EasySim: A Student Aimed Networked Algorithm Development Package

Alexander Heavens

Matriculation No. 080004721

Project Supervisor: Professor Stephen Linton

Abstract

Teaching students the finer points of networked algorithms is a difficult task. A lecturer may wish to provide a visual representation of these algorithms or even set coursework that requires the student to develop their own. However, there are few programs that offer a comprehensive means to both.

EasySim is a networked algorithm development package aimed specifically for students and lecturers. EasySim provides the simulation of user defined network algorithms, visualisation of algorithm patterns and a framework for testing the correctness of these algorithms.

Contents

Declaration

”I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

”The main text of this project report is 13,378 words long, including project specification and plan.

”In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

1 Introduction

The networked algorithm is a complex beast whose dynamic behaviour, sprouting from but a few simple seeds of code, soon amounts to a plethora of interaction that would easily baffle the casual observer. Yet, these core algorithms are the very basis for any modern means of communication, more so than any router, hub or server. Their role within the fabric of any network, although hidden to the masses that use them habitually, is not to concern themselves with the minutia of electrical signals that pass through wires. Neither do they care for the multitude of protocol stacks that dissect and wrap the data sent between points. No, the networked algorithm is concerned only with the dissemination of pure information so that engineers may take advantage of the sheer potential that an abstracted network structure can provide.

It is only through the research and development of these algorithms that the common programmer can effectively command more complex distributed systems, and provide scalability to match the ever increasing demands of contemporary computing. Yet she faces a mighty barrier when first tasked with understanding these algorithms. By their very nature, the complex behaviours exhibited by a networked algorithm can considerably cloud the learning of any student who should wish to develop them. More so, even the particularly gifted will find themselves hampered in their efforts when stung by the expense of time and capital that the creation and maintenance of a physical network demands. As a result, the lecturer must cope with only dry texts of theory, diagrams that provide only snapshots of algorithm execution, and psuedocode that cannot be easily understood on a distributed scale.

Yet there are few other choices for the student. An ideal solution to this problem would be the creation of a simulated testbed that allowed for development and understanding of networked algorithms from the perspective of the inexperienced programmer. By virtue of simulation, the testbed can provide excellent profiling and test information for a given algorithm, all at a fractional cost and vastly increased flexibility comparative to the equivalent physical system. Such a package, if simple enough for the user to manipulate and informative enough for that user to understand how an algorithm affects the network, would prove an invaluable tool for anyone facing such difficulties. It is to this end that this project is intended. Through the creation of a student accessible framework, users may define algorithms against a simple programming-based interface. Once defined, simulation of the user's algorithm will generate results that can be visualised and gleaned for profiling, robustness and correctness information. In addition, a lecturer of courses for which networked algorithms are a core concept may be aided by the created system for the purposes of demonstration of concept or the assignment of coursework.

2 Objectives

The following section describes the objectives that the development of the project as a whole is intended to meet. These objectives are subdivided into primary, secondary and tertiary categories, depending upon the importance of the objective.

2.1 Primary Objectives

The core of the project revolves around the accurate and timely simulation of user-code such that a student user can develop in abstract language a high-level networked algorithm, and test that algorithm to ensure its fitness for purpose. The objectives are of absolute importance to the success of the project as a whole.

1. The creation of a user-friendly application programming interface for the communication nodes within a network written. This API shall allow a student user to write a high-level abstract algorithm without concern for the means of passing messages within the network.

2. The simulation of user code under user-defined conditions. Working against the provided API, a user will be able to simulate their algorithm across a constructed network of their design and given adequate scope to change the parameters of simulation.

3. Analysis of user-code performance, robustness and correctness. The simulation of user-code will provide the user with a adequate profiling and testing of their algorithm to gauge these conditions.

2.2 Secondary Objectives

In addition to developing a means to simulating user-defined network algorithms, allowing the visual representation of the received data from simulation is desired. Although not of such importance as the primary objectives, the completion of secondary objectives is highly desired for this project.

1. The recording of user-code simulations for future playback. To allow the further inspection of how a user-defined algorithm has performed, the simulation of that algorithm must be stored in a re-usable format.
2. Informative visualisation of simulation data. To provide a more accessible view of algorithm performance, the project should provide the user with an easily understandable visualisation of the playback of events throughout its past simulation.
3. Annotation of recorded simulations. To aid in the demonstration of algorithm properties, the user must have some means to distinguish nodes that have reached appropriate states.
4. User-defined event scripting affecting simulation. To allow the user to interact with their algorithm in an unexpected manner, the project will allow the user to define events from outside the problem domain that will be triggered during simulation.
5. Structuring of software to provide unit testing framework for user code. To allow individual components of a user's code to be tested, the project will allow simulation to be performed in a re-testable manner.

2.3 Tertiary Objectives

Although additional to the project, the completion of tertiary objectives can provide an additional distinction to the project:

1. The ability for a user to view multiple event timelines of a simulation at once. This will aid a user in distinguishing how the behaviour of their algorithm works dependent on a particular turning point.

3 Context Survey

The following section provides an overview of similar commonly available system for the simulation of user code. The purpose of this section is to gain place the functionality of the project into a context of these competing system and to distinguish it from them.

3.1 ns

ns is a long standing series of network simulation packages that have seen a wide range of applications in both academic and industrial settings. The design of the ns simulators allows for the simulation of many network protocols, focussing on realism in simulation and completely configurable by the user. In addition to this accuracy, the appeal of the ns simulator also comes from its large collection of simulated protocols.

The original ns simulator was released in 1995, swiftly followed by its most popular incarnation *ns-2* in 1996. Funded by both DARPA and the National Science Foundation, ns-2 saw a long history of continued development. Although ns-2 has been used for educational purposes[15], its technical nature often requires large-scale modification and enhancements to the package. Factors such as a lack of graphical user interface and a requirement for user to have prior knowledge of simulation modelling and queuing theory has limited the accessibility of ns as an educational tool. The latest ns simulator, *ns-3*, has been developed since working from a fresh code base. This version has made an attempt to distinguish itself from ns-2 by allowing better support for educational uses though better tracing of events through simulation and the inclusion of visualisation[10]. However, ns-3 still has no core support for a graphical user interfaced and the technical knowledge required for its use remains a barrier to students of network algorithms.

3.2 Cisco Packet Tracer

Specifically aimed at the student market, the Cisco Packet Tracer tool[2] allows the physical hardware of a network to be simulated and provides informative visualisation of events. Packet tracer provides an environment for student users to tinker with the workings of realistic networks, including low level hardware such as routers and hubs. This tool has already seen use in Cisco's Networking Academy and has received positive critical appraisal from students[16, 12]. In comparison to the functionality of this project, Cisco Packet Tracer work at a for lower protocol layer.

3.3 Sinalgo

The Sinalgo framework is one of the few tools to provide the high-level abstraction over network simulation that this project is aimed at[7]. Although primarily aimed at wireless communication, Sinalgo allows user's to specify algorithms in a Java format similar to what an embedded mote device might view. In addition, users can view visualisations of their simulation. Unfortunately, Sinalgo has seen little development since 2008. Sinalgo provides much of the inspiration for this project, building upon its simplistic interface and visualisations to give the user the optimally abstract view of a network.

3.4 IBM TPNS

TPNS (Teleprocessing Network Simulator) is a software testing tool used to test real world systems against a simulate network interface[11]. The network level of TPNS is low, able to simulate an entire protocol stack. In addition, TPNS can simulate a TCP/IP server or client. Through these means, stress testing and black-box acceptance testing can be administered. However, TPNS is aimed solely at industrial system which interact on a very low level. This form of simulation is that which the project intends to avoid.

4 Requirements Specification

4.1 User Requirements

4.1.1 Functional User Requirements

The produced system must provide users the following functionality:

Network Configuration construction and storage

1. A user of the system shall be able to design the topology of a network as a collection of nodes and arcs.
2. A user of the system shall be able to specify the latency between elements within a network.
3. A user of the system shall be able to specify the configuration of a network outside of the system with ease.
4. A user of the system shall be able to store the topology and component properties of a network for use at a later point.

User Implemented Nodes

1. A user of the system shall be able to define a node as a Java class.
2. A user of the system shall be able to specify code to be run on a node prior to simulation.

3. A user of the system shall be able to define the execution process of a node.
4. A user′s node definition shall be able to send and receive messages from neighbouring networked nodes.
5. A user′s node definition shall be able to wait for a period of simulation time.

Simulation of User Nodes

1. A user of the system shall be able to view statistical information (see appendix A) of a simulation.
2. A user of the system shall be able to specify the configuration of a network used in simulation.
3. A user of the system shall be able to specify the node-definition of the nodes used in simulation.
4. A user of the system shall be able to specify the occurrence of node failure events during simulation.
5. A user of the system shall be able to store a log of a simulation for use at a later point.

Visualisation of Simulation

1. A user of the system shall be able to view the events of a past simulation as a visualisation.
2. A user of the system shall be able to interact with a visualisation to jump to a particular point in simulation.
3. A user of the system shall be able to accelerate or decelerate the passage of time in simulation.
4. A user of the the system shall be able to view the events of simulation in reverse.

User-defined Testing

1. A user of the system shall be able to test the correctness of a designed node.

4.1.2 Non-functional User Requirements

1. A user of the system must find the user interface to be intuitive and aesthetically pleasing.
2. A user of the system must find the Application Programming Interface given for designing custom nodes easy to make use of.
3. A user of the system must be notified of any unexpected error caused in simulation, distinguishing between those resulting from the user code and those from the system.
4. A user′s node definition must be uniquely identified within a network.

4.2 System Requirements

4.2.1 Functional System Requirements

To meet the specified user requirements, the system must do the following:

Configuration construction and storage

1. The system GUI shall provide the user with a graph-based "drag-and-drop" interface for designing network topologies.
2. The system GUI shall provide the user with a form for the specification of network component properties.
3. The system shall allow for the storage on, and retrieval of network configurations from, the local file system.
4. The system shall store network configurations on the local file system in a human-readable, human-writeable form.

