected to get larger batches. When a batch size fulfilling *roLimit* is reached, the estimation phase is entered.

Estimation

The previous phase has found a batch size with acceptable rest correlation according to the user's input parameter *roLimit*. All the samples collected will be reorganised to batches of this size. The user has selected the accuracy and the confidence level for this run. The estimated

accuracy $\hat{\varepsilon} = \Delta B / \hat{\mu}$, where ΔB is the half width of the confidence interval and $\hat{\mu}$ is the sample average. The batch means software estimates the accuracy in run-time and checks against the user's accuracy threshold. When the estimated accuracy becomes smaller than the user's threshold value, the accuracy reached state is entered.

Accuracy reached
This state is entered when both the correlation and the accuracy threshold have been fulfilled. The batch means software signals that this probe does not need further samples. The current run may have many terminating probes that not yet have reached their accuracy limits, and the simulation must not terminate. The probe module continues to save and process incoming samples for this probe also in the "accuracy reached"-state. Due to the stochastic nature of this process, the estimation state may be entered again.

To illustrate the effect of correlated samples, we use the M/M/1 packet queue as the example of a communication network. This is beneficial since it has two parameters only, the packet arrival rate λ and the service rate μ, and we have an exact analytical expression for the response time E[U] (the queuing time plus the service time):

$$E[U] = \frac{1}{1-\rho}\frac{1}{\mu} \quad \text{where} \quad \rho = \lambda / \mu$$

We also know from the queuing theory that:

- the transient period increases with increasing traffic (remember we are interested in steady-state)
- the samples taken from the M/M/1 queue response time distribution are correlated

The practical justification of the correlation test is based on two simulation experiments using the M/M/1 queue. The queue parameters settings are:

service rate μ = 1.0 packets/sec.
arrival rate λ = 1/1.05 packets/sec.

With these parameters, we have a queue under high load $\rho = \lambda / \mu = 0.95$. We know from the queuing theory that the samples have a **high positive** correlation. The theoretical response time for this queue is 21.0 seconds.

The first experiment of the queue sets *roLimit* to 0.99 meaning that we neglect the correlation problem. The figure below shows the course of the confidence interval during the simulation. The solid green horizontal line marks the theoretical response time (21.0 seconds), and shows that the confidence interval does not enclose the true value during the entire simulation run. This experiment discarded the first 1000 samples only, the transient period is probably longer, so you should neglect the leftmost confidence intervals in Figure 2.11.

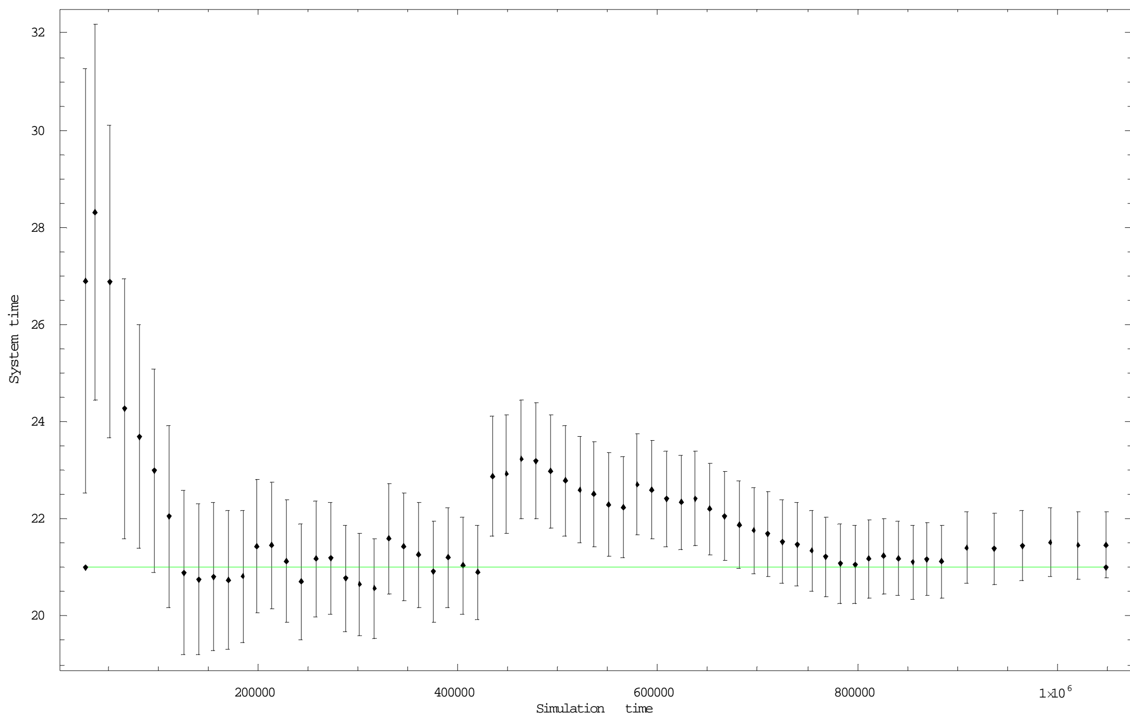


Figure 2.11 95% confidence intervals for the response time using batch means analysis with roLimit 0.99 (that is, the correlation control is disabled).

