SYSC5807 Term Project Report
DEVS Library for
Layered Queueing Networks

by Dorin Petriu (217774)

Overview

This report covers the implementation of a DEVS library for the simulation of Layered Queueing Networks. Section 1. provides an overview of Layered Queueing Networks. Section 2. describes the design, implementation and testing of the DEVS library. Finally, Section 3. concludes the report.

Appendix A also provides a guide to installing a version of the Cygwin environment that is compatible with the DEVS source code distribution.

1. Layered Queueing Networks

1.1. Background

Queueing Networks are based on a customer-server paradigm. Customers make service requests of the servers and these request are queued at the server until they can be serviced. Traditional Queueing Networks model only a single layer of customer-server relationships. Layered Queueing Networks (LQN) allow for of an arbitrary number of client-server levels [5][6]. LQNs can thus model intermediate software servers and be used to detect software deadlocks and software as well as hardware performance bottlenecks [4]. The layered aspect of LQNs makes them very suitable for evaluating the performance of distributed systems [7][8].

1.2. Notation

LQNs model both software and hardware resources. The basic software resource is a task. A task is any software object that has its own thread of execution. Tasks have entries which act as service access points. The basic hardware resource is a device. Typical devices are CPUs and

disks. Figure 1 shows the visual notation for tasks and devices.

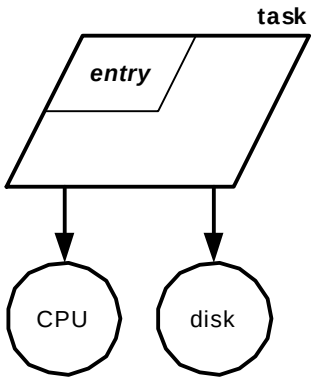

Figure 1: LQN task, entry, CPU and disk devices.

Service calls are shown by messaging arrows. The LQN notation supports three types of calls: asynchronous, synchronous, and forwarded calls. An asynchronous call does not involve any kind of blocking on the part of the sending task. A synchronous call means that the sending client task blocks until it receives a reply. In a forwarding call, the sending client task makes a synchronous call and blocks waiting for a reply, the receiving intermediate server task partially processes the call and then forwards it to another server which becomes responsible for sending a reply to the blocked client task. The intermediate server task can continue operation after forwarding the call and there can be any number of forwarding levels. Calls are made from a task's entries and they can also be made in sequence. Figure 2 shows the LQN notation for these types of calls. Figure 3 shows the time semantics of these different types of calls.

Tasks receive service requests at designated interface points called entries. Entries correspond to service access point for a task. There is a different entry for every kind of service that a task provides. An entry may be defined atomically, with its own hardware service demands and calls to other tasks. Alternately, an entry may be defined by blocks of smaller computational blocks called activities. Atomic entries can also have phases that divide the workload into a first phase that is executed prior to sending a reply and a second phase that is executed after sending a reply.

An entry receiving a synchronous service call is responsible for sending a reply after the request has been completed. Reply are implicit at the end of the first phase for entries that are defined atomically but must be explicitly specified for entries defined by activities. An entry

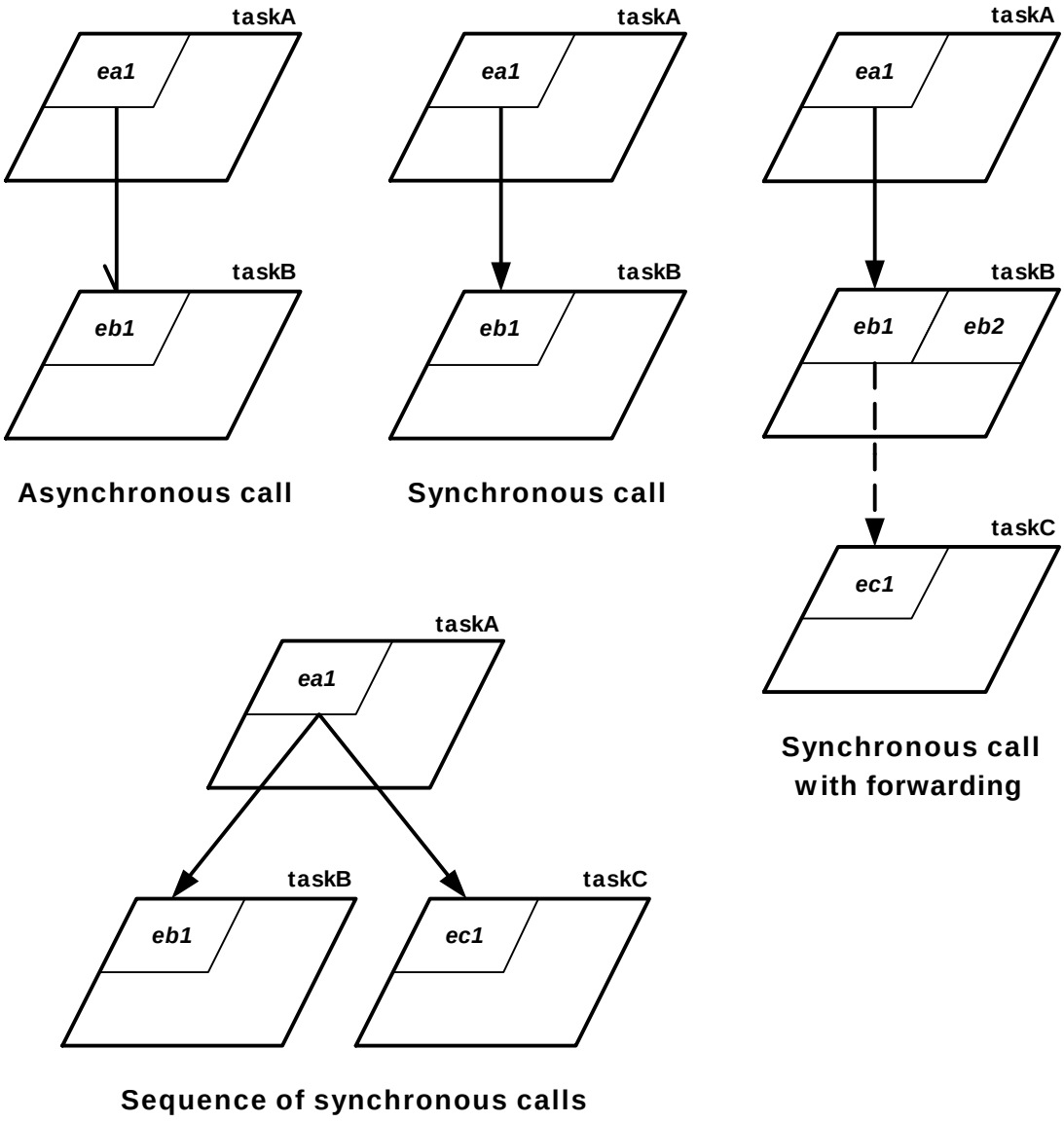

Figure 2: LQN messaging.

receiving a synchronous service request may also forward it to entries in other tasks which then become responsible for sending the reply to the original caller. In the case of a forwarded call, the original calling task remains blocked until it finally receives the reply at the end of the forwarding chain.