Implemented Nodes

1. The system shall provide a Java class that a user can sub-class to define a custom node's execution process.

Simulation of User Nodes

1. The system shall provide the simulation of a network user defined nodes for a specified period of time.
2. The system shall base the topology and properties of simulated nodes on a network configuration specified by the user.
3. The system shall profile the simulation of each user defines node during simulation.
4. The system shall log the events that occur in simulation.
5. The system shall take a list of node failure events prior to simulation from the user, and invoke these events during simulation.

User-defined Testing

1. The system will provide a well documented test simulator class for repeated use in unit testing.

4.2.2 Non-Functional System Requirements

1. The system must provide each simulated node within a network with a unique machine number.
2. The system must ensure the timeliness of network events to an accuracy of one timestep.
3. The system must validate user passed files, providing errors on corrupted data.
4. The system must be secure enough to prevent a user from bypassing the in-built message system.

5. The system must be executable from the Computer Science laboratory machines.

5 Software Engineering Process

In the following section, a description of the methodologies and tools used in the production of the end-system is provided. The combined use of these tools has allowed the smooth development of the code and a better guarantee of suitability for purpose. In particular, the need for reliable, robust code was emphasised. This was a conscious construction choice, based upon the premise that the end-system is intended for use by untrusted users, of whom some may benefit from the subversion of the system. Ensuring a lack of available exploits was of key importance, making the need for reliable code paramount.

5.1 Requirements Engineering Process

The requirements of the system are primarily taken from the experience of students within the Computer Science department and though consultation with the project supervisor. As the possible extensions for the project were varied, regular consultation with the project supervisor allowed new requirements to develop with regular feedback as to how these requirements could be met.

Due to the changing nature of the requirements, a full specification was delayed until the end of the first semester of work. This allowed most of the requirements to confirmed and given enough of a base to the system to foresee the development of more advanced requirements.

5.2 Development Methodology and Tools

The development of the system focussed around an iterative model of construction. This was primarily in aid of the flexible nature of the requirements engineering process, but also allowed feedback from each iteration to affect the next. The overall cycle of development was as follows:

1. Identify, building upon the progress of the previous iteration, the most important objective that is within reach of completion.
2. Note the requirements that must defined to meet this objective.
3. Define a high level design of the code required to meet these requirements.
4. Implement a solution to this design.
5. Demonstrate added functionality to project supervisor to ensure suitability.

5.2.1 Test-Driven Development

The inclusion of Test-Driven Development within the construction of the system was an obvious choice. As *TDD* allows for steady, repeatable development testing of all changes made to the system, it provided an excellent means for catching errors that were a result of further appended changes, amounting to a strong guarantee of code quality. The causation of errors within the existing code base as a result of a resent change was of particular concern in this project given the complex nature of the simulator needed. Furthermore, the non-deterministic effects of multi-threading allows for these changes to be hidden in simulation even though they may be present.

An exception to the code covered by unit tests is that of the simulation visualisation and the construction of network configurations. This was because of the difficulty in testing graphical representations of data from a unit test. In these cases, sample programs were wrapped around the code to provide an application that would demonstrate how the components appeared to the user.

5.2.2 Construction Tools

The construction of any sizeable code base can be greatly aided though the use of proper software development tools. To ensure the smooth progression of implementation and code validation, several key tools were used.

Revision Control

The use of revision control tools has become commonplace in modern software projects. By allowing the commitment of changes to a project in small increments, the causation of errors can be tracked to small sections of change. This combined with the validation of units of code that Test-Driven Development provides, the scope for programming errors in the project was minimised. When considering which particular revision control tool to use, Mercurial appeared as the best choice given its distributed nature and the existing services running on the provided school servers.

JUnit

To unit test the development of the system in accordance with Test-Driven Devlopment, a unit testing framework was needed. Given the accessibility of JUnit, and previous experience of its use, it was chosen for this purpose. A side benefit of this choice was the ease at which the integrated JUnit testing tools provided by the Eclipse IDE allowed fully automated testing within the development environment, allowing a simple transition between writing and meeting tests.

JavaDoc

In order to make the implemented system more maintainable and to allow any user who has created a node script a well defined API, the source code of the system has been documented with JavaDoc 4 compatible comments.

FindBugs

Findbugs is a static code analysis tool for the Java language that detects many common mistakes and risks within a project even if the project compiles and appears to run as expected[14]. Although the use of unit testing and revision control was already in place to provide the necessary guarantee of code reliability, the extra step of using *FindBugs* again bolstered this confidence.

Common situations during development where FindBugs highlighted risks and errors in coding included:

Forgetting to protect wait calls. A risk of threaded programming is the occurrence of so-called *spurious wakeups*. This is the unexpected notification of threads within a waiting state. Currently, Java has no protection from these, so all wait calls should be wrapped in a while loop that breaks if a condition that can only be met prior to the thread being awoken.

```
while (followers.size() < followerThreshold)
        followerThresholdMet.await();
```

Listing 5.1: An example of wrapping a wait call.

Unlock statements outside of *finally*. To guarantee the unlocking of an acquired lock, all code prior to the unlock should be encapsulated within a try statement, with the call to unlock in an attached finally statement. This ensures that no matter the effect of the code within the locked statement, the lock cannot be kept. An example of where this would be an issue is if an exception were thrown within the locked code. Without the finally clause, the lock would never be released as the exception run back through the stack.

```
lock.lock();
try{
        throw new RuntimeException();
}finally{
        lock.unlock();
}
```

Listing 5.2: An unlock call ensured by a finally statement

6 Ethics

The construction of this system has provided little cause for concern over any ethical issues. As the software is intended to run on shared machines, care has been taken to ensure that no data that could identify the user is generated as a result of its use. The nature of the computation and the data used is not inherently identifying and the user is never required to give personal information.

7 Design

7.1 Overview of High Level Structure

The high level design of the top level application has been modelled as a pipeline architecture of simulation to visualisation controlled by the user, as outlined in figure 7.1. The process of designing an algorithm, implementing it, testing or simulating it and finally reviewing the events of the simulation mapped well to a pipeline architecture, with each stage dependent only upon the previous. An additional benefit of such dependence is that it allowed a completely independent construction of stages, working on the assumption that, given the correct input, a stage can produce an appropriate output fitting to an interface given by the next.

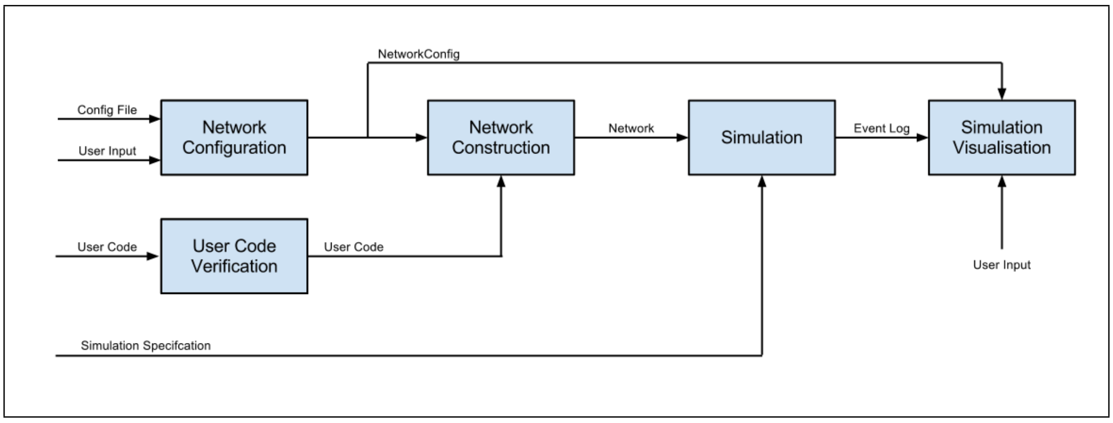


Figure 7.1: Simulation/Visualisation Pipeline

Network Configuration The topology and any prescribed node failures of a network are constructed as a graph, with properties attached to nodes and arcs that reflect those in the simulated network. This provides a description of how the properties of the elements within the network change as a result of outside forces, in addition to merely its structure. The configuration of the network can either be specified by hand as a *JSON* formatted configuration file (matching the specification provided in appendix C), or through the graphical network designer provided by the GUI.

User Code Validation Before any code can be simulated, validation must be applied to ensure that this code does not include statements that could abuse the simulator. Most notably, user code must not be able to circumvent the message passing system between simulated nodes.

Network Construction Given the configuration of a network and the validated code script that each node will execute, this stage constructs an in-memory representation of the network in preparation for simulation. The generated *nodes* of the network are imbued with a copy of the user code script and are connected with message passing links to allow future communication.

Simulation A constructed network is passed to the simulation phase automatically, along with the length of the simulation, any events that a prescribed to occur and an optional limit on the time a node can spend in execution per timestep. This is passed to a copy of the core simulator, which performs the simulation. The core simulator will generate an in-memory log of events that occur during simulation. This is coupled with the network configuration and written to a JSON formatted (see appendix D) file for later visualisation.

Visualisation The results of a prior simulation are loaded from file and a visualisation of these events is created. This allows playback of these events at a user specified rate, pausing and rewinding. The user is allowed to interact with events and messages to provide more information on data contents or profiling results.

As the ability to reuse previous network configurations and replay past simulation event results is a requirement of the project, the design of the pipeline is such that the individual stages can be re-entered with previously stored configurations at the behest of the user. Given the loose dependence of pipeline stages, this is simply done though the serialisation, on-disk storage and deserialisation of inter-stage data, which is implicit within the operation of the pipeline.

7.1.1 Inter-stage Data Storage Format

The decision to store data on-disk in between pipeline stages evolved from the requirement for reuse, yet posed questions of performance and storage medium. First however, it was necessary to consider the prerequisite conditions for storing inter-stage data: what stages of the pipeline would need to store data on disk and was this storage always a necessity? To answer this it is necessary to define which stages in the pipeline the user would wish to re-enter. These would be *Network Construction*, *Network Configuration* and *Simulation Visualisation*. Within these stages the user may wish to tweak a previous network configuration, re-simulate an updated algorithm under similar conditions or re-visualise a past simulation log respectively. Thus, the ability to re-enter these stages with data loaded from disk is a necessity. In contrast to these stages, the *Simulation* stage should not be re-entered immediately from pre-stored data. This is because part the data required for simulation - the in-memory *Network* structure - needs to be passed. Although this could be serialised and stored to disk, it can also be generated from the *Network Construction* and *User Code Validation* stages. The previous stages provide important validation of input data, which the *Simulation* stage lacks. Hence, the design of the system prevents immediate re-entry *Simulation* stage by the user.