The second experiment changes one parameter only - the *roLimit* is set to 0.4. The experiment is repeated and Figure 2.12 shows that the confidence intervals cover the true mean during the entire simulation. The last run has fever but larger batches.

During a number of simulation studies we have found that the following parameter set gives a reasonable balance between statistically sound results [4] and run-time length:

Accuracy $\varepsilon = 0.1$, confidence level 0.9 and roLimit $\rho = 0.4$.

The relative frequency at which the confidence intervals contain the correct response time is named coverage [3]. The last simulation setup had an excellent coverage and the simulation could be terminated at any time instance and still cover the true value.

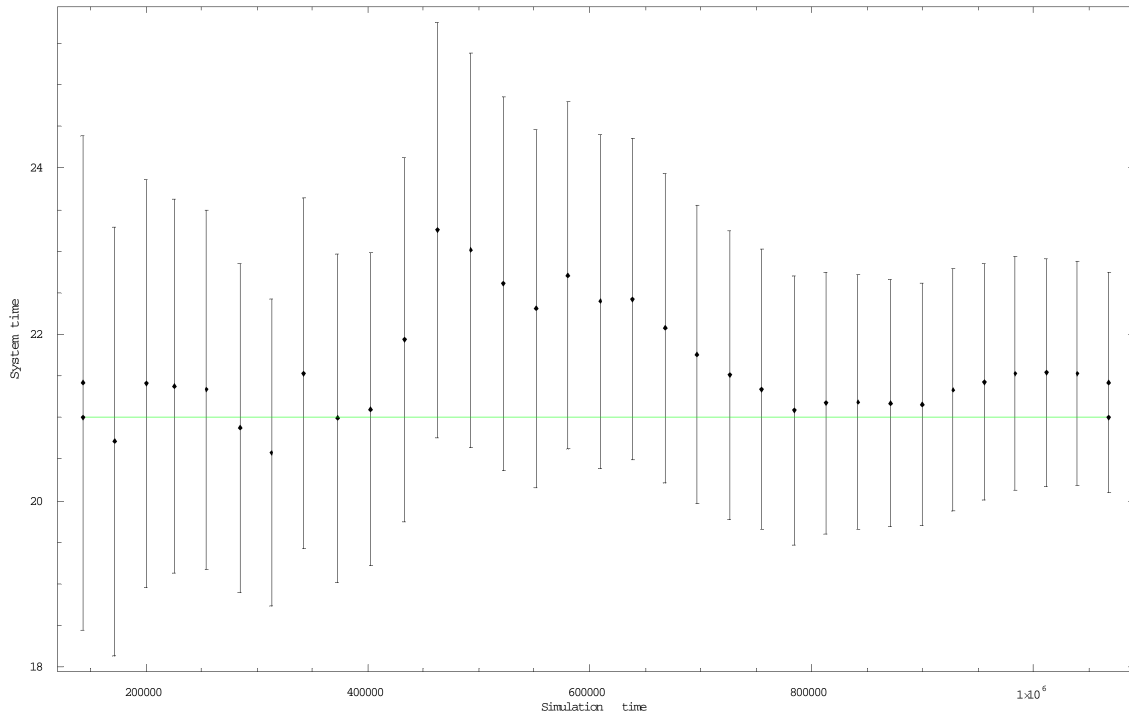


Figure 2.12 95% confidence intervals for the response time using batch means analysis with roLimit 0.4.

3 The Software Architecture

One objective of the *oProbe* project is to improve data analysis and execute control of the *OMNeT* ++ kernel. An important design decision is to try to be compatible with the *OMNeT++* original source code without introducing changes in this code. However, we are allowed to implement *OMNeT* ++ simple modules as long as we follow the principles given by *OMNeT* ++.

The analysis model below expresses that we are bounded by two external interfaces. A UI[5]-package provides widgets to the human user by which he can edit input parameters, perform execution control and handle output data in different formats. An *OMNET*-package contains the functions required to communicate with the *OMNeT* ++ program/software, and may be regarded as a wrapper package between the *OMNeT* ++ and our extension module.

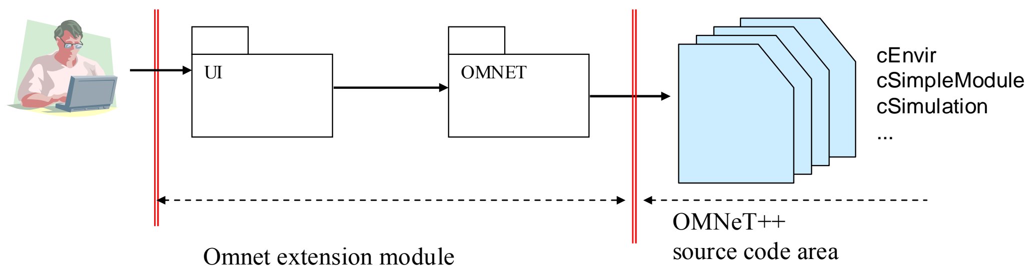


Figure 3.1 The analysis model for oProbe. The two most important interfaces to the external environment are shown as red vertical lines. The solid arrows should be read as "depends on" (e.g., the UI-package depends on the OMNET-package).