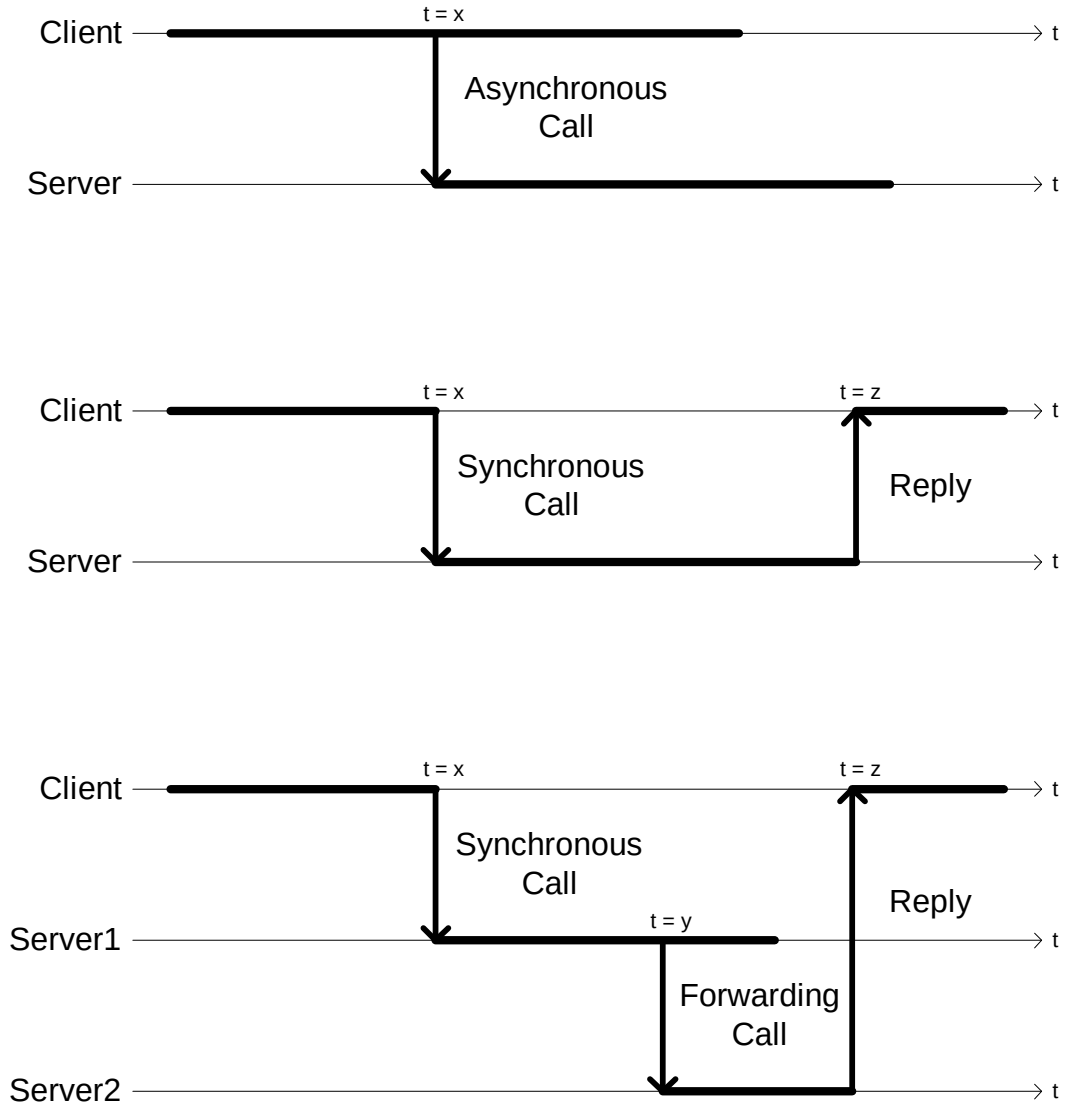

Figure 3: Time semantics of LQN asynchronous, synchronous, and forwarded calls.

1.3. Solving

LQNs can be solved either using either the Layered Queueing Network Solver (LQNS) or the Layered Queueing Simulator (LQSim).

LQNS is an analytic solver developed at Carleton University by Greg Franks as part of his Ph.D. research [1]. LQNS breaks the LQN layers down into separate queueing network sub-mod-

els. The individual queueing networks can be solved analytically using mean value analysis (MVA). The MVA results for each sub-model are then used to the parameters for the other sub-models it is connected to and the MVA is performed anew. This process is repeated either for a maximum number of iterations or until the results converge on a convergence value specified by the user.

LQSim uses the ParaSol simulation environment. ParaSol can simulate multithreaded systems that support transactions and provides built-in statistics for monitoring simulation objects [2][3]. LQNs are simulated by creating tokens for each call and following those tokens through the system. The performance metrics are arrived at by recording the wait times and other statistics for each token. LQSim requires large number of runs in order to gather statistically meaningful data. This makes LQSim appreciably slower than LQNS and requires long simulations in order to generate accurate results.

Both LQNS and LQSim generate results that show entry average service times, average waiting time, throughput, and utilization, as well as processor throughput and utilization.

2. LQN Simulation Library for DEVS

2.1. Requirements

The LQN simulation library must support the LQN notation as well as provide as much of the functionality of the LQSim simulator.

From a notation standpoint, the minimum the DEVS LQN library must represent are:

- tasks
- entries
- phases
- processors

Additionally, the library might also represent:

- disks
- activities

From a simulation standpoint, the DEVS LQN library should provide:

- entry average service time, throughput, utilization
- phase average service time
- processor throughput, utilization
- queue average waiting time, average queue length (this is not provided in the existing LQN solver and simulator and although it can be calculated, it would be desirable to provide it outright)

2.2. Design

The main design issues for the DEVS LQN simulation library involved deciding which LQN elements to include and which elements or artifacts to model as DEVS atomic models and which ones to model as DEVS coupled models.

It was decided that processors, tasks, entries and phases needed to be included in the library. Activities were left out of the initial version of the library since they are optional in the LQN notation and including them right away raised too many integration issues with respect to entries and phases.

LQN elements have queues that are implicitly supported by LQNS and LQSim. Unfortunately no implicit support for queueing was found in the DEVS environment so queues had to be incorporated explicitly into the LQN library. Since queues behave the same way for software or hardware elements, a decision was made to implement the queue as a separate atomic model.

