Simulating Queueing Networks with OMNeT++

Nicky van Foreest

April 3, 2002

Abstract

This documents aims at providing a quick introduction to simulating queueing networks with OMNeT++. It should get the interested user up and running with a simple M/M/1 FIFO queueing demo. After this the user can start exploiting more of OMNeT++'s functionality based on the extensive user manual included in the standard distribution. The software that comes with this document is a rudimentary library of OMNeT++ functions useful for queueing simulation.

Contents

1 Introduction

OMNeT++ is a discrete event simulation environment based on C++. András Varga is the principal author and currently maintains it.

When I started to use it for simulating queueing networks, I was somewhat awed by the amount of functionality that OMNeT++ provides and the size of the user manual. Besides this, I had no experience with C++ programming, which was an extra hurdle in becoming acquainted with the power of the tool. Hence I had to invest some time to find out which parts and functionalities of OMNeT++ are specially useful for queueing systems analysis, and how to get it working. My hope is that after you have read this document the burden of learning OMNeT++ is somewhat lessened and that your appetite has grown towards exploiting it for your own researches in the realm of queueing networks.

Summarized my goals of this document and software are to:

- provide a start for queueing systems simulation with OMNeT++
- give some useful pointers to the user manual of OMNeT++
- provide a start of an extensive software library that enables users to quickly set up queueing simulations.

In this tutorial I assume that the reader has experience with programming, at least with C, and has some basic understanding of queueing theory, say the first few sections of the chapter on queueing theory of [**?**]. Furthermore, I will not explain the details of the code. These are for the reader to study. This may seem like a thread, but the code is not hard to understand.

Perhaps I should have started by mentioning the advantages of using OMNeT++. I refrain from doing this at this point of the document, but instead postpone it to the end. The reason for this is twofold. In the

first place the OMNeT++ user manual and the OMNeT++ home page mention plenty of examples. In the second place, I want to focus on my personal experiences, and this is less important than having you starting to play with OMNeT++ and this demo.

1.1 Where to get the software?

You can find OMNeT++ at `http://www.hit.bme.hu/phd/vargaa/omnetpp.htm` If you are too lazy to type this in, a search with `google.com` on `omnetpp` will give an instant hit.
If you want to make advanced plots of the simulation results, make sure to get gnuplot: `http://www.gnuplot.org`

1.2 Feedback

Please do not hesitate to send me your comments, especially if you think it contains inaccuracies, unclear passages, etc. When you have something to contribute to the software library, please inform me so that I can include it in this package. I will grant full acknowledgment with respect to authorship.

My email address is: `n.d.vanforeest@math.utwente.nl`

OMNeT++ related questions can be posted at the OMNeT++ mailing list: `omnetpp-l@it.swin.edu.au`. Chances are high that András himself answers your questions.

1.3 Structure of this document

First of all, in chapter **??**, I want to discuss a working example of a queueing simulation in OMNeT++. The implemented example is the archetypal M/M/1 FIFO queue. The example will be slightly baroque in terms of its output–it will be more than you will typically need—but I want to show as many features of OMNeT++ as possibly appropriate with this one example. Chapter **??** will hint upon some ways to change the way the simulation works, the simulation parameters, the simulation output, and the like. Chapter **??** will be devoted to understanding the implementation of the example. The last section lists a few of the interesting ways in which the current example can be extended so as to turn it into a full-fledged library of queueing simulation tools.

1.4 Acknowledgments

András: Thanks a lot for OMNeT++.

1.5 Versions

1. Jan 2001
2. May 2001

2 Simulating the M/M/1 FIFO queue

2.1 Getting the simulation to run

The first step is of course to install OMNeT++. Before running the M/M/1 example it is a good idea to check that OMNeT++ and all the

software it depends on is installed properly. I usually test this by running one of the standard samples included in the OMNeT++ distribution, such as the Nim game.

Once you are convinced that OMNeT++ works, run `opp_makemake -f` in the directory that contains the source files, supposedly `queues`, to make a `Makefile`. The binary **opp_makemake** should be provided with OMNeT++. In case it cannot be found, try running it directly from the `src/utils/` directory of the OMNeT++ distribution. Once you have a `Makefile`, a simple `make` should do to get the binaries for the FIFO simulation.