The *OMNET*-package is the user of the *OMNeT* ++ on behalf of the human user, and must use a number of *OMNeT* ++ classes to carry out this task. We must have a responsive user interface while the simulator runs and hence, we must have one additional thread to the main thread (the GUI-thread as called by Qt). Let the *OMNET_Thread* be the class that implements this second thread. It is natural to let the analysis class *OMNET_Thread* be the "main()" as seen by the *OMNeT++*. The most important *OMNeT++* class is then *cEnvir,* which is the *OMNeT++* user interface to the simulator. *OMNeT++* has one instance only of this class - a static global variable named *ev*.

We make use of the Qt Designer to build GUI widgets and rely on the Qt class library as much as possible. In contrast to many of the *OMNeT++* classes, most of the Qt classes are thread-safe. To implement our services we certainly need services from the *OMNeT++* kernel. This is accomplished by introducing a new simple module (cfr. the *OMNeT++* class *cSimpleModule*) as shown in the Figure 3.2.

[5] User Interface

The *oProbe* provides functions to the user for setting up simulation sessions. Data shall not be sent directly from the probe module to the *OMNeT*++, but shall be stored in files and read via the *OMNeT*++ kernel when the simulator starts (cfr *OMNeT++::initialize()*). The same principle shall apply to output data (cfr *OMNeT++::finish()*). The communication between the extension module and the *OMNeT*++ is then mostly based on files. The preferred file format is XML.

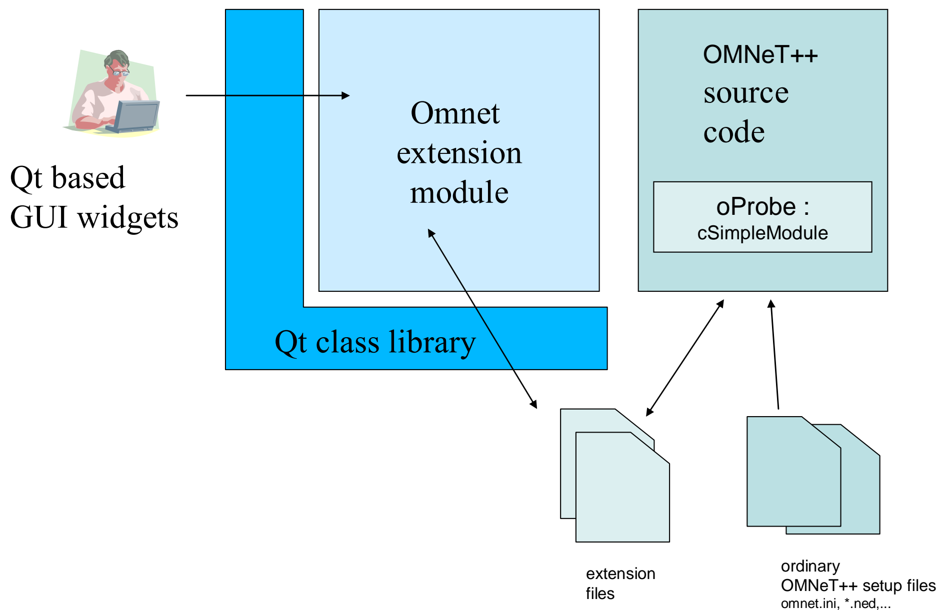


Figure 3.2 The oProbe software architecture.

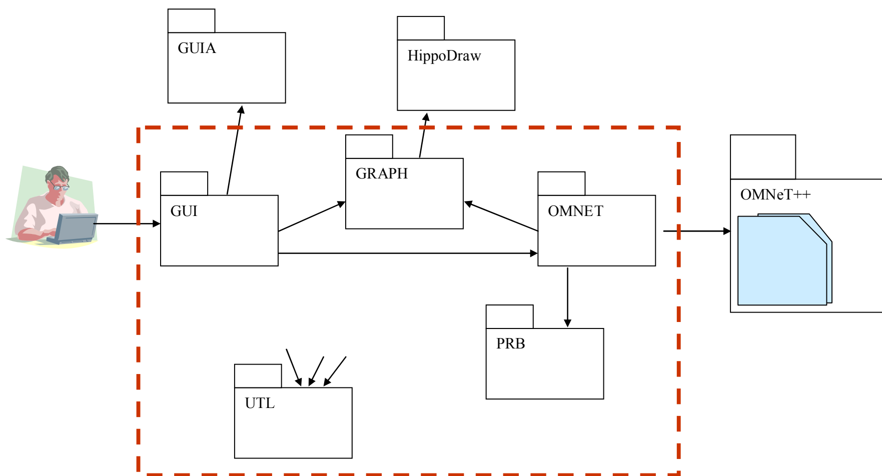


Figure 3.3 Design packages and their relationship.