LQN calls are made using entry names to identify the call target. Unfortunately no port addressing mechanism was found for DEVS so a mechanism was needed to deal with the directing the calls. The solution was to implement a DEVS version of a multiplexer and demultiplexer to either aggregate calls into a given queue (either for an entry or a processor) or to distribute calls from an entry to other entries.

The resulting model design for the LQN library is given in Table 1 below.

LQN aspect or element	DEVS atomic model	DEVS coupled model	Functionality
Processor	Processor		• receives call • executes it for the specified amount of time • replies when done • calculates utilization and throughput
		Processor	• combines multiplexer, queue, and atomic processor for full LQN processor functionality
Entry	Entry		• receives call • executes the associated workloads (phase 1 and phase 2 processing, server calls) • replies when done • phase 1 and phase 2 processor demands must be initialized through the initproc port prior to executing the entry • phase 1 and phase 2 server calls must be initialized through the initserv port prior to executing the entry
		Entry	• combines multiplexer, queue, atomic entry, and demultiplexer for full LQN entry functionality
implied queue	Queue		• adds call to queue • sends first element in queue to attached processor or entry if it becomes idle • sends reply back up to the call source
aggregating calls from multiple sources	Multiplexer		• aggregates calls from multiple input ports and sends them out the single output port • adds a message with the input port index • sends reply from the reply port at the “output end” to the appropriate response port at the “input end”
distributing calls to different entries	Demultiplexer		• distributes calls from the single input port and sends them out the appropriate output port • sends reply from the reply port at the “output end” to the single response port at the “input end”
task		Task	• composed of multiple entries

Table 1: DEVS models for the LQN simulation library.

LQN aspect or element	DEVS atomic model	DEVS coupled model	Functionality
disk		Processor	• approximates disk using the same functionality as a processor
activity	not represented	not represented	

Table 1: DEVS models for the LQN simulation library.

2.2.1. Messaging

LQN messages can be thought of as having a *source* field denoting the entity making the call, a *destination* or *target* field denoting the entry for which the call is destined, and a *demand* field denoting the workload associated with the call, although any of these fields can be optional. DEVS messaging only provides one variable field per message, therefore it was necessary to sometimes send and receive multiple messages in order to have the same functionality as LQN messages. Table 2 shows the messages sent between atomic models in the DEVS LQN simulation library, how they are ordered and how they should be interpreted.

Sender (port)	Receiver (port)	LQN Equivalent Message	DEVS Messages (in order)	Interpretation
Processor (reply)	Queue (response)	done	• reply	• notify the source entry that the processing is done, the message value represents the actual processing time in ms
Processor (ready)	Queue (ready)	done	• ready	• ready for another job, the message value is irrelevant
Processor (throughput)		throughput	• throughput	• the message value represents the processor throughput in number of jobs per ms
Processor (utilization)		utilization	• utilization	• the message value represents the fraction/percentage of time that the processor has been busy

Table 2: DEVS LQN simulation library messages.

Sender (port)	Receiver (port)	LQN Equivalent Message	DEVS Messages (in order)	Interpretation
Entry (proccall)	Multiplexer (in[0...9])	processor call	• processor service demand	• the message value represents the processor demand in ms
Entry (servcall)	Demultiplexer (in)	service call	• service call	• the message value represents the index of the target server
Entry (avservtime)		average entry service time	• average entry service time	• the message value represents the average entry service time in ms
Entry (avph1time)		average phase1 service time	• average phase 1 service time	• the message value represents the average phase 1 service time in ms
Entry (avph2time)		average phase2 service time	• average phase 2 service time	• the message value represents the average phase 2 service time in ms
Entry (throughput)		throughput	• throughput	• the message value represents the entry throughput in number of jobs per ms
Entry (utilization)		utilization	• utilization	• the message value represents the fraction/percentage of time that the entry has been busy
Queue (out)	Processor (in)	processor call	• processor service demand	• the message value represents the service demand in ms
Queue (out)	Entry (in)	service call	• service call	• service call, the message value is irrelevant
Queue (reply)	Multiplexer (response)	reply	• reply	• the message value represents the index of the source that must be replied to

Table 2: DEVS LQN simulation library messages.

Sender (port)	Receiver (port)	LQN Equivalent Message	DEVS Messages (in order)	Interpretation
Queue (averagesize)			• average queue size	• the message value represents the average number of elements in the queue at the time the message was sent
Queue (averagewait)			• average queueing wait	• the message value represents the average number of milliseconds a message spent in the queue at the time the message was sent
Multiplexer (out)	Queue (in)	service call	• source of service call • service call demand	• the message value represents the index of the call source • if attached to a processor then the message value represents the processor service demand in ms, otherwise the message is irrelevant
Multiplexer (reply[0...9])	Demultiplexer (resp[0...9])	reply	• reply	• reply, the message value is irrelevant
Demultiplexer (out[0...9])	Multiplexer (in[0...9])	service call	• service call	• service call, if attached to a processor then the message value represents the processor service demand in ms, otherwise the message value is irrelevant
Demultiplexer (reply)	Entry (response)	reply	• reply	• reply, the message value represents the index of the call target returning the reply

Table 2: DEVS LQN simulation library messages.

Sender (port)	Receiver (port)	LQN Equivalent Message	DEVS Messages (in order)	Interpretation
	Entry (initproc)		• phase number • processor demand	• the message value represents the phase number to initialize • the message value represents the processor demand in ms
	Entry (initserv)		• phase number • calls • call target	• the message value represents the phase number to initialize • the message value represents the number of calls to make to the target server • the message value represents the index of the target server

Table 2: DEVS LQN simulation library messages.

2.2.2. Messaging Alternative Design

An alternative approach to allow DEVS messages with three variable fields was attempted at an earlier stage of the project. The *event.h, message.h, model.h, process.h,* and *root.h* DEVS files were modified to add the additional message fields. The *lqn*.h* and *lqn*.cpp* files made use of the additional messaging fields.

It was possible to compile all these files individually without a problem, but this approach broke down when it came to linking. Unfortunately the DEVS distribution does not include all the code necessary to recompile every all the object files and the *libsimu.a* library and some of those legacy objects were found not to be compatible with the expanded message format. The files for this alternative approach can be found in the *devslqn-altmsg.zip* file.

2.3. DEVS Atomic Model Behaviour

Figure 4 through Figure 8 show the FSM for the DEVS atomic models listed in Table 1.