A general remark is of importance here. In case you change something in one of the files, you should run `make` again. When you decide to copy all files to another directory, or include other files in this directory, run `opp_makemake -f` again. As an aside, run `make clean` to remove all object files and binaries.

The binary to run has the same name as the directory that contains the Makefile that was generated by `opp_makemake`. In my case this is `queues`. Run `queues`. If everything works the way it should, a few windows should pop up.

I consider it a bit too much work to type in `make` and then the binary again and again. Therefore I included a very simple script `run` which does this for me.

2.2 Running the demo

Once the simulation has started you should see two windows:

- A graphics window showing a small queueing network: a job generator, a FIFO queue, and a job sink.
- The other, called `OMNeT++/TKenv`, containing the main functionality to run the simulation.

Double click on the FIFO widget on the canvas in the graphics window. A new box should appear, called `(Fifo) fifonet.fifo[0]`. Click on `Objects/Watches`. You will see a few lines with text appear. The line with the string `(cQueue)` contains the word `(empty)`. This will change to the number of customers in queue (= one less the number in the system when a customer is served) once the simulation is running. Then double click on the line with the word `Job Distribution`. You will see a blank blue (that is, on my screen) window. Here a histogram will show the probability density function of the number of jobs in the system as seen by an arriving job. Finally, click on the line `Jobs in system`. A yellow canvas will come up. This will contain a simple plot of the number of jobs in the system, the one in service included.

Now you can start the simulation by pressing the `RUN`-button in the `OMNeT++/TKenv` window. In the graphics window you should see jobs hopping from the generator to the queue, and from the queue to the sink. Press `STOP` after several seconds. The first line of the `(Fifo) fifonet.fifo[0]` window will now, with probability > 0, show that the queue contains some customers. When you double click on this line, another window will tell you which jobs are currently waiting in the queue.

To be able to distinguish between individual customers may seems not very interesting from a queueing perspective; the M/M/1 queueing model assumes that customers do not have real identity—they only differ by their service requirements and arrival time. However, when studying

more complicated models, for instance when jobs have different type, this information becomes relevant. It becomes even more important if you want to study protocol and queue interactions, such as when multiple TCP sources share buffered resources, i.e., routers.

The histogram window should now contain a few black bars. When you put your mouse on one, it will change color. (Mine becomes gray.) There is a line at the bottom of window that contains the number of arrivals that observed a certain number of jobs in front of it in the system. In other words, if cell # 1 contains 10, this means that 10 arrivals saw one job in the system.

Clicking on the `Job Distribution` line in the `(Fifo) fifonet.fifo[0]` window with the right mouse button, enables you to choose between a graphical representation of the gathered data, and a textual one, called `Object`. I discussed the graphical one above. The text information will provide you with aggregate statistics such as the mean number of jobs found upon arrival, etc.

The last step will be to press `EXPRESS` in the `OMNeT++/TKenv` window; I leave the other buttons for you to discover. In fact, my best general advise is to press on any buttons you may see, and find out what they do. Do not forget that often both mouse buttons, the left and the right, can be used for different effects.

2.3 Analyzing the simulation results

If you have not interrupted the simulation by pressing on one of the big red `STOP` buttons, the simulation will finish after having generated 5000 jobs. On my machine this takes less than a second, and it is not a particularly fast one.

Notice that the textual and graphical representations of the histogram are updated after the simulation has finished. The `OMNeT++/Tkenv` window contains as well a number of results. You might need to scroll a bit up and down in the window to view all the simulation results.

The directory will now contain a few new files as well.

- `fifo.sca` This file contains the statistics that were gathered in the variable `jobDist` of class `cDoubleHistogram`, see the user manual section 6.13, such as the number of jobs generated, the histogram data, etc. Please have a look at it now. The textual info of the histogram should be contained in this file too.
- `fifo.vec` This file contains info about the dynamics of the number of jobs in the system. You should process this with the OMNeT++ tool `plove`. Chapter 9 of the user manual contains the instructions. Briefly, start up `plove`. The left most button allows you to `load` the file. Send it to the right window with the arrow button in the middle. Press `PLOT!`. Now `gnuplot` should fire up and show the dynamics of the job distribution. You may have to fiddle around with the `options` to get a nice graph.