The data re-used in the Network Configuration and Network Construction stages are pre-constructed configurations. A variety of storage formats were considered for their storage. Succinctness in information storage was a desirable feature of any chosen format as not to limit the scale of a network through impracticably large file size requirements. Conversely, the requirement for user constructed network configurations imposes a need for a human-readable/writeable format. The latter requirement suggested common web-based data formats such as XML, JSON and YAML[3, 4, 5]. Of the options considered, JSON fitted the needs of the system best, providing a clean, easily understandable means of representing network configurations. With this choice, an edge list based network format definition was created (defined in appendix C).

7.2 Simulator

The core aim of the project is to produce the behaviour of user code within a simplified network structure. For this purpose, the use of simulation is ideal. The user code is to be executed over a highly abstract network structure, with no concern for low-level protocols. Were the documentation of the more complex behaviours of networked algorithms required, the size and complexity of the simulation would be far greater - perhaps infeasibly. In the given case, simulation is simplistic enough to be considered.

With any non-trivial model, the flavour of simulation used can have wide-ranging effects, likely limiting how the system can be implemented. Furthermore, if poorly made this choice can have serious performance implications. It is is because of this that several key variations of simulation were explored for use within the system. The design chosen was a discrete-event simulator, custom built for the project.

7.2.1 Preference for Custom Simulator over 3rd Party Framework

Computer based simulation is well researched area with many well supported simulation frameworks available. Given this, the use of a 3rd-party would be preferential to one specifically built for the purposes of this project. However, through consideration of the project's requirements, several key reasons make the use of a bespoke simulator appealing:

- Users' code must be heavily sandboxed. As the user script for a given node is to be considered untrusted, all its actions must be enacted within an environment where their failure is assumed and that this failure should not affect the execution of other nodes or the simulator itself. This requires that any information of other nodes is hidden from a given node script and that actions that do require interaction with other nodes or the simulator are atomic even given the unexpected failure of the script. Such properties would be difficult to guarantee with the use of a third party library, requiring a thorough knowledge of the inner-workings of interaction between entities within simulation.
- User code that become unresponsive may need to be terminated abruptly. This operation within Java (the required implementation language) is highly unsafe. To ensure that this can be done without risking the consistency of other nodes or the simulator as a whole, thorough knowledge of the underlying simulation and safe methods for node termination are required.
- A complex simulator is not required. The network simulation of a node fits a very simple model of simulation that can be implemented with relative ease.

The combination of these factors has necessitated the creation of a custom-built simulator.

7.2.2 Simulation Method

When considering common methods of simulation, the most popular has been *discrete-event simulation*. However, in the interest of the simulator's suitability for the project, several variations of simulation were considered. Following is a summary of these methods, contrasting their respective benefits and costs.

Continuous vs Discrete-Event

One such common method of *continuous simulation*. Within a continuous simulation, the system modelled is described by a representation in which state variables change continuously with respect to time. This allows the entire progression of the system to be described using differential equations. In simplistic models, the differential equations may be simple enough to be solved for specific input values from time zero, making the retrieval of results from simulation simple and efficient[13, p. 87].

Although such a method is useful for some models (for example simple predator-prey systems), the possible complexity of the differential equations makes them difficult to use for larger systems. Importantly, these simulations require knowledge of how the system is expected to change through the differential equations produced. As the behaviour of the models that this project intends to simulate are written by the user and hence partially unknown, this would be far more difficult in this case, if not impossible. Although it may be possible to allow the user to model the systems behaviour as a set of equations, this would detract far from the simplified API aimed towards novice programmers and would require a far greater knowledge of mathematics and system simulation on the part of the user. Indeed, such a set of equations could be highly complex to model a high-level algorithm and would provide little benefit to the programmer.

In contrast to continuous simulation, *discrete-event simulation* only allows the instantaneous state change of elements within the model of a system at a finite number of points through time. These changes are a result of *events* that are triggered at these points, scheduled at previous times through simulation. The nature of this method required a clock that keeps track of the current timestep within simulation. This clock only forwards the timestep of simulation on the complete processing of all events within the previous timestep. This allows two primary means to timestep advancement: *fixed-increment* and *next-event* advancement. In fixed-increment, the timestep of simulation only advances a set amount with each call to the clock and any events that are in previous timesteps are processed as if they had occurred at the end of this increment. A benefit of this method is it allows a large granularity in the number of timesteps that can be advanced at a time. As a results, fewer points in simulation need to be simulated, although the same number of events will occur. This could provide a performance benefit in a model that would require events at many timesteps in each increment, allowing each increment a large granularity and encapsulating many timesteps. However, this comes at the cost of accuracy in simulation as events are erroneously processed only at the end of each increment. Furthermore, even increments that do not contain events may be considered, adding to the required workload. In contrast to fixed-increment timestep advancement, the next-event method advances the clock to the timestep of the next occurring event. In this case, a fine granularity is given to the processing of timesteps, perhaps infringing upon simulation performance. However, because it is only timesteps that need to be processed - those which have events - this is actually far more beneficial for performance[13, p. 8]. In addition, the fine-grained timestep advancement provides as high a degree of accuracy as the accuracy of the timestep will allow.

For the purposes of this project, the use of discrete-event simulation fits its needs well. Events provide a natural means for the project to represent the actions of each node within simulation can be easily understood and generated through calls to the API on the part of the user. Furthermore, the processing of these events provides an opportune time to log simulation. With respect to the clock advancement mechanism used, next-event is the most suitable. This is because the project simulator is not expected to deal with a particularly large number of timesteps and successful solutions to a problem may terminate far ahead of schedule. Thus, it is likely that fixed-increment simulation would provide a performance decrease compared to next-event, this coming at the additional cost to accuracy that fixed-increment imposes.

7.2.3 Hierarchical Structure

The design of the simulator is such that model knows nothing of the simulator, and the propagation of events within simulation is generic enough to work for any model. This allows changes within the problem definition and the simulator without modification of the other. It is also in-keeping with information-hiding design principles as the model has no knowledge of the simulator.

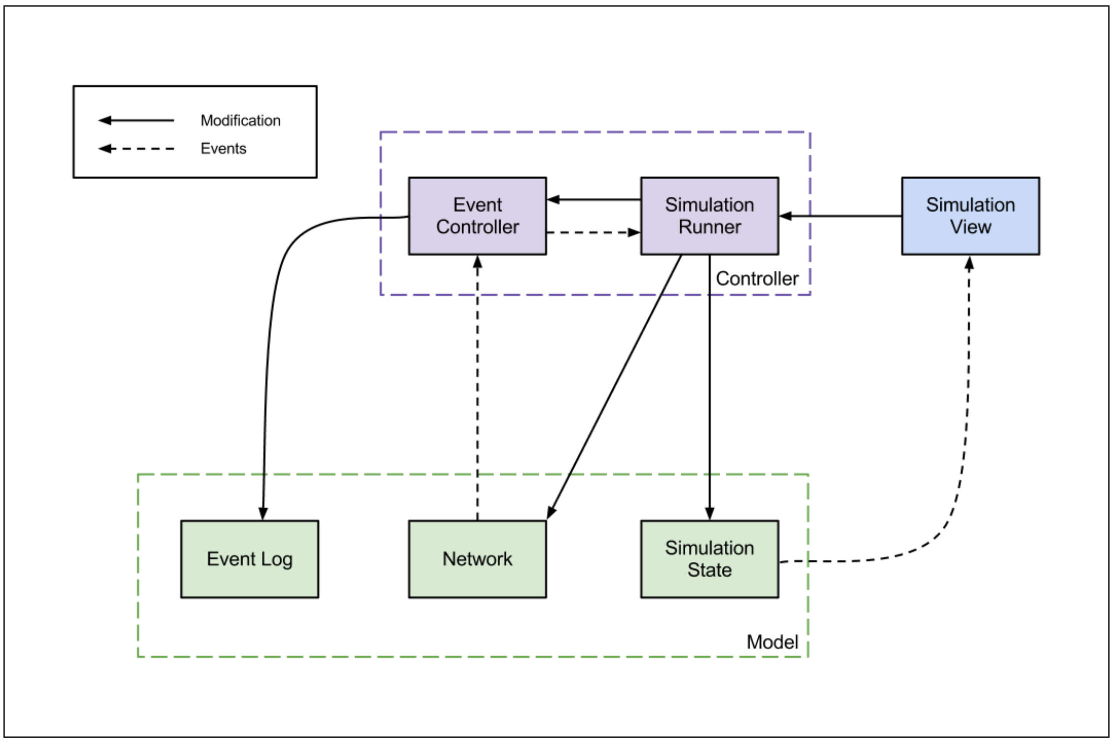


Figure 7.2: A high-level description of the simulator.

The overall structure of the simulator matches the *model-view-controller* design pattern. Changes in the network - events - are passed to the *event controller* and, if of a high enough priority, on to the *simulation runner*. Advancing the timestep processes these events, affecting the model as either changes to network properties (adding a message to a message queue, for example), logging these events in the *event log* or modifying the state of the simulation.

Simulation Runner Initiates the simulation, creating the network of simulation nodes and advancing the timestep of simulation. The simulation runner provides the trigger to all operations in simulation and ensures that the processing of events stay within a consistent order.

Event Controller Manages the scheduling and timestep-ordered processing of events within the simulation. As events are processed, the event controller adds them to the event log for use in visualisation at a later point.

Network The model of the simulation. A collection of nodes and links mapping to the topology of the network configuration. Each node is given a copy of the user's node script. This includes means for the queueing of messages, and the execution and pausation of node scripts.

Event Log A log of all events that are processed during simulation. This is kept as a record for visualisation, so that simulation can be reconstructed. The event log stores the triggered events in a timestep ordered fashion, allowing this data to be conveniently accessed in visualisation. In addition, the event log keeps track of all messages sent in simulation, kept separate from the event list in order to aid the speedy construction of a visualisation matching such a structure.