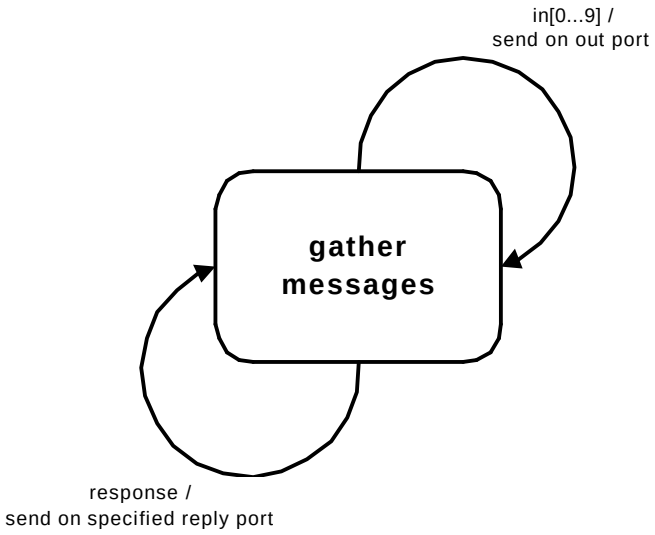

Figure 4: FSM for the message multiplexer atomic model.

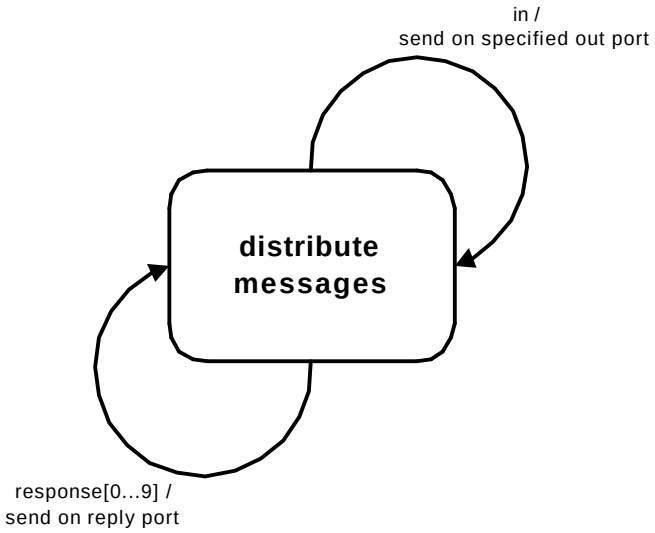

Figure 5: FSM for the message demultiplexer atomic model.

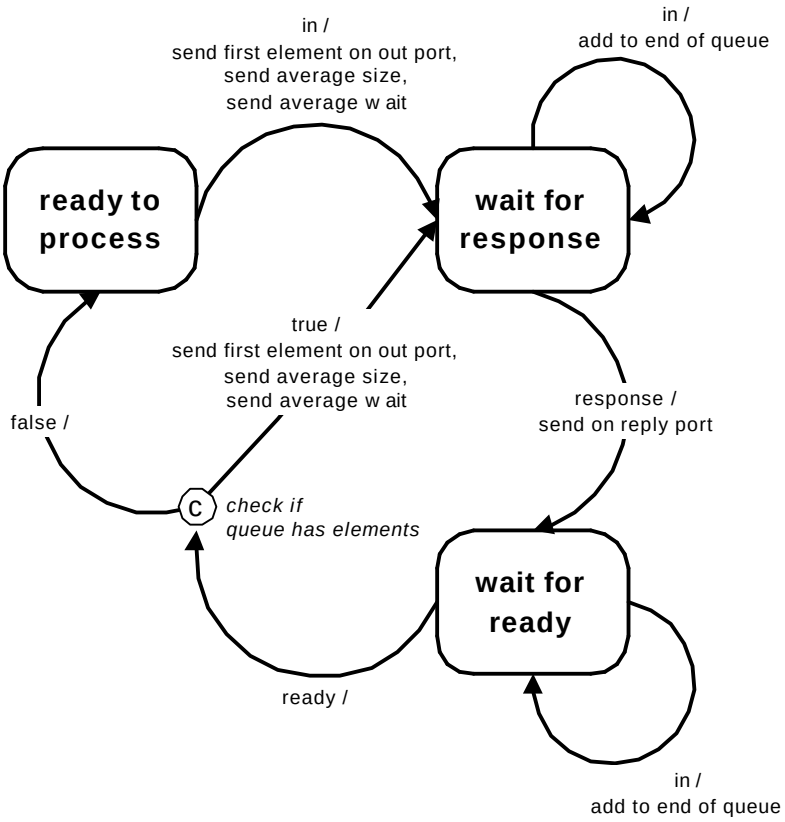

Figure 6: FSM for the queue atomic model.

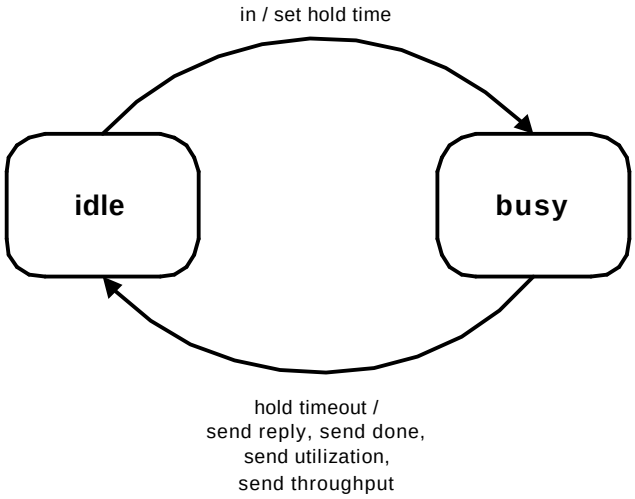

Figure 7: FSM for the processor atomic model.