In general you can process these files, with `awk` for instance, or another C++ program for that matter, to postprocess it. You might instead want to do the post-processing in the function `Fifo::finish`, about which I will talk later. Here comes in the some of the power of OMNeT++'s use of a real programming language: you can incorporate your own post-processing functions in the simulation itself, in case you want to.

Summarizing, OMNeT++ provides a number of ways to present simulation data:

- dynamically, by means of jobs jumping from one node in the network to another;
- graphically, by means of dedicated collection classes such as `cOutVector`;
- verbally, by means of the `ev <<` statement in the code which is output to the canvas of the `OMNeT++/Tkenv` window;
- by means of files.

3 Modifying the basic example

The above discussed only one type of queue, the M/M/1 queue, with one specific parameter setting. You probably want to simulate other types of queues, more generic service distribution, etc. Another interesting degree of freedom is to speed up the simulation by leaving out the windows.

3.1 Changing the simulation

I will discuss some ways to change and extend the current simulation environment. The most important way to do this is with the `omnetpp.ini` file, see chapter 8 of the user manual for all details.

Changing the interarrival and service rate Change the numbers in `omnetpp.ini` related to the exponential distribution. Mind that the interarrival rate and service rate are the inverse of the parameters you specify, e.g. `fifonet.fifo[1].service_time = exponential(2)` means that the expected service duration $= 1/\mu = 2$.

You cannot only change this via the `omnetpp.ini` file. Another possibility is to press `Params` in the `(Fifo) fifonet.fifo[0]` window. Click for instance on the line with `service_time`. Now you edit the field in the window that will appear, for instance try another distribution, see the next paragraph. Do not forget to press the `Enter` key on your keypad to make the edit effective.

Changing the interarrival and service distribution OMNeT++ provides a few standard distributions, such as the uniform and exponential distribution, see sections 4.9.6 and 6.13 of the user manual. In case you need other distributions, you have to build them yourself. This is in itself quite interesting, see Ross [**?**], and, of course, Knuth[**?**]. Once you know, mathematically speaking, what to do, OMNeT++ makes the implementation very easy. You should define your new distribution in the file `distributions.cc`. I have already provided a few simple examples. Note that you can use the standard distributions of OMNeT++ in your new functions right away. Once you have implemented your new distribution, you can simply use its name in `omnetpp.ini` and pass appropriate parameters to it. Explicitely, if you want to use, as an example, the distribution `perturbedExponential`, which you can find in the file `distribution.cc` as the service distribution, than you should use the following line in `omnetpp.ini`: `fifonet.fifo[0].service_time = perturbedExponential(2,1)` Take any parameters you like, for this distribution, but check out the code first in case you do not want to be surprised during the simulation.

Changing the network The `omnetpp.ini` parameter `fifonet.num_buffers` enables you to make a network of tandem queues. Changing it from the current value 1 to say k, will put k queues in tandem. You should as well change the line `fifonet.fifo[0].service_time = exponential(2)` to `fifonet.fifo[*].service_time = exponential(2)`, that is, the `0` has to change to `*` to reference all fifos, instead of just the first one with id zero. I actually chose a slightly different approach here. First I give `fifo[0]` its value, and then the others by means of the `*`.

You will find included as well a ring network. This is built out of simple fifo queues that are connected sequentially to each other. If you want to run it, you have to change a few lines in `omnetpp.ini`. This file contains where the changes should be made. Since the ring is a closed queueing network, there are no external arrivals of jobs, neither sinks. Therefore these two are not included in the ring. Furthermore, the buffers should contain some initial number of jobs, that will start circling around. You can set these numbers by the parameter `ring.fifo[*].num_init_jobs = 20` to be found in `omnetpp.ini`.[1]

More general networks are for you to build. Section 4.10 and 6.19 of the manual will tell you how. You should as well consult the following OMNeT++ samples. The sample directory `token` provided with the standard OMNeT++ distribution shows a circular network. The `fddi` sample shows a large network. (Do not forget to click on one of the rings to see how complicated networks you can actually simulate with OMNeT++.)