Simulation View Although the core simulator requires little interaction on the part of the user (besides the initial parameter of simulation), some feedback to the success of a simulation is needed. For this purpose, the simulation view allows the success or failure of simulation to be displayed to the user as a popup box, with a stacktrace of the offending exception triggered in the case of the latter.

7.2.4 Node State

The state of each node fluctuates throughout simulation as a result of stimuli from other nodes and the programmed operation given by its user-script. All nodes are given an initial state before control is passed to the user-script, which continues execution within its own thread until it reaches a blocking action or times out.

The decision to run each node within its own thread is an important one. As a node sends a message, it may allow another node - waiting for a message - to resume execution. As a result, the ordering of node execution is important, in particular, ensuring that all nodes that can be resumed before the timestep of simulation is advanced are done so.

Given consideration of this requirement and the ease of implementation of various options, two primary options were apparent. The choice was between the *Thread-Per-Node* model and

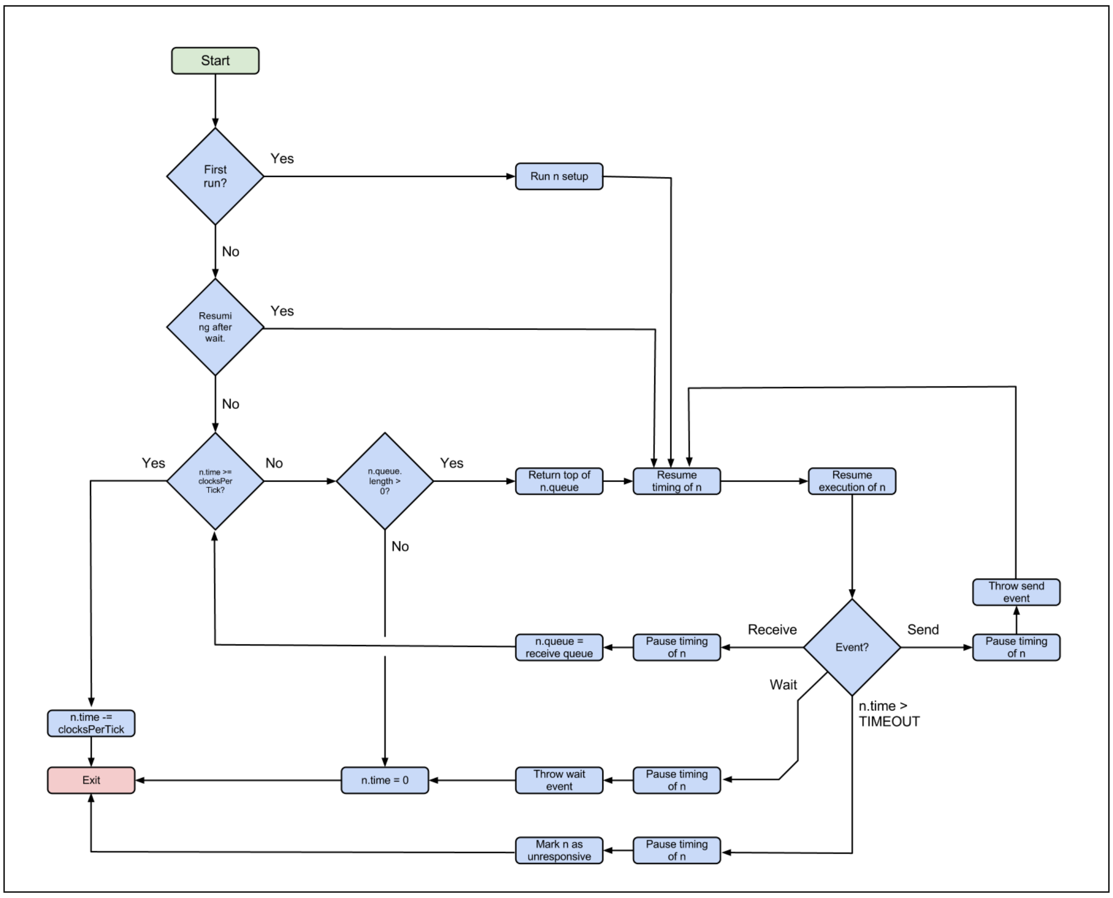


Figure 7.3: Node simulation flowchart.

a queue-based model in which the blocking and unblocking of nodes is treated as a batch-processing operation.

The latter model was an attractive option for its sheer simplicity, requiring little in the way of thread creation overhead compared to Thread-Per-Node. However, the Thread-Per-Node model was deemed the better option for several key reasons:

- The use of a thread to implement the user-script should provide any student user who has worked with threaded Java programs prior a familiar structure and interface for writing their algorithm.
- By confining the execution of the node to a thread, it can be isolated from the execution of other nodes, allowing the node to be sandboxed easily and halted if necessary by another thread, as is the case with non-terminating nodes. This would be far more difficult with a queue based system, likely requiring additional filtration of dequeued items.

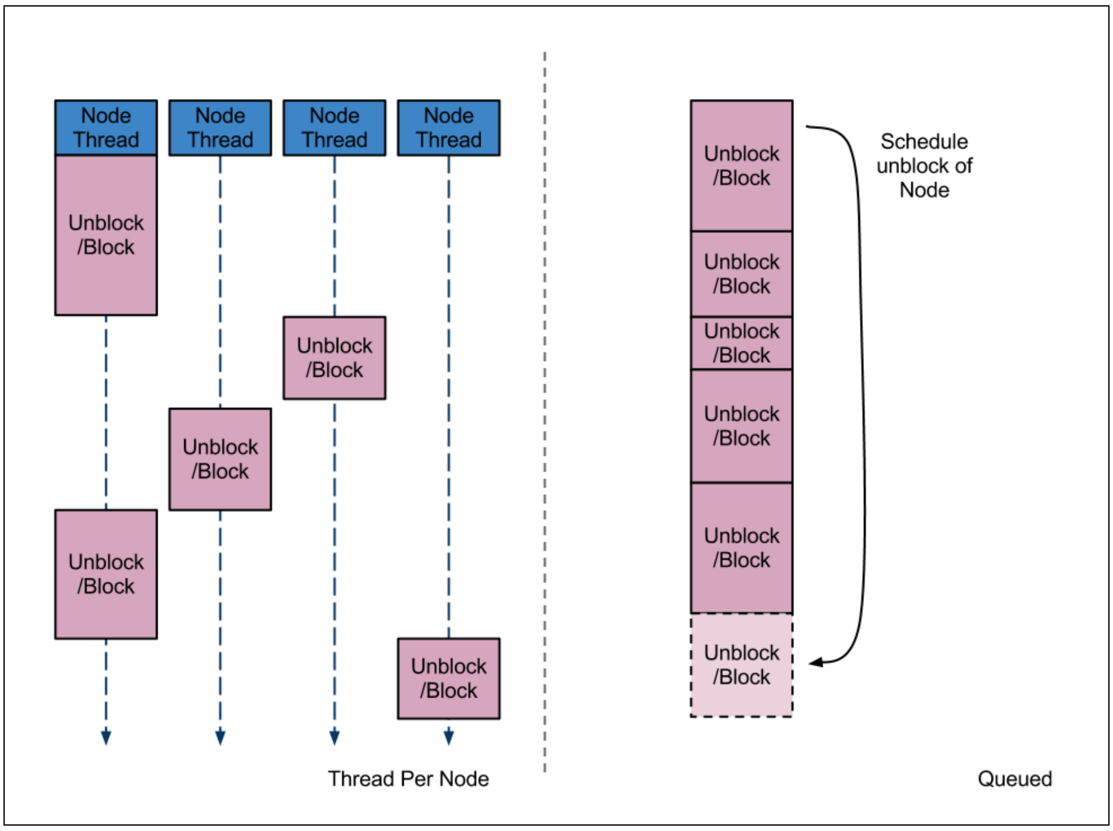


Figure 7.4: Considered node execution models.

- Most simulations, through tendency to use a large number of nodes, can take advantage of multi-core machines, with each core able to simultaneously run the execution of a node. Given that the single thread model of the queue-based alternative would not be able to run across multiple cores as actively, it is likely that the Thread-Per-Node model gives a greater multi-core performance, despite the overhead of thread creation.

7.2.5 User Code Sandboxing

The requirements of the system enforce the predictable simulation of user code and a resilience to failures on the part of the user code. However, this code has no guarantee of safety, risking the throwing of runtime exceptions. To meet this requirement, should a failure occur in user code, the rest of simulation should not be affected. This is to provide resilience in the execution of a each node in the network, allowing partially correct simulations and aiding in the debugging of user code. In such a case, the system will mark the node as failed and attach the exception thrown to an event stored in the event log. By logging a failure event this information is available for display by the visualiser. Importantly, the failure of a node due to an error on the part of the user script should never occur during an interaction with the simulator (the sending of a message, for example). In such a case the simulator could be left in an inconsistent state. To prevent this, the system will ensure the atomicity of any interaction a node makes with any other component of simulation through adequate locking of the applicable methods.

The design choice to sandbox the execution of user code is beneficial for an additional purpose: the prevention of user code abusing the regular use of the simulator. Through sandboxing measures, the following system abuses outlined in the requirements documented can be prevented:

Acting as another node To prevent mimicry of one node by another, the user code is never given access to the node object of any other node. The system will do this by providing a severely limited interface to other nodes, providing only an integer identifier for all neighbour nodes in *send* and *receive* operations. These indexed operations will wrap around the actual operations that are taken by the nodes themselves. By making the underlying node restricted to the user code, there is no means (barring Java *reflection*) for user code to directly access a node within simulation.

Use of underlying methods The user code must not be able to access methods unintended for its use. To hide such methods, this code will be contained within a *NodeScript* class, separate from the body of code that controls actual interaction with the system. This NodeScript class provides a wrapper around only the methods that the user code should have access to, thus preventing such a situation.

Overriding Underlying Methods Due to the nature of overriding methods within Java, the risk of a user removing or replacing the functionality of these methods requires caution in what methods are even accessible to that user. The best means to prevention is to simply provide as few methods as possible in the node script class as possible. However, for those that are required, the results of the method must be checked after their call.