The analysis model in Figure 3.1 is expanded to the design package diagram shown in Figure 3.3. The packages outside the red dotted rectangle are external packages. The analysis package UI (User Interface) is expanded to the packages GUI (Graphical User Interface) and GUIA (GUI Automatic). This project uses the Qt3 Designer for widget production, and the C++ classes produced by this tool belong to the GUIA-package. The GUI-package uses the GUIA-package to

build widget for interaction with the user. The GRAPH-package contains classes to plot data and is based on the open source project *HippoDraw*, see *www.slac.standford.edu/grp/ek/hippodraw*. The PRB (probe) package is a class set to handle probe input data and output data, and the controlled statistical sampling technique used to produce reliable results is placed in this package. The package UTL (Utility) contains classes of general usage, and is used by many internal packages.

In our research we build models including the entire OSI stack - spread spectrum radio models at the bottom and application layer protocols at the top. Yes, we need fast processors and are faced with large input sets. An important principle for us is to save all model input data together with the output data from the simulation experiments. We are then able to repeat our experiments later - we often do to check, among others, the effect of bug fixes. Figure 3.4 illustrates the basic design principle applied to fulfil this work principle. All input/output are saved in XML-files. The probe module reads the XML-file *probeInFile.xml* upon startup and produces the file *probeOutFile.xml* at the end of simulation. Most users do not like to edit XML directly, therefore we build editors to read and write XML files. For example, the class *GUI_ProbeEditor* is an editor tailored to the file *probeInFile.xml.* All interactions with XML-files go via DOM trees. This is easy due to the excellent XML API module provided by Qt [6].

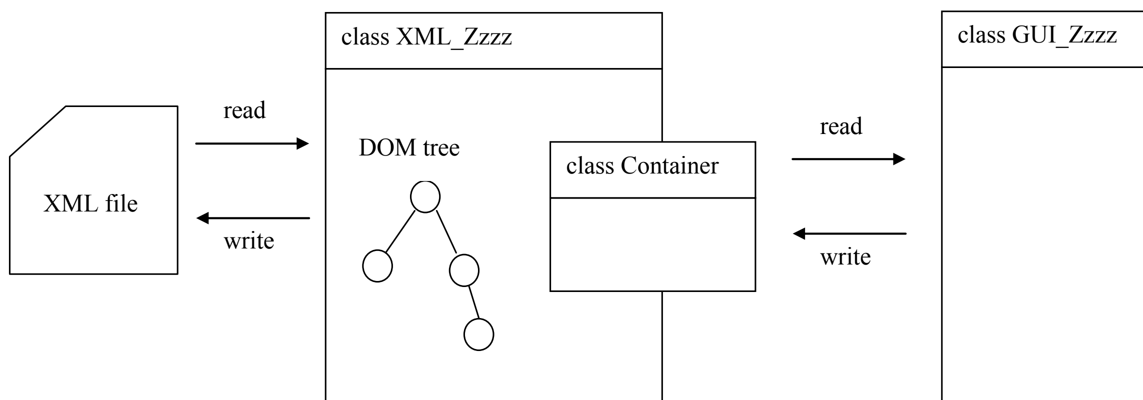


Figure 3.4 Creation and visualisation of XML file content.

The file and class name convention used for the *oProbe* project is to tag the names with a prefix to identify the design/implementation package they belong to. In our simple project, we do not find it practical to differentiate between a design package diagram and an implementation package diagram. When you see ZZZZ_zzzz, you know that this class belongs to the ZZZZ-package defined in Figure 3.3, while *zzzz* expresses something about its functionality. For example, *GUI_ProbeEditor* belongs to the GUI-package and implements a widget by which the user can edit probe attributes, hence “ProbeEditor”. *GUIA_ProbeEditor* is the corresponding C++ class produces by the Qt Designer, and this class shall never be touched by any editor!

3.1 The OMNET-package

The *OMNET*-package is the user of the *OMNeT++* on behalf of the human user, and in this perspective, the most important class is the *OMNET_Thread*. As the name says, this is a thread and is based on the Qt class *QThread*. The *OMNET_Thread::run()* is the main loop of the thread and this is the only point where the *oProbe* interacts with the *OMNET++* kernel (except for start-up where XML-input files are read).

The *OMNeT++* kernel looks for the file *omnetpp.ini* upon start-up. The *oProbe* builds this file automatically before the *OMNET_Thread* starts the *OMNeT++* kernel (*ev.run()*). The class *OMNET_IniFile* takes the user setting on the *oProbe* dashboard (e.g., "run in Tkenv") and inserts the corresponding statements in the *omnetpp.ini* file. The *oProbe* overwrites any existing *omnetpp.ini* file, but expects to find a file *simsetup.ini* instead, and copies the content of this file into the *omnetpp.ini.*

4 The Probe Module

This chapter gives an overview of the probe module, which is implemented as an OMNeT++ simple module (cfr the *OMNeT++* class *cSimpleModule*). The module is implemented by the class *OMNET_Probes*.