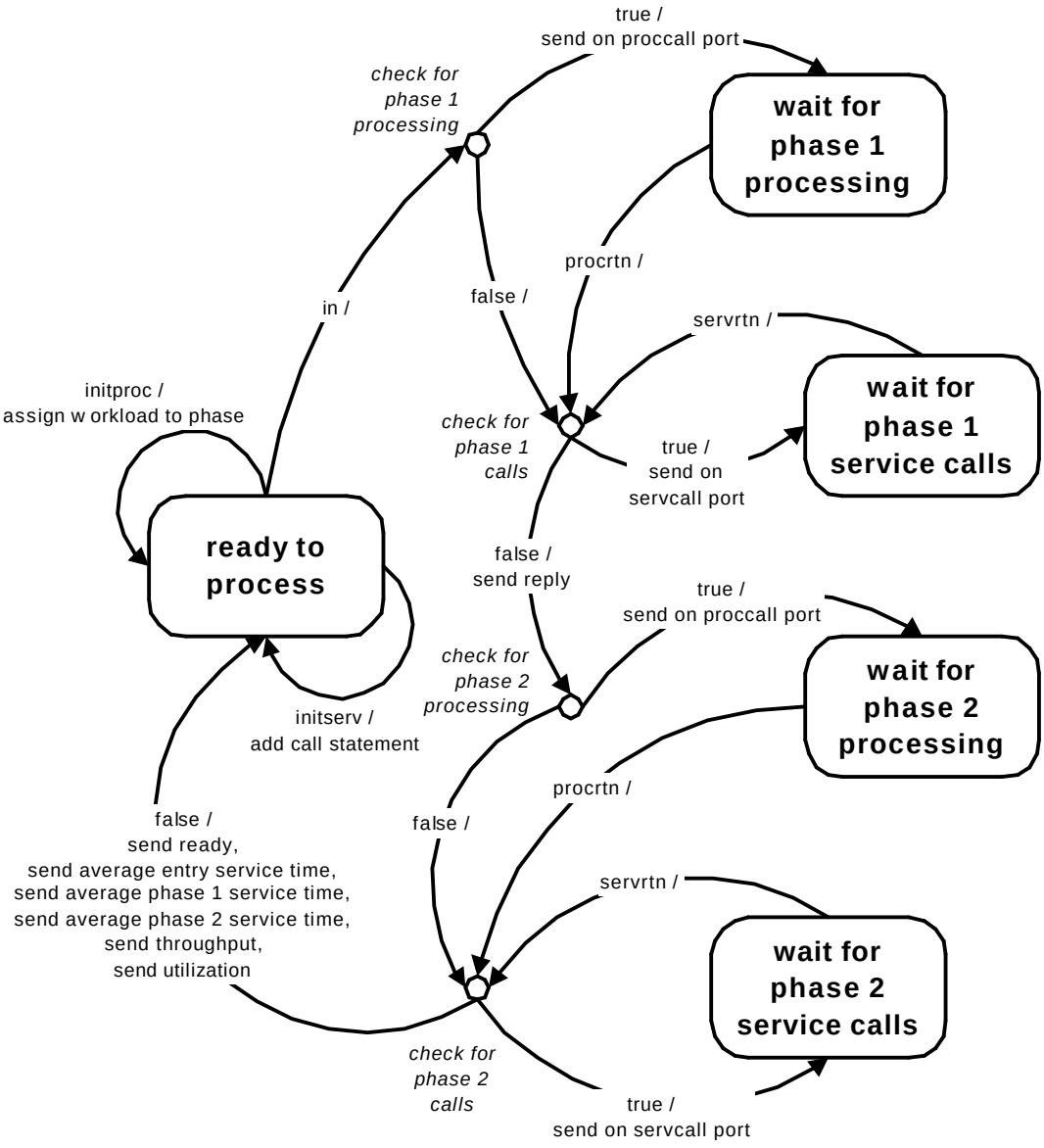

Figure 8: FSM for the entry atomic model.

2.4. DEVS Atomic Model Structure

Figure 9 through Figure 13 show the structure of the DEVS atomic models listed in Table 1 using a ROOM actor notation.

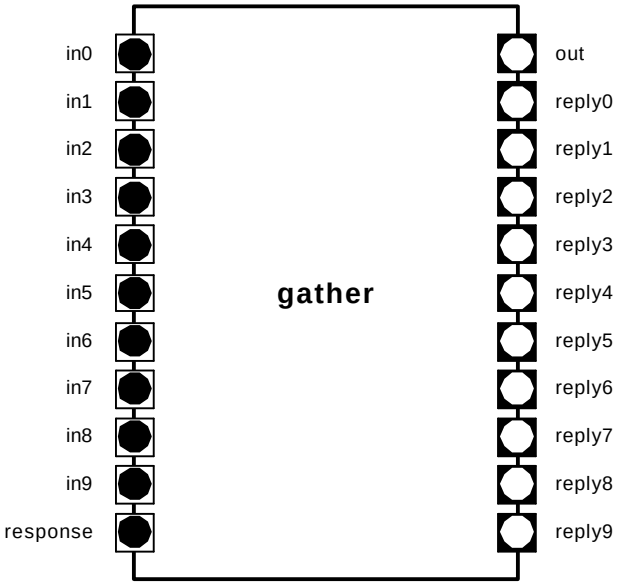

Figure 9: ROOM structure for the message multiplexer atomic model.

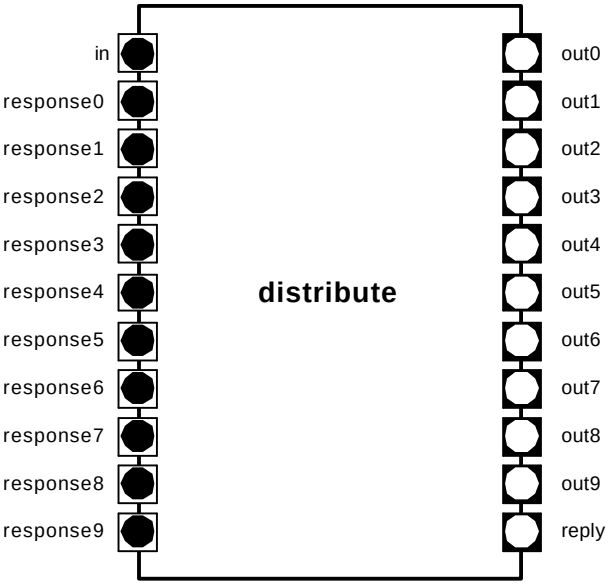

Figure 10: ROOM structure for the message demultiplexer atomic model.

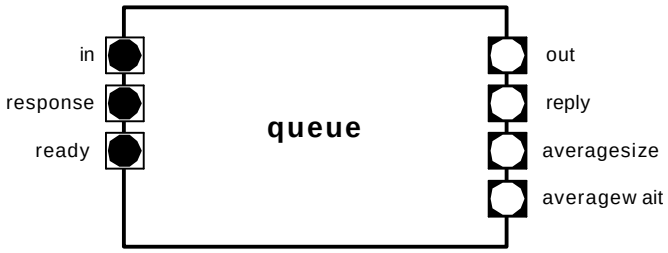

Figure 11: ROOM structure for the queue atomic model.

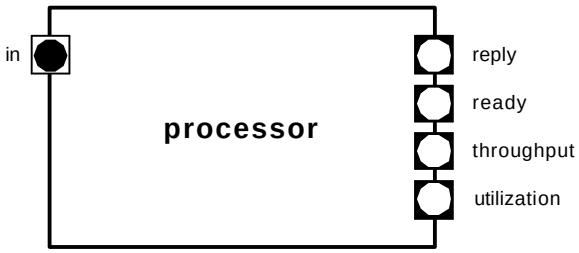

Figure 12: ROOM structure for the processor atomic model.