7.2.6 Simulation Profiling

In accordance with requirements, the design of the simulator includes a profiler that maintains the metrics defined in appendix B. This profiler is passed to each node on their creation and is updated with new timing data every time the node changes state between simulating and not. This timing data contains how long the node has been simulating for, which is added to a count of the total time in that timestep of simulation for that node. This keeps track of *local execution time* for each node to a millisecond degree.

This timing information has an additional use. By maintaining the time at which a node begins its most recent bout of execution, the profiler can detect when the user code script of a node appears to be non-terminating. This is done by allowing the user to specify a timeout at which a node is considered unresponsive. Once the profiler is created, it regularly searches through its list of registered nodes, comparing how long each node has been executing since it began its most recent simulation. From this, any nodes that exceed this timeout are terminated and an event marking their death is propagated to the event controller.

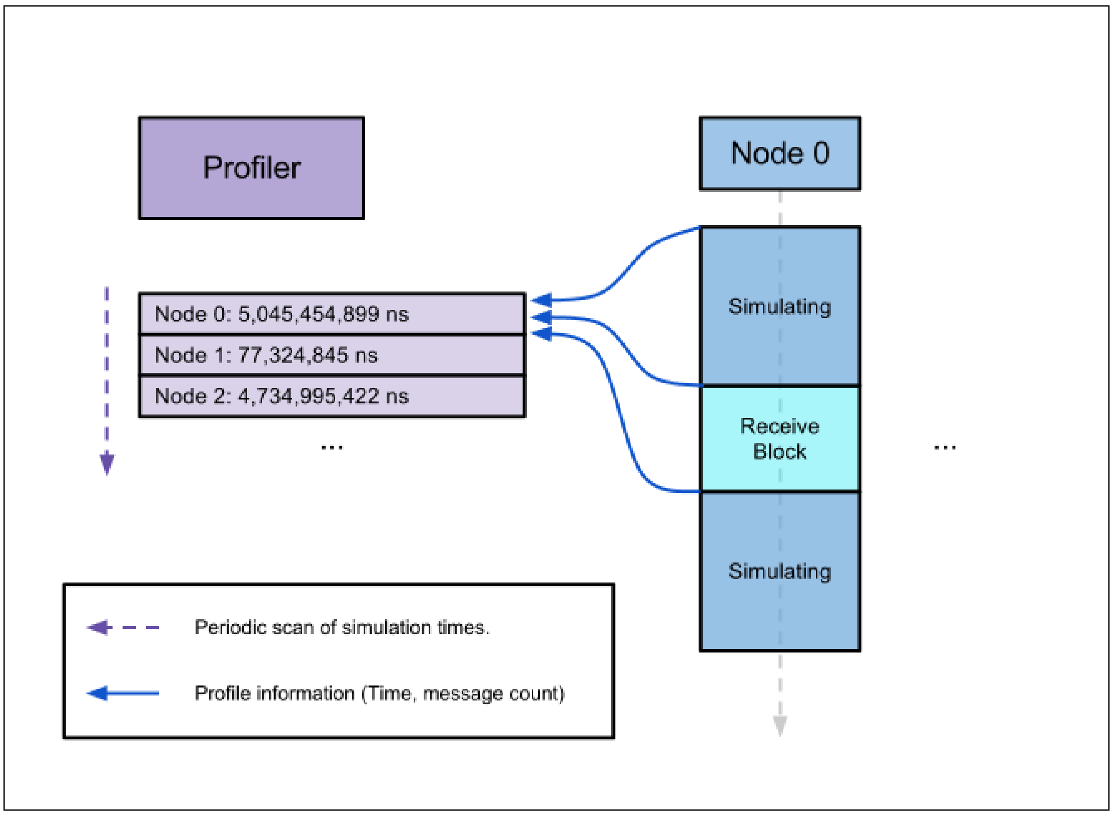


Figure 7.5: Simulation profiler.

In addition to timing information, the system must keep a count of how many messages each node sends, receives and reads. Although this information could be collected at the simulation stage, this is unnecessary as it can be peeled from the event list passed to the visualiser.

7.2.7 Event Queueing and Propagation

To maintain the consistency nodes within simulation, any event that affects simulation must first be scheduled at a common *event controller*. This controller maintains a queue of events ordered by the timestep in which they occur. This allows events for varying future timesteps to be scheduled. From this queue, events are removed and processed once the timestep of simulation reaches that of the foremost event, in-keeping with traditional discrete-event simulators.

To maintain a well-defined hierarchical structure to simulation, events within simulation must have attached all the information needed to enact the event. This allows use of the delegate design pattern, a *process* method for each event, called by the event controller on the processing of the event. Furthermore, the event controller does not require any knowledge of the model it works upon. This abstraction provides a loose coupling of the simulator to the model simulated, in-keeping with engineering practice.

7.2.8 Event Logging

The aim of simulation is to obtain an idea of how a user's code has performed in the constructed model. To this end, a log of all events processed by the event controller is maintained. this log provides a timestep ordered history of these events. On successful completion of simulation, the log will be attached to the network configuration and any simulation parameter to be stored in a file representation of the simulation. In addition, all messages in simulation are scraped from the event log and included in a separate log of messages. This has the benefits of allowing the visualiser easier access to messages and removing the duplication of messages within events in the event log. For example, the same messages can appear in an event for both its sending, arrival and reading.

7.3 User Code Validation

As the system is required to take unknown user code for use in simulation, there is an opening for users to abuse simulation by writing code that attempts to deny simulation or subvert the message passing system in place. These abuses, defined in the requirements document, can be partially combated with user code sandboxing. However, in the case of others, static analysis of user classes provides the easiest means of detection and prevention.

One of the easiest means of abuse is the use of static stored variables. By allowing data to be shared amongst all members of a class, a NodeScript can easily be imbued with an instantaneous (in terms of simulation timesteps) means to communicate some value between any member of a network. To prevent this, the simulator will provide an optional filtering of all user classes that disallows the use of static fields. An exception to this abuse is where primitive constants are used in code. Although a constant is static and shared by all, a final modifier prevents it from changing and hence it cannot be used for communication. Of course, for object types, although the object reference cannot by changed, the date within object itself can still be, hence not all static final types are safe. As a result, the system also allows the combination of static and final modifiers to be filtered at the discretion of the user.

7.4 Network Configuration

The construction of each network prior to simulation is specified through user-defined network configurations. This format provides the network topology and properties of the individual links and nodes within the network. Importantly, this format lends itself to easy configuration from the user, a requirement of the project. The user is able to create a network configuration in two ways: by hand or through the system's graphical network creation interface. Combined, these methods provide two main benefits: understanding of the network and ease of creation.

7.4.1 By Hand Network Definition

As per the requirements, the system must allow the user to specify the network configuration used in simulation by hand. The reasoning behind this being that a student user can gain a far better understanding of how the network is constructed if they must spend the time to specify every link themselves. To this end, the system is designed to take network configurations as a JSON formatted file. The decision to define a network as an edge list in this manner, with only a node count, was made to keep network definition simple and easy when written by hand.

7.4.2 Graphical Network Constructor

Although specifying network configurations by hand does enforce an understanding of network topology, it can be tedious to the point of frustration should modification of the network be required in the case of any large network. Similarly, if the network must follow a rigid pattern. In such cases, only allowing the user to specify the configuration by hand would be at the cose of the simplicity that it allows for smaller networks.

To counter this, the system is designed to include a graphical network creation tool that provides a *drag-and-drop* interface for the user to define more complex networks easily. With this the user can add and remove network components within a few clicks and can create complex network topologies. This interface will allow constructed in-memory design to be written to the same format as the hand-written. Previous network designs will also be able to be loaded from file and appended.

To provide the user with such an interface, the use of a third party visualisation library was chosen as a sensible option. Many such libraries already exist, saving the time and effort required to implement more complex graphical manipulation techniques. Those libraries that focus primarily upon data visualisation - and in particular graph visualisation - provide an excellent foundation for this task. The use of a graph visualisation is ideal as a network naturally maps to a graph structure.

Considered Visualisation Libraries

The primary desire for the use of a third party is ease-of-implementation. As a result, any chosen library should provide a stable platform for the representation of network configurations. The required degree of functionality should not hinder development with an overly complex API. The following are a range of frameworks considered, with explanation of their suitability for the purpose of visualising network construction. For compatibility with the system, only Java-based frameworks were considered.

Processing Processing is an open-source development environment, based upon Java 1.4, that supports draw-loop graphical applications through a particularly simplistic API[6]. The sizeable environment, combined with such an API, allow rapid development of hand-built visualisations. Such a system would allow the display of the network environment with little required effort for the rendering of items. However, Processing provides little functionality for more complex visualisations. A considerable amount of work would be required to build the fluid movement, element creation, element removal and element selection that the interface requires. Furthermore, Processing suffers from a lack of extensibility and is structured in a rigid manner that would force the system to work within the draw-loop.

Prefuse The Prefuse framework provides a large, highly extensible toolkit for information visualisation of large and varied datasets[9]. Prefuse works through use of the *Java2D* graphics library, allowing use of commonly familiar canvass tools. With a *visualisation pipeline* that filters source data, renders it and provides user interaction, Prefuse offers more than enough functionality to accomplish the task of visualising network configuration. Ingrained within the framework are abstractions over the filtering of source data and their visual representation, allowing new groups of data, layouts, colour-schemes and even more complex visual objects for use in representation. In addition, Prefuse has already been used in many large-scale visualisation applications[8]. As a result of Prefuse's wide-ranging functionality, the use of this tool would require considerable research into the workings of the visualisation pipeline to make proper use of it, far more so than Processing. This said, the abstractions and structure that Prefuse provides are well documented and simple enough to make it a serious candidate.

Given full consideration of these candidates, the chosen library was Prefuse. The combination of strong, demonstrated functionality, with a relatively small learning curve given its complexity made Prefuse an obvious option.

7.5 Results Visualisation

To meet the requirement of simulation visualisation, included is a visualisation tool that displays the passing of messages during simulation in a graph-based construction. This tool reads files of the format produced by the simulator, constructs the specified network configuration as a graph - where simulation nodes and link match to visual counterparts. This allows playback of messages being sent through simulation represented visually on the link they were sent though.