Changing the job scheduling Only FIFO scheduling is implemented at the moment. In case you are interested in building other ones, go ahead.

Changing the random number generator For the more suspicious of you, section 6.9 of the OMNeT++ user manual discusses the random number generator. You can replace this with your own if you want this.

3.2 Speeding up the simulation

If you are convinced that everything works the way it should, and you are just interested in numerical output, you can run the simulation straight from the prompt with the `cmdenv` mode. No more windows will appear, only the output files will be produced. Due to this, the simulation will become quicker as well.

To achieve this, change these lines in the `Makefile`

```
# User interface (uncomment one) (-u option)
#USERIF\_LIBS=\$(CMDENV\_LIBS)
USERIF\_LIBS=\$(TKENV\_LIBS)
```

to

```
# User interface (uncomment one) (-u option)
USERIF\_LIBS=\$(CMDENV\_LIBS)
#USERIF\_LIBS=\$(TKENV\_LIBS)
```

[1] For the interested. If you analyze the expected number of jobs in the queues, and add them, it will appear as if one job is missing, i.e, this is the application of the Arrival Theorem, see e.g. [**?**].

Do a `make clean` and `make` to remove all window related code from the simulation executable. Now it should work. Be aware that a new `opp_makemake -f` reverts the makefile to the old situation, i.e., the simulation with tk windows.

You will want these options in the following section of `omnetpp.ini`:

```
[Cmdenv]
runs-to-execute = 1
module-messages = no
verbose-simulation = no
Display-update = 1h
```

Play with these options to discover that the number of lines of simulation output will be a bit to much to handle. See section 8.5 of the manual for more info.

The user manual contains as well some more hints to speed up the simulation still further, see sections 6.18 and 8.7.

4 The implementation of the M/M/1 simulation

Before starting the main subject of this chapter, I need to define one concept: a *functional entity*. A functional entity is a part of a simulation that carries out a specific action on a job, or a message. For example, a server or a message generator are functional entities. They are, so to say the essential functional units that take care of one process step in the lifetime of a job, or message. The implementation of such entities will be called *modules*. Now back to the simulator.

The simulation environment is built out of, mainly, two types of files. The `.ned` files roughly describe how the entities should communicate; the `.cc` files contain the C++ code that implement the behavior of the entities. These file types I will discuss in some more detail below. I expect you to have the `.ned` and `.cc` files belonging to this demo at hand.

After having read this, be sure to check out the discussion of the Nim game in the manual as well, and the samples `fifo1` and `fifo2`. They are instructive and show additional functionality of OMNeT++ relevant for queueing systems analysis.

4.1 A basic approach to understanding the code

There are various ways to try to understand the implementation of a new simulation, for instance, this demo or one of the sample simulations. I found the following approach the most useful. First I run it, of course, to get an understanding of where the various entities reside and how they exchange jobs. Then I work through the files belonging to each entity separately. I start with reading the `.ned` file to understand what goes in and out of a module, and the parameters it will need. Then I give the header files a brief look to become familiar with the module's specific internal variables and functions. Finally I study the C++ code belonging to the module. Once I somewhat understand what it does, I tackle the next module in the chain, that is, the module that gets its messages from the one I studied. Working this way, I gained a quicker understanding of what was going on than by first working through all `.ned` files, than all the C++ files, etc.

In this document I will not, however, follow the above suggestion, mainly for brevity. Here I only want to illustrate certain key points of the simulation, and leave the studying for you.

4.2 The `.ned` files

Each entity in a simulation needs to communicate via *messages* with itself and other entities. Messages can be used for various purposes. One is to represent jobs that need service at queues. Another is to convey information of the entity's state to itself in the future, or to other entities.