Since *OMNET_Probes* is a simple module, a file named *OMNET_Probes.ned* must also exist. The entire file layout to simulate the M/M/1 queue example from chapter 2 is shown below. The *probeInFile.xml* is the XML based probe set up file, but *oProbe* provide a widget to make this easy.

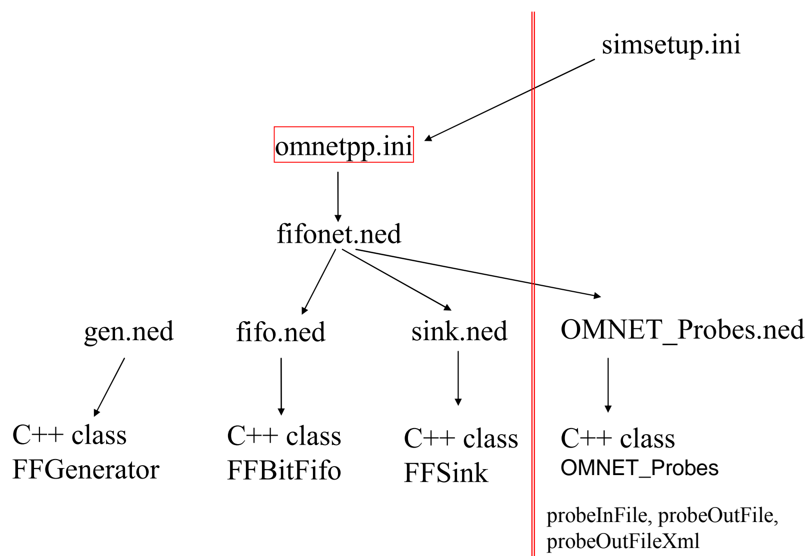
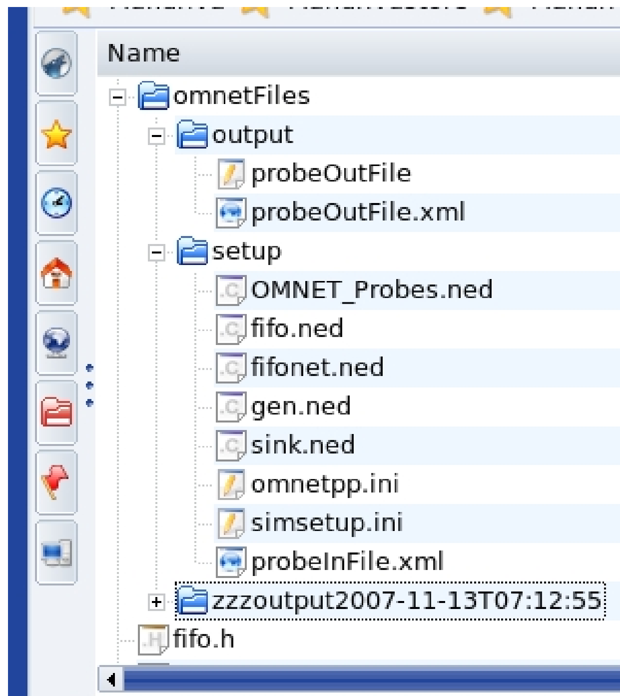


Figure 4.1 The file hierarchy for the M/M/1 queue example. The files to the right of the vertical line are probe specific files.

The probe module looks for input files under the directory named *setup* and places the output files under a directory named *output*. If an output directory with this name exists, it will not be deleted but renamed to prevent accidental loss of earlier simulation results. The probe module overwrites any existing *setup/omnetpp.ini* file, but expects to find the file *setup/simsetup.ini* instead, and copies the content of this file into the *omnetpp.ini* **before** the *oProbe* specific data (e.g. the run-time environment to use) is written to the *omnetpp.ini.* The top level simulation directory of example 1 in the "Getting Started" section is *omnetFiles* and the directory structure looks like this:



Here you see examples of the setup directory and two output directories. The directory prefixed by "zzz" is the renamed version of the old output directory.

Upon termination, probe statistics are written to two files - one in a human readable text format (*probeOutFile*) and one based on XML (*probeOutFile.xml*) suitable for automatic processing. The last file is the input to the report generator (*GUI_ReportEditor*).