The primary design decision for this section of the system was the choice of suitable visualisation library. Although a custom visualiser would not have been infeasible, there were few constraints on visualisation that would prevent the use of a third party library. One such constraint is the graph-based manner of visualisation. Thus, the chosen framework would ideally provide data and graph-based visualisations. In addition, any framework chosen must be extendible enough to allow the additional messages and node state data to be visually represented with the network configuration.

All of the previously selected frameworks for the construction of network configurations were consider for use in simulation visualisation. Of these, prefuse again appeared as a clear choice. In particular, the extensibility of prefuse actions and the flexibility with which these actions could be queued allows the manipulation of visual data - such as the links in the network - a simple task. In addition, much of the functionality in representing and manipulating the network configuration in the configuration editor is duplicated in the representation required by the visualiser. As the simulator attached the network configuration to simulation results in exactly the same format that these configurations are loaded by the configuration editor, even the code for reading a configuration from file could be re-used. Thus, there was more case for code re-use by choosing prefuse.

7.5.1 Playback

To allow the user of the system to view the events that have occurred in the simulation of their code, the visualiser design is such that they can playback these events in the same manner to a video, increasing and decreasing the speed of playback at their choosing. During this playback, the transmission of messages between nodes are represented as boxes moving from the source node to the target node. The position of each message whilst in transmission is a position on the link it was sent as a ratio of when it was sent to when it arrived.

To maintain the smooth transition of messages from source to sender, each timestep of simulation is subdivided into a number of "ticks". At each tick, the clock controlling playback announces to any listeners the new state of the clock. This allows the visual display to update to location of message representation in between timesteps, thus avoiding the instantaneous jumping of messages from source to target. The number of ticks per timestep used by the clock is scaled depending on the rate of playback specified by the user. This allows the the update of the display to stay at a consistent rate regardless of this playback speed, again ensuring smooth animation.

In addition to displaying their transmission, the visualisation allows for user to interact with elements within the network. By clicking on a message object, the object expands to display the full tag, data and time of sending. As this data is fairly small (both the tag and data are limited to 32 characters), their expansion is allowed as it is unlikely to clutter the screen even in the case of a very active network. If nodes are clicked by the user during visualisation, a panel including profiling information to display profiling information. This panel displays these statistics for any timestep that they change. As this can possibly be quite sizeable, the panel is attached to side of the screen and only display one node at time.

Node Colouring

To provide the functionality to annotate the visualisation of user nodes, the API given to the user provides a call to change the colour of the node executing that script. This call creates a change colour event which, like all other events, is stored in the event log. This event corresponds to a matching change in colour in the visualisation for the timestep that the event occurred. This colouring method is also used to signify the failure of a node, in which case the node is coloured grey for the timestep it failed.

7.6 Custom Node Death Event Triggering

As a requirement of the project is to allow the user to test her code for robustness of an overall algorithm, the design of the system provides a means for the user to specify the unexpected failure of nodes within simulation. As part of the parameters to simulation, a number of node death events can be passed, a maximum of one per node. These events are then passes to the event controller to be queued for future processing in the same manner to any other simulation event, and by virtue of the event controller design, no modification to the existing design needs to be made. On the processing of these events, the underlying thread that execute the specified node is terminated. Care is taken in the design of these nodes to ensure that this termination - although unsafe - can only occur outside of interaction with other elements of simulation through the locking of node methods. To allow re-use of these event lists, they are attached to the storage of the network configuration of the network in which they affect.

The front-end user interface to the system provides the means to specifying node deaths as a table in rows are added by the user which matched the machine ID of a node to the timestep of its failure. This table is appended to the graph constructor, allowing the user a clear understanding of the network in which they are specifying failures.

7.7 Unit Testing Framework

One of the core requirements of the project is for the user to be able to test the correctness of her algorithm. For this purpose the system was originally designed to allow the the user to set an expected value from all nodes and require the user to implement a method that would allow the retrieval of each node's best guess at any time. This method has the benefit of allowing the system to regularly test whether the algorithm has met its goal state periodically and would allow the timestep at which it is met to be identified. However, this is a clunky means to testing and would be only limited to one clause for correctness.

A better design for testing correctness came from the test-driven methodology used. As the tools for unit testing exist and are highly accessible to even novice programmers, all that was required to allow unit testing simulations was a pausable simulation runner, much like that used to run the simulation normally. Combined with access to all node scripts running in simulation, this allows the user to write any number of unit tests and test the values stored in their scripts to test for correctness.

8 Implementation

From the design specified in the previous section, the following text is a description of the implementation details of the subcomponents of the system. Throughout the construction of the system, there has been a focus for good programming practice. This includes parameter validation for most method calls, appropriate wrapping and re-throwing of exceptions, and information hiding.

8.1 Simulator

The classes used to implement the simulator map to the components of the simulator design. The *SimulationRunner* class maintains the entire simulator whilst it is executing. The first step the runner makes on receiving the order to begin a simulation, it spawns a thread in which all its operations are performed through. This step was taken as a result of bug discovered in unit testing in which the thread that had instantiated the runner was not that which made the call to the simulate method. The SimulationRunner makes used of a thread to co-ordinate the event log and node threads at creation time. If a different thread is used to simulate the code, deadlock can occur as the simulator attempts to wake the incorrect thread. Through the use of a dedicated thread this can be avoided, but requires the thread that makes the call to simulate to wait for the completion of the dedicated runner thread.

8.1.1 Node Execution

Once it has begun operation, the dedicated runner thread first initialises all nodes through a setup method. Within each node this method creates a thread that begins to execute the user defined code. In the first timestep, this requires that the user's *setup* method is called for all simulated nodes, before seamlessly moving to the *execute* method. Care is take to sandbox setup method to prevent the user from make any calls to pause or receive on the part of the user code. These operations are confined to the execute method as they can cause the execution of the node thread to wait. Should these methods be invoked within setup, inconsistent state within the simulation would occur wihtout these protections.

Once the node has entered the execute method, the key operations of the user script are calls to either send messages to neighbouring nodes, attempt to wait for incoming messages from neighbours, or to pause the execution of the node for a number of timesteps. In the case of calls pause and receive where no messages are available, the execution of the node needs to pause. This is implemented through the waiting of the node thread that initiated it. All calls to pause, send and receive generate events that are fed to the event controller, who stores them within a priority queue based upon their timestep.

Once all nodes within a timestep have reached a waiting state, the SimulationRunner knows that it is safe to advance the timestep and queries the EventController for the timestep of the earliest event it holds. Assuming that the queue of events maintained by the Controller is non-empty, the SimulationRunner begins instructing the EventController to process all events it contains within that timestep. These events may include the unpausing of nodes in a paused state or the arrival of a messages to a node that is waiting for one. In such case, the Event-Controller - via a delegate method that handles the processing of such events - will notify the nodes from their blocked state.

8.1.2 Concurrency Issues as a Result of Incomplete Node State

Although the design of simulation advancement was correct to wait for the blocked state of all nodes in simulation, through testing of a tree leader election algorithm simulated by the SimulationRunner class, two serious concurrency issues were discovered. It is paramount importance that the methods accessible to the node threads are adequately synchronised as they are accessible by the SimulationRunner thread as well as the node's. Initially the synchronisation of the methods was through the Java *synchronized* keyword. However, it was observed that a receiving node could pass the monitor of the thread before completion of the receive. In addition, the parent thread of all node threads can incorrectly jump to the next timestep before full completion of all nodes.

Given the current system at the time, it was possible for a Node to arrive in an inconsistent state during a call to receive. This occurred when thread that has reached the receive method has notified the SimulationRunner that they are in blocking state, allowing the SimulationRunner to advance a timestep before that Node's thread had fully reached a wait call. This is because event though the simulation should advance to the next timestep when all Node threads are in a waiting thread state, by locking through only use of the synchronised keyword, Nodes must first inform the runner that they will wait. To do this, a count of how many nodes that were currently executing was kept in the SimulationRunner, which was modified through the events that propagated to it. Premature timestep jumping occurred when the count of running nodes decrements to zero - hence releasing the runner Thread - before all Node Threads had reached the waiting state. Should an event that would affect such a node, it is possible for the send to arrive before the node is waiting. This means that, although the state of the Node would allow this send to wake the Node Thread, this notify falls on deaf ears. Eventually the Node Thread does reach the wait state, but may never be resumed if it is was only expected to receive that message. In such a case, the system will tend to end prematurely, as these messages may never appear to arrive causing the events that would have occurred as a result to never happen.

The primary failure that allowed these issues was poor consideration in code design for state inconsistencies that concurrency could lead to and a misunderstanding of how the Java *synchronize* statement worked. Although unit testing has been employed throughout the development process, testing concurrency issues had proven difficult as unit cases often cannot generate behaviour of complex interactions.

To solve this issue, rather than attempt to design a suite of tests that would prove that the system did not fail under concurrent threading, it chosen that an outline of certain thread-safe "policies" that, once implemented, would ensure that the system can never reach these inconsistent states. It was perceived that it would be easier to test that the system meets these policies than prove it had no concurrency problems.

To define the thread safety policies of this code, the following definitions applied:

- *Pause Block*: The state a Node enter when it makes a call to *pause()*.