Specifically, the fifo example contains three entities: a source module, a fifo module and a sink module. The source module generates messages that represent jobs. These job-messages are sent to the fifo module. This in turn delays the messages according to the present queue, then services them, and finally sends the messages to the sink module. The sink module processes these messages to extract some final statistics. The sink is where the messages leave the queueing network. The sink module releases as well the memory allocated to the job messages.

Clearly all these modules need in- and output gates to receive and send messages. The `.ned` file specifies these gates. In our example, the fifo module has an input and an output gate. Furthermore these modules may need parameters such as service rate, queue size, etc. These parameters need be *declared* in the `.ned` files too. The *definition* of the parameters, i.e., giving them a value, takes place when the simulation starts.

The next step is to glue all the modules together to form a network. This should be done by means of another `.ned` file that defines a *compound module*. This new module uses the simple modules and connects them into networks. Besides this it passes parameter values on to them during the simulation. Finally it gives some directives on where on the canvas the simple modules should appear. To state this in terms of the fifo queue, `FifoNet`, defined in `fifonet.ned`, connects the output of the source to the input of the fifo queue, and the output gate of the queue to the sink. Consult `fifonet.ned` for all details. Take especially notice of the syntax, and be aware of when to use a comma ',', and a semicolon ';'. I made some time consuming errors by not using them in the appropriate way.

Note that this hierarchy of modules—simple modules forming compound modules, forming in turn other compound modules–provides a very efficient way to build large, complex queueing networks. Start by building the parts, and then connect these parts to larger compound modules up to entire networks.

4.3 The `.cc` and `.h` files

Now we have defined all entities that take part in the process cycle of job-messages, and specified the routing between these entities in the compound modules, we should tell how each entity should process job-messages. In concrete terms, the generator should generate jobs with certain interarrival times distributed according to some specified probability distribution. These jobs are represented by a message, so that in fact the generator generates messages. Then it sends these messages out of its output gate. If you will have a look at `gen.cc` you will see that it will produce in total `fifonet.gen.num_messages` messages, each separated some time `ia_time` apart. These two parameters are defined in the file `omnetpp.ini`.

`fifo.cc` contains the heart of the simulation. Once you understand this code, you have mastered the most important aspects of OMNeT++. It may be somewhat hard, but keep on, other people could understand it too.

At the fifo queue jobs arrive, and depending on whether the server is idle or not, have to spend time in the queue. Lets first consider the function `Fifo::handleMessage` and see what it does. Suppose the server is busy. Then the event stack of the simulator should contain an event that indicates when the job's service is supposed to be finished. If you think a bit about how to implement such events, taking into account that all events are related to messages, you will understand that the server should send itself an `endService` message at the moment a job's service starts, to indicate when this job's service should stop. This `endService` message will be put on the event stack by the event scheduler (not to be confused with the queueing scheduler!). At some point in time, the event scheduler will remove this message from the stack, and will give it to the fifo module. Once this module receives the `endService` message, it knows that the job's service has ended, and that it should send it to the sink. When the queue is empty after the departure, the server should just wait. On the other hand, if there are jobs in queue, it should take the job in front of the queue, and start another service period. Now have a look at `fifo.cc` and think deeply about it until you understand what is exactly going on. Remember well that a simulation entity has no other means to communicate with itself in the future than via self-messages, and that these messages have to be put on the internal event stack.

`fifo.cc` contains four more functions. The functionality of `Fifo::initialize` should be clear from its implementation. The function `Fifo::finish` processes part of the simulation results when the simulation has finished. The other two functions `Fifo::serviceRequirement` and `endService` enable you to specify what to do when a job starts service—here only a random service duration is generated—and when it leaves. This provides interesting flexibility with respect to queueing simulation. For instance, in multi-class networks, jobs may change class from service station to service station, or a fraction may be sent back for another processing step at the server, etc. etc. Do not be tempted too easily to handle this kind of functionality in the `handleMessage` function. It can become quite complex.

The last module of interest, the sink, should be a breeze once you have mastered the above. It is a good idea to think about why we compute the waiting time of a job is this module, and why this is not possible in `fifo.cc`. This is of course by no means a generic property of OMNeT++, but just a consequence of the current implementation of the queueing entity.