The *OMNET_Probe* class is the manager of all the probe objects, and provides an API that other simple modules may use for sending samples to the probes. Each probe is identified by a unique text string - two probes cannot have identical names. The code section below illustrates how the probe named *ResponseTime* is used by the M/M/1 queue example.

```
void
FFSink::initialize()
//called by the omnet kernel on startup
{
001   ...your code
002   ...
003   responseTimeId = OMNET_Probes::getId( "ResponseTime" );
004   assert(responseTimeId > 0 );
}

void
FFSink::handleMessage( cMessage* msg )
//called by the omnet kernel when a message arrives
{
001   ...your code
002   ...
003   double d = simTime() - msg->creationTime();
004   OMNET_Probes::sample( responseTimeId, d );
}
```

FFSink::initialize() line 003 calls the probe module to get an identifier from the probe name. The motivation for introducing this step is improved performance since we do not want to make a string comparison for each sample (at *FFSink::handleMessage()* line 004). The variable *responseTimeId* in the class *FFSink* should be a private variable of type *OMNET_Probes::ProbeIdentifier*. Do not omit the assert statement at *FFSink::initialize()* line 004 because this simple test may detect many errors at low cost (e.g., spelling error in the string "ResponseTime", or a missing name in the internal probe object container).

The last line of *FFSink::handleMessage()* sends the samples to the probe module. The probe module performs real-time data analysis of *Terminating/NonTerminating* probes. When all the *Terminating* probes have reached their accuracy, the probe module will raise an exception to terminate the *OMNeT++* kernel. The *OMNeT++* kernel will then call the *finish()* method in all simple modules (i.e., all class that inherits the *OMNeT++* class *cSimpleModule*).

Probes are created by editing the file *PRB_ProbeContainer.cpp*. Open the file in an editor and insert a call to the *PRB_ProbeContainer::ProbeData()* constructor as shown here:

```
PRB_ProbeContainer::PRB_ProbeContainer()
{
  // Add your probe object here
  probeList << ProbeData( "ResponseTime", "This is the end-to-end delay in sec" );
  probeList << ProbeData( "Throughput",   "This is throughut in packets/sec",  1.0 );
```

The statements shown declare two probe objects. The first argument of the *ProbeData* constructor is a unique name for the probe and the second is an optional text string.

A rate probe (e.g, throughput [packets/s]) posses a special challenge because samples must be collected over a time window. Generally, event rates change during a simulation session (run 1, 2, ...) and the window size should be sized automatically to catch a certain number of events. The current version supports only change of window sizes before the simulator starts, either from the front end, or the XML based probe input file.

The current software release shows traces of source code for adaptive rate probes. However, this part of the code has been disabled since it needs more work, so our suggestion is to stay off the code. We have not deleted the source code since we use the functionality daily. Our ambition is to improve the quality suitable for public usage.

5 Interface Levels

This *oProbe* project supports three interface levels according to the functionality wanted and the distribution includes three example projects to assist users. The *oProbe* core software is built as a shared library and placed under *oprobe/lib*, and the header files are located in *oprobe/src*.

5.1 Level 1 (no GUI)

This is the lowest level where you want minimum functionality - you accept to edit the probe input XML file directly in a text editor, or by other means set the probe parameters. The directory *example3* of the *oProbe* tar file contains an example project for this case. None of the Qt Designer files (*.ui) is needed since no GUI shall be provided.

5.2 Level 2

This is an intermediate level where the user wants to use the probe editor functionality supported by this project. The directory *example2* of the *oProbe* tar file contains an example project for this case.

5.3 Full functionality

This is the case where the user wants all the functionality. The easiest way is probably to start out with the source code found under oprobe/*example1/src* and remove all files but the *main.cpp* and the *PRB_ProbeContainer.cpp*. Modify the constructor of the class *PRB_ProbeContainer* to create your own probe objects.

6 Tips and Tricks

The objective of this chapter is to communicate tips and tricks we have experienced by using *oProbe* in our modelling and simulation activities.

6.1 Checking input files

It is annoying to simulate for many hours and then crash in simulation run *i* due to an error in one of the input files. In general, simulators have many input files, and it is easy to set input parameters that are inconsistent, or assign illegal values. If some of the XML based input files are edited by a standard text editor, you have no syntax checks nor range checks.

To prevent this situation, the *oProbe* provides the function "File Input Check" shown in Figure 2.5. Select this function and then start the execution from the execution controls page. Each run defined on the run line of the simulation controls page is executed for a time limited period. This assures that all the input files are read and converted to internal data structures within the simulator. A number of events are executed and the simulator stops whenever an error is detected.

6.2 Checking the run-time length

Section 6.1 described a procedure to check the correctness of the input data with regard to value ranges and consistency. But will the simulation run forever when terminating probes are added? To get an answer to this, you should find answers to the following three sub questions by running some test simulations under the Cmd-environment:

Question 1: What is the lowest sampling rate among the terminating probes?

Question 2: Will the simulator reach a steady-state?

Question 3: Will the correlation be sufficient low in the steady-state?

The following paragraphs give some hints about these questions.

Question 1
Only terminating probes affect the run-length, so select one or more terminating probes that you anticipate will have a low sampling rate (e.g., a probe that measures the link delay on a single point-to-point link in a large network), and the session run which gives the lowest event rate for the probes. Use the probe editor to activate **tracing** and set the transient period to, say 100. Execute the single run in the Cmd-environment and observe the output trace. When the sample size passes 100, the probe module outputs "Transient period ended at". The probe module needs 5000 samples before data analyse is started and this gives you an estimate of the convergence speed.