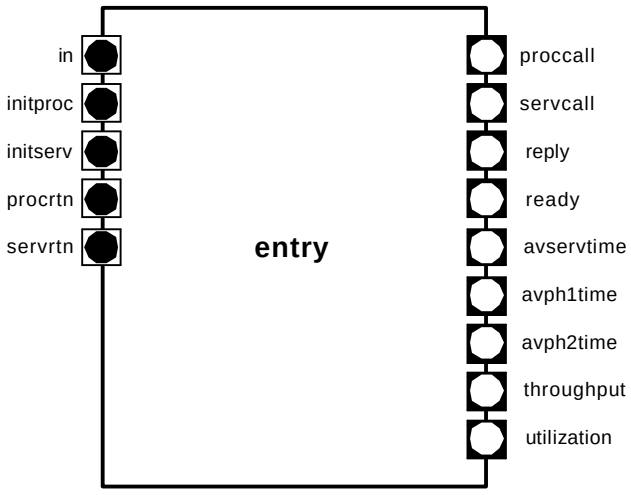

Figure 13: ROOM structure for the entry atomic model.

2.5. DEVS Atomic Model Structure

Figure 14 and Figure 15 show the structure, using ROOM actor notation, of the DEVS coupled models for LQN processors and entries incorporating queues and message routing multiplexers/demultiplexers. It is these coupled models that fully represent LQN processors and entries.

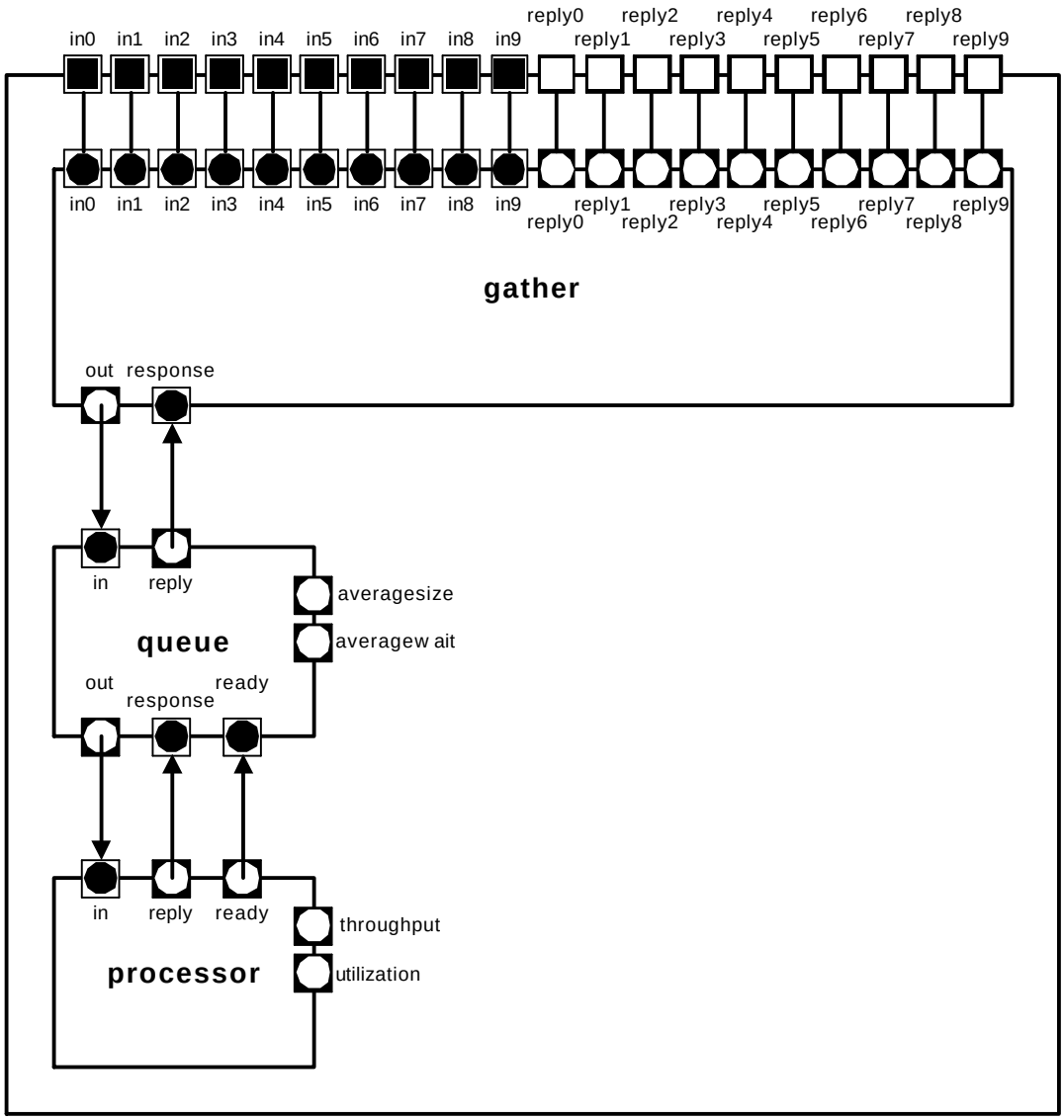

Figure 14: ROOM structure for the LQN processor coupled model.