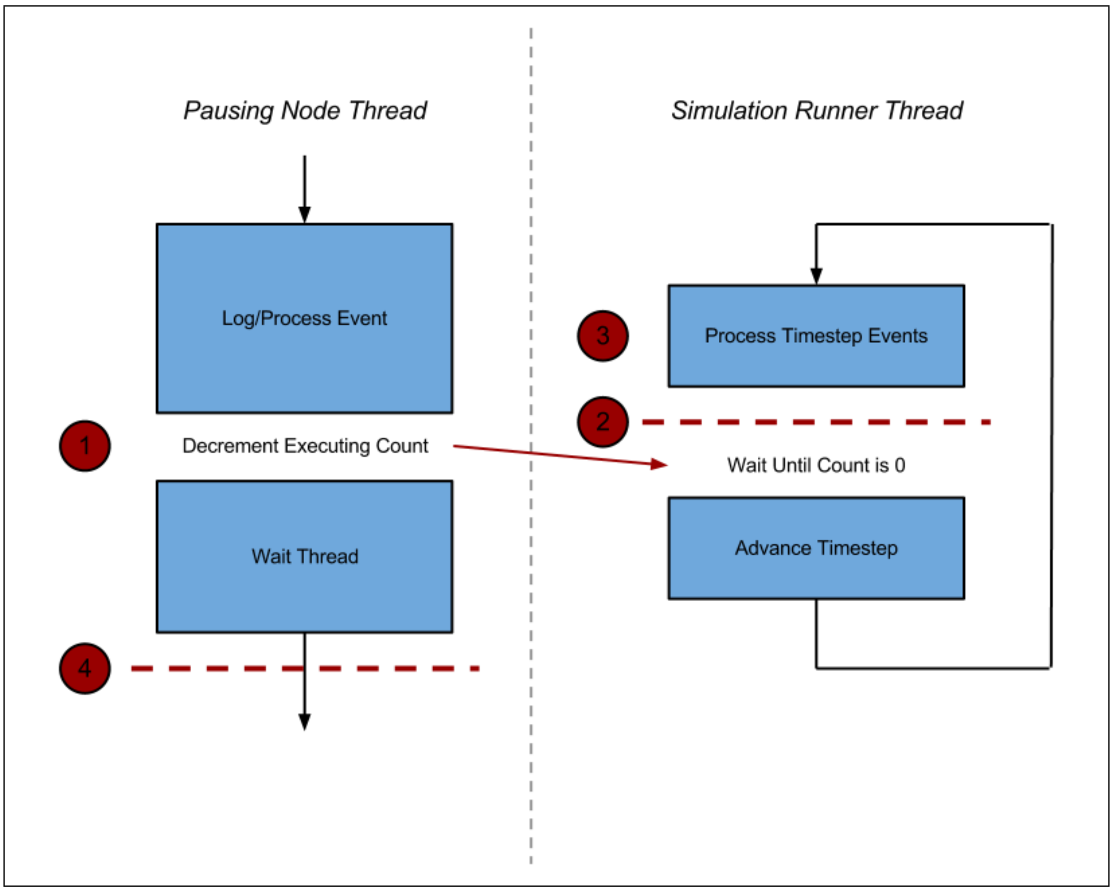


Figure 8.1: Premature timestep advancement

- *Receive Block*: The state a Node enters where, in a call to *receive()*, no messages are yet contained in the message queue the Node is attempting to receive from.
- *Blocking operation*: a call to the *pause()* method, or a call to *receive()* that results in the receiving Node entering a state of Receive Block.
- *Unblocking operation*: the processing of an *NodeUnpauseEvent*, or a *MessageArrivalEvent* where the recipient Node is in a state of Receive Block.
- *Simulating State*: The state a Node enters passed setup and resumes as the result of an unblocking operation.

These policies are:

1. The SimulationRunner cannot advance the timestep until all Node threads are waiting.

2. A blocking operation will prevent any other changes to the node before the state of the node has been updated.
3. An unblocking operation will prevent any other changes to the node before the operation is completed in full.
4. The thread of Node in an blocking operation will enter or will have already entered a waiting state.
5. An unblocking operation cannot begin until the Node has reached a Receive or Pause Block.
6. An unblocking operation cannot complete until the thread of the target Node has reached a waiting state.
7. An unblocking operation will leave a Node in a Simulating State on completion.

Additionally, the design and creation of an innovative thread container made the protection of these policies simple. This container - termed a *LeaderBarrier* - allows one "leader" thread to continue execution only once enough other "children" threads have also paused execution at the container, yet allowing threads to be prompted out of the barrier. As no such structure exists in the Java standard libraries. This is barrier was partly inspired by Java's *CyclicBarrier*, which allows many threads to wait at a barrier until enough are contained that the barrier breaks and all threads continue execution[17]. The core benefit of LeaderBarrier is that it allows the SimulationRunner thread to advance the timestep of the simulation only once all the Nodes have reached the wait call for that Thread (either because of a call to *pause()* or a receive block).

These policies, combined with the LeaderBarrier design, have solved these concurrency issues with no known defects for large-scale tests.

8.1.3 Halting Node Execution

The design of the system requires the halting of execution for both responsive and unresponsive Nodes. This task, although seemingly simple, can prove to be somewhat difficult within the Java language. This is because the unexpected termination of Threads within the Java Virtual Machine at the behest of another Thread is deprecated as of Java 1.4.2 as this is an inherently unsafe operation.

As the basis for the execution of each Node was already the Java Thread class, the obvious choice for the halting of a Node was the *interrupt* method that allows a Thread in a waiting state to be awoken unexpectedly. In the case of responsive Nodes, they will eventually begin to wait on some blocking operation. Should the SimulationRunner wish to halt the Node's execution, it does so by calling this interrupt method. As this method will set the interrupted flag of the Node Thread, even if it has not yet reached a wait state, on it doing so it will throw an *InterruptedException*. By catching this, the Node can perform a graceful termination, simply returning from the execute method.

However, it is quite possible that the user code will not terminate in a timely manner. As the interrupt flag is only observed at the point of a Thread making a call to wait, user code that never reaches a call to block will be unable to halt with the previous method. To prevent the Node Thread from running indefinitely, it the use of the unsafe *stop* method is required. Whilst this would first appear to be highly undesirable, certain properties of the the Node execution allow it to be permitted. As the Node can only interact with any other part of the system through calls to functions already protected by a locking mechanism, by including the call to kill a Node in this unsafe manner within an acquisition of this lock will prevent the Node Thread's termination whilst it is outside of user code. Thus, no damage to the existing can occur as a result of this. As this is coupled with the fact that the death of a Node within user code is an acceptable state - if the change in state is known to the system - the termination of unresponsive Nodes can be guaranteed to be safe.

8.1.4 Importing User Code

The first step to allowing the user to define a Node within the simulator was to provide them with a Java class with the inbuilt commands the user's algorithm can issue whilst simulating. This is defined in the *NodeScript* abstract class, which wraps around the functionality of the *Node* class, ensuring the user access to only these defined methods. To define her own node class, the user simply extends the NodeScript class and implements the execute method.

So that the user can load her class in at runtime, the system includes a *DynamicClassLoader* class that - through the user of Java's inbuilt *ToolProvider* API - allows the user class to be defined from a string read from a file. After a user class is created, it is passed to an instance of the *CodeValidator* class to ensure it does not break any limitations placed upon it. This Code-

Validator makes use of Java reflection to detect whether fields within the user class are static or final and throws a *LimitationFailureException* in the case that a limitation was not met.

8.1.5 Network Configuration

To generate a network defined by the user, the system first lets the user specify a file path from which the JSON formatted definition for that network is loaded. The serialisation library used to provide access to the JSON format is *JSON-lib*[1]. This library was chosen above others that performed similarly for its ability to infer data from getter and setter methods, allowing this information to be written back to files easily. Once JSON-lib has serialised the text file to an in-memory *JSONObject*, the node count and links are peeled from the object to generate a new instance of the *NetworkConfig* class. By storing the network configuration within a new class, the system avoids possible data type errors or missing data that the flexibility of data storage methods for the JSONObject class allow.

In addition to the validation of parameter types that the use of NetworkConfig allows, the data structure used to store the link configurations within NetworkConfig provides an efficient means to checking for duplicate links within the network configuration file. By storing link configurations as a *LinkConfig* object and overriding the *hashcode* and *equals* methods to match links that span the same nodes (whether source and targets are in the same direction or not), each link can be stored within a hash set inside the NetworkConfig, giving optimum storage and lookup time. To match link configurations, any properties of the link needed to be matched, but the direction of the link needed to be irrelevant. To do this, bitwise operations were applied to the source and target IDs to create an integer with the lower ID as the first 16 bits and the higher as the last.

8.1.6 Network Construction

Once the network configuration used for a simulation has been specified and the code intended to be run on each node validated, the next stage in the simulation process is to construct a network structure matching these specifications. This first requires the creation of an instantiator class that will take the user code class an create dynamic instances of it. From this, the number of nodes specified in the NetworkConfig are created and Links added to match the LinkConfigs.

8.2 Simulation Results Visualisation

Should the user wish to view the visualisation of a past simulation, they first select the "Visualisation" tab in the application frame. To allow them to specify the location of a results, they are provided a path field and file browser. Again, this browser is the default JFileChooser and enters a file location in to the path field on the user successfully choosing a file. On the instruction of the user to simulate a file, the system loads the JSON formatted log of events using json-lib.

Once the JSON is loaded, the following operations are performed by the *VisualisationPanel* to initialise the data required for visualisation:

Clock Creation To keep track of the timestep and sub-timestep (or "tick") of simulation that the visualisation is displaying at a particular point in time, the class *VisualisationClock* is used. This class provides functionality to play, pause and stop the advancement of visualisation playback. In addition, the rate at which a timestep passes can be adjusted, even allowing the playback of simulation in reverse. To allow other classes to easily observer these changes, the VisualisationClock class maintains a list of event listeners, each of which are informed on tick, timestep, play state and speed change events. To ensure the thread safety of multiple observers, the methods of this class are appropriately synchronised and volatile variables declared as such.

Event Conversion To prevent the type checking errors that can be possible with the use of JSONObjects, the JSONArray of events that is included in the simulation results file is traverse to peel these events. On peeling them, the events are converted to *Visualisation-Events* which require the type information of data. These events are stored in a HashMap of their timestep of occurrence to each event, allowing the visualiser instant access to all events that need to be displayed in a timestep.

Message Conversion Similarly to the event list, the message list passed is also converted to a list of type safe message representations, *VisualiationMessages*. These messages, provide additional information, such as the visualisation representation of the target and source node to allow easier positioning of their representations as displayed by the layout algorithm used.

Activity-based Message Log As the visualisation is to display the messages passed between nodes only during the timesteps that they are in transit, it is beneficial to have access to all messages that are active within any given timestep. To this end, a mapping of the set of messages sent from a given node as they are active within a given timestep is created, maintained by the *MessageMap* class. To generate this mapping, the list of messages passed to the visualiser is traversed, and a HashSet of all messages are created for every timestep for which the send and receive time are between.