Question 2
Select the run within the simulation session which gives the highest traffic load and execute it using the Tk-environment. Look at the number of created messages (cfr OMNeT++ cMessage) and continue the execution until the number reaches a steady-state. As long as the average increases, the process is in transient state, or you experience memory leaks :(.

Question 3
In a scenario with high load the most demanding stochastic variables are those who measure queuing delays due to the positive correlation problem in queues. Select a queue delay probe and activate trace from the probe editor. Then start the simulation using the Cmd-environment and observe the output trace. The first output is the "Transient period ended at". This gives you an impression of the sampling frequency of this probe. If the sampling frequency is satisfying, the next challenge might be correlated samples. The probe module outputs the correlation test results during tracing and you can follow its progress in run-time.

6.3 Performance issues

Probes disabled during simulations introduce small overhead since the *OMNET_Probes::sample()* returns nearly immediately. Enabled probes specified as non-terminating/terminating use batch means analysis and will therefore use processor resources. In our research we study complex network protocols with use of spread spectrum radios [5], so the burden introduced by the batch means analysis is insignificant.

A good practice is to redefine non-terminating/terminating probes as sample mean probes if you do not need the confidence and correlation control, and disable all rate probes you do not need. Note that disable means to remove a probe from the *probeInFile.xml* only, and not to remove it from the source code!

7 Installation

This chapter explains how to install the *oProbe* project. A prerequisite to succeed is that the following software components have been installed on the computer:

- Qt version 3.3.8[6]
- the omnetpp-3.3 include files and libraries

The build process assumes the following default paths:

$OMNET /usr/local/omnetpp-3.3
$OMNETI /usr/local/INET-20061020/Base
$HIPPO /usr/local/hippodraw

The FAQ section in this chapter explains the step needed if the omnetpp-3.3 files are located in other directories. *Hippodraw* is an optional component, but to have the full functionality you need it. The installation process depends on what functionality you want, and we have supplied subsections for the most likely cases.

The installation steps described herein have been tested on an "out-of-the-box" installation of *Mandriva 2007.0,* and the FAQ section contains a description of the steps. Also see the file *oprobe/mandrivaInstall*. No testing has been conducted on other Linux variants, nor on Windows.

7.1 Basic Build

The section assumes that *HippoDraw* is not required. Follow the steps below.

Step 1 *Unpack the tar file*

Go to the directory where you want to install the project and then execute the command "tar xvfz oprobe.tar.gz". The simulator configuration files and source code files are extracted from the tar ball.

Step 2 *Build the oProbe library*

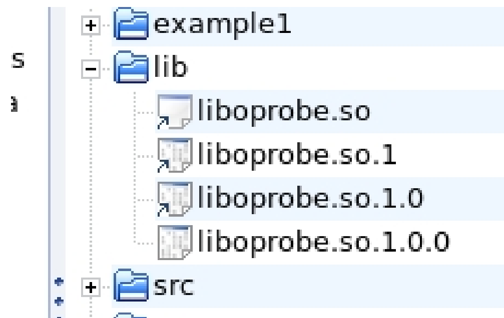
The previous step placed the *oProbe* source code under *oprobe/src*. Execute the following commands:

cd oprobe - you are now located at the top level of the *oProbe* project
cd src - you are now located at the source code directory of the *oProbe*

[6] The development was done under 3.3.8. However, *oProbe* may work under earlier versions, but this has not been tested.

qmake -makefile oprobe.pro - build the makefiles from the qmake file ".pro".
make distclean - remove files from earlier build
make - compile the source code and build the library named *liboprobe.so*

Check that the *oprobe/lib* looks like this:



If not, this step failed and you have to search for error message in the text stream. The FAQ section in this chapter may be useful. To bring the installation back to the qmake stage , just type "make distclean".

Step 3 *Add shared libraries*

Create a link to the *oProbe* lib directory as follows:
su
cd /usr/local
mkdir oprobe
cd oprobe
ln -s YOUR_INSTALL_PATH/oprobe/lib lib

The link to the location of *liboprobe.so* is now ready, and it is time to update the system file with this and the other shared libraries needed. Add the following directories to the system file */etc/ld.so.conf*:

/usr/local/omnetpp-3.3/lib, /usr/local/oprobe/lib, /usr/local/lib.

Then make the changes active by the command:
ldconfig

Step 4 *Build example1*

Build the example1 program as follows:

cd oprobe - you are now located at the top level of the *oProbe* project
cd example1 - you are now located at the source code directory for example1
qmake -makefile example1.pro - build the makefiles from the qmake file ".pro".

make distclean - remove files from earlier build
make - compile the source code and build the executable file *example1*

Step 5 *Start example1*

Start the program as follows:
cd oprobe/example1
./example1

Reference [6] describes qmake projects.