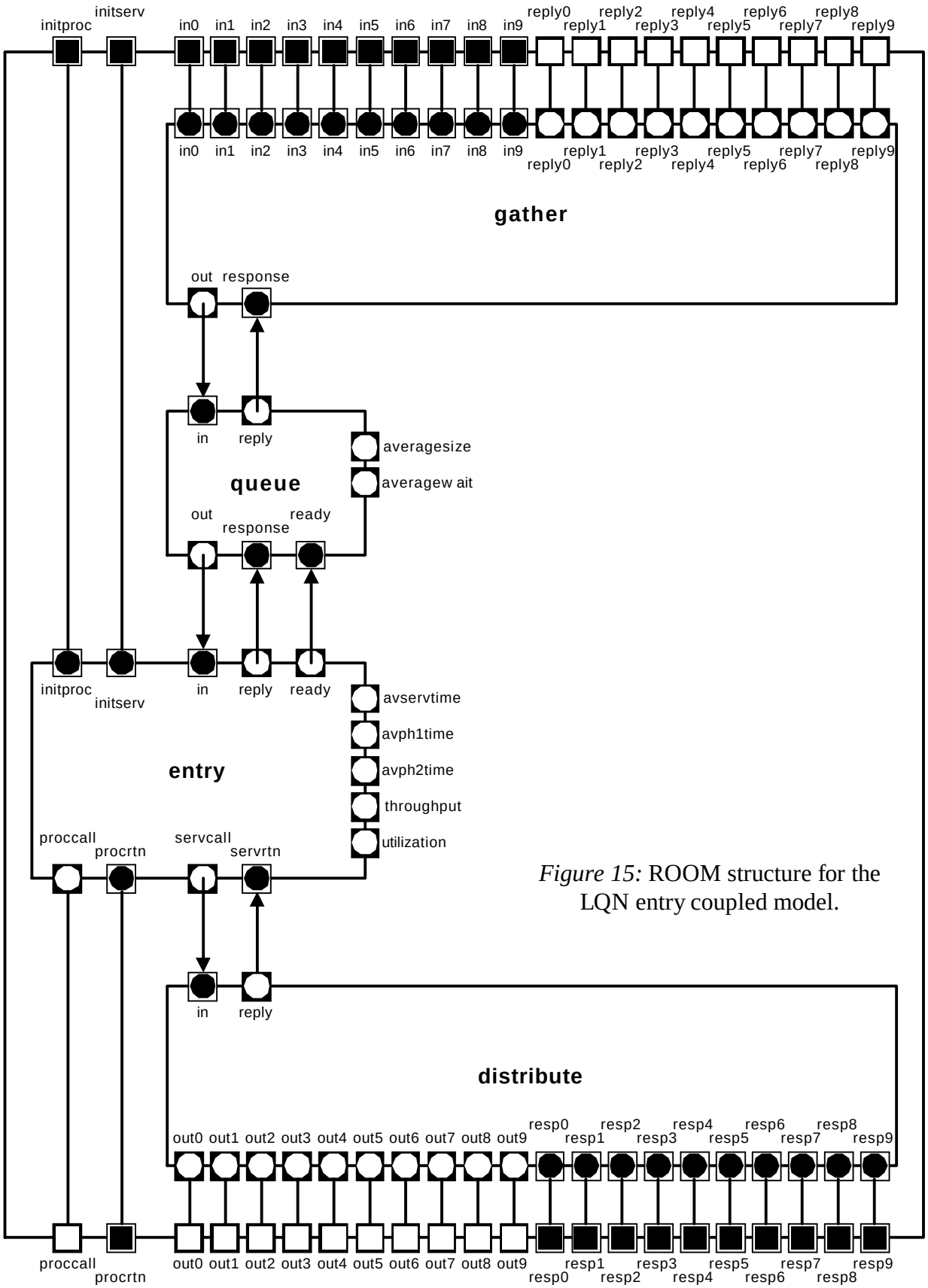

Figure 15: ROOM structure for the LQN entry coupled model.

2.6. Implementation

Table 3 lists the new files that were created for the implementation of the DEVS LQN simulation library.

File	Contents
lqndefs.h	• common definitions for all the LQN files • defines constants and types used by the other files
lqndistribute.h	• header file for the message demultiplexer atomic model • declaration of class LqnDistribute
lqndistribute.cpp	• implementation file for the message demultiplexer atomic model • implementation of class LqnDistribute
lqnentry.h	• header file for the entry atomic model • declaration of class LqnEntry
lqnentry.cpp	• implementation file for the entry atomic model • implementation of class LqnEntry
lqngather.h	• header file for the message multiplexer atomic model • declaration of class LqnGather
lqngather.cpp	• implementation file for the message multiplexer atomic model • implementation of class LqnGather
lqnmath.h	• header file for the mathematical methods used by the LqnEntry and LqnProcessor classes • based on randlib.h with the addition of the roundlqn method to round a double value to the nearest whole number • required for linking with the C++ compiled LQN object files as the linker does not recognize the C compiled methods from randlib.o
lqnmath.cpp	• implementation file for the mathematical methods used by the LqnEntry and LqnProcessor classes • based on randlib.c with the addition of the *roundlqn* method to round a double value to the nearest whole number • removed sources of warnings when using the C++ compiler rather than the C compiler used for randlib.c • required for linking with the C++ compiled LQN object files as the linker does not recognize the C compiled methods from randlib.o
lqnprocessor.h	• header file for the processor atomic model • declaration of class LqnProcessor
lqnprocessor.cpp	• implementation file for the processor atomic model • implementation of class LqnProcessor
lqnqueue.h	• header file for the queue atomic model • declaration of class LqnQueue

File	Contents
lqnqueue.cpp	• implementation file for the queue atomic model • implementation of class LqnQueue

Table 3: New files created for the DEVS LQN simulation library.

The file *time.h* was also modified with the addition of:

- a *Time* constructor that uses a double milliseconds argument
- a *fromMsec* method that creates a Time value from a double milliseconds argument
- an overloaded = operator that assigns a Time value from a float milliseconds value
- an overloaded = operator that assigns a Time value from a double milliseconds value

Additionally, *register.cpp* was modified to register atomic models for the LqnDistribute, LqnEntry, LqnGather, LqnProcessor and LqnQueue classes.

2.7. Test Cases

The DEVS LQN library was tested using the tests listed in Table 4 below.

Test	Purpose	Observations	Files
lqndistribute	unit test of the LqnDistribute atomic model	• model performed as expected • in messages distributed to the correct out ports • error messages generated when using index outside the allowed 0 to 9 range • all replies received and sent up	• lqndistribute.bat • lqndistribute.ev • lqndistribute.ma • lqndistribute.log
lqnentry	unit test of the LqnEntry atomic model	• model performed as expected • correct number of calls generated • inputs when busy ignored • performance results make sense	• lqnentry.bat • lqnentry.ev • lqnentry.ma • lqnentry.log

Table 4: DEVS LQN library tests.

Test	Purpose	Observations	Files
lqngather	unit test of the LqnGather atomic model	• model performed as expected • in messages gathered and sent to the out port • response received and sent to the correct reply ports • error messages generated when using index outside the allowed 0 to 9 range	• lqngather.bat • lqngather.ev • lqngather.ma • lqngather.log
lqnprocessor	unit test of the LqnProcessor atomic model	• model performed as expected • processor was busy for the correct amount of time • arrivals when busy were ignored • throughput and utilization results make sense	• lqnprocessor.bat • lqnprocessor.ev • lqnprocessor.ma • lqnprocessor.
lqnqueue	unit test of the LqnQueue atomic model	• model performed as expected • elements queued and dequeued properly • size and wait results make sense	• lqnqueue.bat • lqnqueue.ev • lqnqueue.ma • lqnqueue.log
lqn1	test of processor coupled model	• model performed as expected • handled extremely small and extremely large numbers • performance results make sense	• lqn1.bat • lqn1.ev • lqn1.ma • lqn1.log
lqn2	test of atomic LqnEntry running on processor coupled model	• model performed as expected • performance results make sense	• lqn2.bat • lqn2.ev • lqn2.ma • lqn2.log
lqn3	test of entry coupled model and processor couple model	• model performed as expected • performance results make sense	• lqn3.bat • lqn3.ev • lqn3.ma • lqn3.log
lqn4	test of two layer model with two entries, top layer entry makes 100 calls to the lower layer entry	• model performed as expected • performance results make sense	• lqn4.bat • lqn4.ev • lqn4.ma • lqn4.log

Table 4: DEVS LQN library tests.