5 Interesting extensions

There are a couple of interesting extensions to make to this fifo example. Let me number up a few.

- More interarrival and service distributions, such as an efficient implementation of the Coxian distribution, multi-server stations, etc.
- Other scheduling disciplines, such as Processor Sharing, Last Come First Serve pre-emptive resume, etc.
- More methods to statistically analyze simulation results

- Efficient implementations to simulate rare events.

In case you decide to build one of these extensions, please tell me so that I can take it up in the distribution.

6 Why use OMNeT++ for queueing system simulation?

Up to a few month ago, I worked for a large telecommunication company. (I only recently started to work for a university. Hence my time to write this document, and do some good to the global community) In one of the projects, we tried to reduce the convergence time of a distributed network restoration protocol. The interaction between the protocol, the involved timers, and the states of the queues in the equipment became soon completely intractable, analytically speaking. Hence we decided to analyze the behavior of the network and equipment with a simulator. As the developers wrote the protocol in C++ we wanted to use this code, as it would be implemented in real equipment, and carry it over with minor modifications to the simulation environment. Besides this we needed a graphical environment to see whether the protocol messages traveled the way we wanted, and ended up in the right queues. OMNeT++ provided this, and more.

The general experiences I gained during this and other projects with respect to simulation complex systems were briefly as follows:

- Using production code in a simulator saves a major amount of time compared to having to redesign the work in some kind of 'high-level', seemingly user friendly simulation environment. Besides this, the simulation itself helps to test the real code, that is, the code that will actually implemented in commercial equipment.
- Having insight in the code of the simulation environment is helpful to understand details of the simulation. Furthermore, it can be necessary to extend parts of the tool's functionality. Being dependent on companies to give you support, but keeping you securely away from the details of the implementation of their simulator, is not always what you want. Worse, you have to wait for their answer, which not always relates to your question . . . :-(.

Based on the above experiences as well as on some past work with commercial simulators, I want to point out some advantages, at least in my opinion, of running queueing simulations within OMNeT++.

- The behavior of systems is programmed in a 'normal' programming language, in this case C++. This provides the user way more flexibility than a 'high-end', tool-specific programming environment. For instance, if you use Bones, your programming skills gained during working with Bones only apply to the Bones environment. Besides this, these high-end languages appear, as long as you stick to the provided tutorials, to be quite generic. But when you try your hands on your own examples, you will feel how restrictive these tool-specific languages sometimes are.
- As a consequence of the use of C++, OMNeT++ is flexible and extensible. As an example, you can easily implement your own job interarrival and service distributions, statistical tests on the simulation results, more general networks, scheduling distributions, etc.

- It is under GPL license.
- OMNeT++ enforces 'separation of concerns'. You have to specify the separate functional entities, such as queues, schedulers, job generators, separately from routing functionality, that is the definition of the network topology.

Let me say one more thing about the first bullet. Of course, these commercial tools allow you to specify parts of the system in your own code, and connect this code with the simulation platform by means of software hooks. But then you have to figure out how this works plus all debugging, and you still have to program in a real programming language.

With respect to the last item, I hesitated to use the possibly vague words 'object-oriented', but that is what it really is. In my experience this is a bonus, as it makes the simulation environment modular. You can easily change only one part, or a module. As long as the interfaces between the modules remain the same, everything will remain working. Changing the topology is easy too. Take for instance this demo. If you want to build Jackson networks—a bunch of M/M/1 queues connected in a network such that jobs can arrive and leave the network—you only have to figure out how to set up such networks. the functionality of job sources, sinks and service stations is already there, ready to use. If you want to change some of the service rates, you can easily do this in the `omnetpp.ini` file.

Let me stop here, and let you convince yourself about the uses of OMNeT++.

References

[Knu97] D.E. Knuth. *The art of computer programming*, volume 2, Seminumerical algorithms. 3 edition, 1997.

[Ros93] S.M. Ross. *Introduction to Probability Models*. Academic Press, 5th edition, 1993.