7.2 Build with HippoDraw

We recommend that you try a basic build before you include *HippoDraw*. If not, you must go through step 1 (unpack the tar file) as explained in the section ”Basic Build”. Then give the following sequence of commands:

cd oprobe/src - you are now in the source code directory of the *oProbe* project
qmake -makefile withhippo.pro - build makefiles that include *hippodraw*
make distclean - remove files from earlier build
make - compile the source code and build the library named *liboprobe.so*

Continue with step 3 in the section ”Basic Build”.

The C pre-processor macro that enables *HippoDraw* is *ENABLE_HIPPODRAW*.

7.3 KDevelop users

The *oProbe* development has been done using KDevelop and the KDevelop project files are included in the distribution. KDevelop users need only to open the KDevelop project files (**.kdevelop*) as they are used to. The development environment used in this open source project is KDevelop 3.3.6 running on the Linux version *Mandriva 2007.0*. The desktop environment is KDE release 3.5.4. Doxygen version 1.4.7 was used to generate html based documentation. Qt version 3.3.8 was used to develop the user interface and for handling XML-files. Currently, the software has not been tested on other platforms, but since we use Qt, it should be portable.

7.4 FAQ

7.4.1 Release/debug mode

The *oProbe* project compiles by default in debug mode. If you changes the “CONFIG += debug\” to “CONFIG += release\” in a qmake project file (*.pro) and runs qmake, the makefiles will build in release mode.

7.4.2 Changing default paths

Instead of making links, you may change the qmake input files. If, for example, *omnetpp* is not placed under */usr/local*, the build process will fail. Then you must edit the file *oprobe/src/oprobe.pro* (or *oprobe/src/withhippo.pro* if hippodraw shall be included). Open the file in your favourite editor and change the INCLUDEPATH elements to fit your environment. Then continue from step 1 as explained in the section "Basic Build". The qmake command will create new makefiles.

Of course, KDevelop users must never edit the file. They make the changes needed through KDevelop menus.

7.4.3 Installation on Mandriva

This section contains the building steps conducted to install all the software components on *Mandriva 2007.0*. The following components were installed from rpms: *bison-2.3*, *libtcl8.4-devel-8.4.13*, *libtk8.4-devel-8.4.13*, *doxygen-1.4.7*, *flex-2.5.4a*. Then we installed *qt-x11-free-3.3.8* from source:

```
$./configure -thread
$make
$su
$make install
```

To build *OMNeT++* we used:

```
$PATH=$PATH:/home/tore/software/omnetpp-3.3/bin
$export PATH
$./configure
$make
```

You must replace the red bold text to fit your path. When the build completed, we created the following links to the *OMNeT++*:

```
$su
$cd /usr/local
$mkdir omnetpp-3.3
$cd omnetpp-3.3
$ln -s /home/tore/software/omnetpp-3.3/lib lib
$ln -s /home/tore/software/omnetpp-3.3/bin bin
$ln -s /home/tore/software/omnetpp-3.3/include include
```

The *OMNeT++* installation was validated by running the *OMNeT++* FIFO queue example:

```
$cd samples/fifo
$LD_LIBRARY_PATH=$PATH:/usr/local/omnetpp-3.3/lib
$export LD_LIBRARY_PATH
$TCL_LIBRARY=/usr/lib/tcl8.4
$export TCL_LIBRARY
$./fifo
```

Installation of *INET* started by editing the INET configuration file *inetconfig* where we changed ROOT to *ROOT=/home/tore/software/INET-20061020.* Then the following commands were given:

```
$PATH=$PATH:/usr/local/omnetpp-3.3/bin
$./makemake
$make
```

Finally, we made links to the INET:

```
$su
$cd /usr/local
$mkdir INET-20061020
$cd INET-20061020
$ln -s /home/tore/software/INET-20061020/Base Base
```

The *HippoDraw* demands *graphviz*, which was installed from the rpm *graphviz-2.8-6mdv2007.0*. Then we unpacked the *HippoDraw* tar ball and built it:

```
$./configure --enable-boostbuild=no
$make
$su
$make install
```

All the external software required by the *oProbe* project is now installed. We then continued with the installation steps in section 7.1.

8 References

[1] OMNeT++,
www.omnetpp.org

[2] Bruce Schmeiser,
"Simulation output analysis: A tutorial based on one research thread",
Proceeding of the 2004 Winter Simulation Conference

[3] Edjair de Souza Mota,
"Performance of sequential batching based methods in distributed steady-state stochastic simulation", www.edocs.tu-berlin.de

[4] Kurkowski, Camp and Colagrosso
"MANET Simulation Studies: The Incredibles",
Mobile Computing and Communication Review,
Volume 9, Number 4

[5] Berg, et.al,
"Spread spectrum in mobile communication",
The Institution of Electrical Engineers, 1998,
ISBN 0-85296-935-X

[6] Alan Ezust and Paul Ezust
"An introduction to design patterns in C++ with Qt 4",
Prentice Hall 2007, ISBN 0-13-187905-7

[7] Qt Centre,
www.qtcentre.org

[8] HippoDraw,
www.slac.standford.edu/grp/ek/hippodraw