Test	Purpose	Observations	Files
lqn5	test of three layer model with three entries, top layer entry makes 10 calls to middle layer entry which makes 10 calls to the lower level entry	• model performed as expected • performance results make sense	• lqn5.bat • lqn5.ev • lqn5.ma • lqn5.log
lqn6	test of two layer model with three entries, top layer entry makes a call to the first lower layer entry and then to the other lower layer entry	• model performed as expected • performance results make sense	• lqn6.bat • lqn6.ev • lqn6.ma • lqn6.log

Table 4: DEVS LQN library tests.

In order to test that the equations used to calculate the average performance metrics are correct, a copy of the library was put in the *norandom* directory and was modified not to use random numbers for processor or service calls. Inspecting the log files resulting from running the tests in Table 4 showed that the service time averages remained constant through the length of a run, thereby showing that the calculations are indeed correct.

2.8. Observations

The following observations were made while working on this project:

- the DEVS distribution does not contain enough source code to customize the environment (there the attempt to add more fields to DEVS messages ultimately did not link and thus did not work)
- there are errors and inconsistencies in the DEVS parser for .ma files, all comments had to be removed from the .ma files since their presence caused simulation errors in some instances
    - there did not seem to be any kind of recognizable pattern of when or why these errors occurred
- there is an error in the *fmod* function supplied with the Cygwin include libraries, in some cases fmod 1 returns a remainder of 1 instead of when applied to an integer number
    - this can be tested by running the *fmodtest(.exe)* program that was created for this project
    - this error was compensated for in the *roundlqn* method implemented in *lqnmath.cpp*

- the random number generating routines in *randlib.c,* and by extension in *lqnmath.cpp,* do not truly generate random numbers since the random numbers generate are the same during every run of a particular test
- furthermore, the *genexp* and *genexplqn* methods generate exponentially distributed numbers with an ever increasing mean, this was observable in the performance metrics over long runs
- the *makeFrom(float milliseconds)* method in *time.h* actually generates time values in seconds not milliseconds
    - the *fromMsec(double msec)* method was added to remedy this failing
- there is a danger in overwriting or losing incoming messages during a DEVS simulation when using active state hold times of zero
    - this can create a conflict in the LQN model when multiple messages arrive at the same time and one of them is overwritten

3. Conclusions

The DEVS LQN simulation library provides a reasonable starting point for simulating simple LQN performance models in the DEVS environment. The weaknesses in the random number generation, as well as the inability to deal at a smaller level of time granularity than milliseconds make this library unsuitable for extensive performance modeling.

Additional work should also be undertaken to add support for asynchronous and forwarding calls to the library - the current version uses only synchronous calls. As well, it is necessary to protect the library classes from the potential for message loss if multiple messages arrive at the same time. Eventually the LQN library should also include support for activities, although this will probably necessitate a redesign of the whole library.

Finally, this project proved to be an extremely interesting capstone to the advance simulation course, although it may have been too ambitious for a single person.

References

[1] Greg Franks, “Performance Analysis of Distributed Server Systems”, Report OCIEE-00-01, Ph.D. thesis, Carleton University, Ottawa, Jan. 2000

[2] E. Mascarenhas, " A System for Multithreaded Parallel Simulation and Computation with Migrant Threads and Objects", Ph.D. Thesis, Department of Computer Sciences, Purdue University, West Lafayette, USA, 1996

[3] E. Mascarenhas, F. Knop, and V. Rego, "ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads", Winter Simulation Conference, 1995

[4] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendez-vous Networks", IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, September 1995

[5] J. A. Rolia, K. C. Sevcik, "The Method of Layers", IEEE Transactions on Software Engineering, Vol. 21, No. 8, 1995, pp. 682-688

[6] C. M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks", Performance Evaluation, Vol. 9, No. 2, Apr 1988, pp. 143-160

[7] C. M. Woodside, J. E. Neilson, D. C. Petriu and S. Majumdar, " The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", IEEE Transactions on Computers, Vol. 44, No. 1, Jan 1995, pp. 20-34

[8] C. M. Woodside, S. Majumdar, J. E. Neilson, D. C. Petriu, J. A. Rolia, A. Hubbard and R. B. Franks, " A Guide to Performance Modeling of Distributed Client-Server Software Systems with Layered Queueing Networks", Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Nov 1995

APPENDIX A - Cygwin Installation for DEVS

This appendix describes the steps required to install the Cygwin environment in such a way as to be able to compile the DEVS package.

The current distribution version of Cygwin comes with the gcc 3.2 compiler and libraries. This version of gcc is not compatible with the DEVS code and/or makefile which is meant to work with gcc 2.95. Although the Cygwin distribution can be donwloaded with the gcc 2.95 compiler (accessed as gcc2 or g++2) and also includes the gcc 2.95 libraries, but apparently not a gcc 2.95 linker and loader, the installation scripts do not set gcc 2.95 up in such a way that it can be used as the default compilation environment out of the box.

One solution would be to modify the DEVS makefile to make use of the gcc 2.95 compiler and libraries that come with the Cygwin distribution. Another possibility would be to migrate the DEVS code to gcc 3.2. Unfortunately both of these solutions require more in-depth compilation and configuration knowledge than the average student possesses.

The compilation solution used for this project makes use of both the current Cygwin distribution for its installation scripts and the Cygwin version available on the SCE network for its gcc 2.95 compiler, linker, loader and libraries. Please note that all of the following steps should be followed. Simply copying the Cygwin version from the SCE network does not work because it will not set up all the necessary paths and system variables in Windows. The installation scripts that come with the Cygwin distribution from the Cygwin site do set up those paths and variables.

The installation steps used were the following:

- download the Cygwin distribution from www.cygwin.com
  - select whatever options you wish, but make sure that gcc, bison, tar and post-installation scripts are selected
- install the Cygwin distribution
- tar and gzip the Cygwin environment from the SCE network (DO NOT use any Windows zip programs to do this, they do not preserve Cygwin/UNIX symbolic links)
  - tar -cvf cygwin.tar /*
  - gzip cygwin.tar

- copy the resulting cygwin.tar.gz file into the Cygwin root directory
- unzip and untar the cygwin.tar.gz file
    - gunzip cygwin.tar.gz
    - tar -xvf cygwin.tar

This process will overwrite some of the Cygwin files and directories installed by the Cygwin distribution installation program with their older counterparts from the SCE network. The resulting Cygwin hybrid installation should now work with the DEVS code. (N.B.: The cygwin1.dll file should be deleted from whatever directory DEVS is installed in before attempting to compile.)