Node Colouring Data For any of the events given to the visualiser, those which indicate a change in colour are noted to keep track of the colour of the visual representation for any given node. This data is stored within the *NodeColourMap* structure, which makes use of a mapping of any visualisation node to a sorted list of the timestep at which the node changes colour. This allows easy access to the most recent colour change for any node.

Statistics Generation As the count of messages sent, received and read by nodes can be offset to the visualisation phase, a further pass of all messages is made to strip this information. This data can then be accessed by the profiling information panel on the selection of a visual node by a user.

The generation of this data is offset to the visualisation stage of the pipeline to save on the size of the data stored in the results file. Although this requires the regeneration of this data every time the simulation is re-visualised, this is considered an acceptable cost as it is only done once for each visualisation.

Once the prerequisite data has been computed, the prefuse visualisation is created, first generating the network topology with a graph structure. The prefuse framework breaks the process of visualisation into a pipeline of four distinct phases[9]:

Source Data The retrieval of the data we wish to visualise. In the case of the visualisation of our network algorithm, this includes all data within the results file outputted by the simulation stage.

Data Tables From the the source data, prefuse tables are generated that store all the pertinent data within the source data that is needed for visualisation.

Visual Abstractions The data within the source tables is mapped to applicable visual representations through the application of prefuse *Actions*. For example, the network topology that is stored within a prefuse graph (an abstraction over sub-tables for nodes and edges), is represented with a *VisualGraph* object with given properties like node colours, labels and edge stroke widths.

Views The display an interaction of the user with the visual representations we have created is possible through multiple views. In the case of this visualisation, this is done via an awt based canvass with Java event listeners.

Part of the visual abstraction applied to the data visualisation is a force directed layout, causing nodes to be repelled from each other and ascribing a degree of elasticity to edges. This provides a convenient means to space even large graphs, allowing for the clear distinction between large clusters of data. However, although convenient for spacing item, it may be convenient to disable the forces of this layout to allow simple user interaction with elements. To this end, the implementation includes a prefuse action that toggles the force directed layout with the middle mouse button.

8.3 Providing User Testing

To allow the user to unit test their algorithms, the *TestSimulator* class was created. This class provides a well defined wrapper around a *PausableSimulationRunner*. The PausableSimulation-Runner acts like the standard runner, providing simulation of user code. However, as the name suggests, the simulation can be paused to allow user tests to check properties of NodeScripts within simulation. Much like the original simulation runner, the PausableSimulationRunner makes use of a dedicated Java Thread to ensure that the Thread maintaining the overall simulation did not have to be that which created the runner. However, this posed some problems in allowing repeated calls to the simulate method as Java Threads are not intended to be rerun. A solution was created that put the execution of the SimulationRunner's *simulateUntil* method within an infinite loop in the dedicated Threads execution. Within this loops the Thread would first wait, and would only make an iteration of the loop (thus simulating the code), on a notify call wrapped in the overridden *simulateUntil* method.

To further aid the testing allowed by the user, the TestSimulator class also generates a collection of all NodeScripts within simulation, mapped to the machine IDs of the Nodes that they are in charge of. This allows the user to access the methods of the scripts, which can be used to obtain and validate the values as appropriate. Finally, a reset method is included in the TestSimulator that allows the user to restart simulation. This is implemented by simply creating a new PausableSimulationRunner.

9 Evaluation and Critical Appraisal

9.1 Testing

To evaluate the fitness of EasySim for its given purpose, testing of the system against its requirements document was conducted. These test - whose results can be found in appendix A - show that the system meets all of the functional requirements it was set. In addition the majority of the non-functional requirements were met, with the possibility of ensuring that the system is fully secure from abuse by user code. However, this is primarily a result of Java's extensibility. By allowing techniques such reflection, Java provides tools for the abusive user to subvert the system. In addition, it is impractical - if not impossible - to detect where a library used by a user file allows abuse of the system. These limitations considered, the system provides an adequate defence against the easier forms of manipulation.

By virtue of the use of Test-Driven Development, the reliability and robustness of the code is well assured. In addition, positive feedback from the project supervisor indicates that the system meets the needs of the client. Testing of the system has shown it to run easily with hundreds of nodes, reaching limits on the number of threads that the Java machine could create on the laboratory machines before encountering any scaling problems.

9.2 Comparison with Existing Software

The final system meets the needs of a poorly served section of the simulation framework user base. By providing a fully comprehensive package for developing networked algorithms at the most abstract level possible, EasySim distinguishes itself from other educational simulators, such as IBM TPNS and Cisco Packet Tracer. Offered is a tool aimed *specifically* at the development of high level algorithms, something these tools cannot provide. In addition, EasySim provides a vastly more user friendly interface for simulation than industry level simulator like ns.

9.3 Future Work

Although the project has reached all of its intended aims, it leaves plenty of scope for future expansion. Although the visualisations generated are useful to the user, they could be improved through increased statistical analysis of the results of profiling. This might include the data plotting of profiling values, and analysis of the frequency of messages or events. In addition, the information of what messages are waiting within node message queues or the state which a node is within could be given.

When considering improvements to the back-end of the project, further work could be given to generating realistic network conditions. This could include the emulation of packet loss or variation in link latency. A particularly interesting addition would be the habitual pausing of node execution to simulate context switches and garbage collection at a node, thus providing a more realistic impression of how the algorithm would perform on an actual server.

10 Conclusions

10.1 Achievements

The development of this project has been a large achievement given the scale of the code base created and the thoroughness of the Test-Driven Development methodology employed. The project has created a highly usable framework that should aid any student who is unfamiliar with networked algorithms. Although some difficulties were encountered, these were all surpassed through rethinking of the design of the system and perseverance in developing a matching solution.

10.2 Experience Gained

10.2.1 Discrete Event Simulation

Initially unfamiliar with the core mechanics of discrete event simulation, this project has allowed me to gain a good understanding of its uses and construction. Initially appearing quite an abstract construct, building a discrete event simulator has given the impression that simulation has very concrete applications.

10.2.2 Java Concurrency Measures

Although already versed in the basics of Java concurrency control, the challenges that the scale of the multi-threading required to implement EasySim has forced me learn the details of these controls to a far greater degree. This even extended to the creation of thread containing structures, required to maintain thread-safe policies within code.

10.2.3 Prefuse Visualisation Framework

The prefuse visualisation framework has been an invaluable tool within the project and learning to make full use of the framework has been intriguing. Allowing a very wide range of functional applications, I am likely to consider prefuse highly in the development of further similar applications.

10.3 Personal Summary

Overall the project has been a great success. The program is stable, meets the requirements specified and is visually appealing. Unfortunately, I was unable to find extra time to add further functionality beyond the initial project aims. However, I have still enjoyed the development, which has provided many interesting challenges and has given me far better experience in building large scale systems.

11 Appendices

A Testing Summary

The entire project, with the exception of the code used to visualise and build networks is unit tested to ensure reliability and robustness. The unit of the *TestSimulator* also show its use as a testing framework. All of the tests have been passed, demonstrated in figure 11.1.

Following is summary of the tests used to gauge the projects satisfaction its defined requirements. Each test is mapped to a particular user requirement that it is intended to demonstrate within the produced system:

1. **Test:** the user interacts with the graphical network configuration tool to create complex graph, tree and ring topology networks.

   Expected: the structures specified by the user are stored within filenames of their choosing, matching the defined format. These topologies can be reloaded into the configuration editor after storage on the file system.

   Actual: the files were created as expected and when reloaded matched the forms that they were created to have. See *networks* folder for examples.

   Results: PASS

2. **Test:** the user specifies varying latencies of links in the graphical network configuration tool and writes them to file.

   Expected: the stored link latencies correspond to those shown when reread in the designer.

   Actual: the latencies seen in the designer match those in the loaded file exactly.

Result: PASS

3. **Test:** the user attempts to simulate the *RingLeaderScript* test class under conditions: 120 timesteps, timeout 1000ms.

   Expected: corresponding .json file containing appropriate events.

   Actual: created file contained appropriate events. See *results/RingExample.json*.

   Results: PASS

4. **Test:** the user attempts to visualise the *RingExample* at varying rates of playback, including reverse. User attempts interact with messages and nodes.

   Expected: The events of the results file are represented in the visualisation at an appropriate rate. Message information is given on selecting a message. Node profiling information given on selecting a node.

   Actual: an appropriate playback of events was give. Interaction with messages and node was as expected.

   Results: PASS

5. **Test:** the user simulates and visualises the *LoopScript* example on the network *GraphNet*.

   Expected: standard error shows the node timeouts and the visualisation displays the failure colour of the nodes as appropriate.

   Actual: error messages match to the expected behaviour. The visualisation represents the failures with appropriate colouring. See *examples/LoopTest.json*

   Results: PASS

6. **Test:** the user simulates and visualises the *BrokenRingLeaderScript* example on the network *RingNet*, alongside the existing *RingExample*.

   Expected: the visualisation shows the death of the initiator node in the broken example alongside the actual example, running at the same rate.

   Actual: the visualisation shows both examples as expected.

   Results: PASS

B Simulation metrics

The following metrics must be tracked per Node during simulation and be provided to the user on successful termination.

- Approximate CPU time spent, in clock cycles, in unblocked execution.
- Number of messages sent.
- Number of messages received.
- Number of messages read.

C Network Configuration Storage Format

The configuration - topology and properties - of a simulated network must be specified to match the following JSON Schema:

```
{
        "nodes": (node count)
        "edges": [ [(source ID), (target ID), (latency)] ]
        "deathEvents": {{"(node ID)": (timestep)}}
}
```

nodes A count of how many nodes are within the network.

edges An edge list. A source, target and message latency (in timesteps) are defined in that order for each item in *edges*.

deathEvents A list of the nodes that will die within simulation as a result of outside forces, mapped to the timestep that they will die.

This format has been designed as such to encourage the easy construction of the edge list by defining the topology of a network. As edges are defined with simply three integers, they can be expressed succinctly, allowing students to write simple topologies by hand.

D Simulation Results Storage Format

The results files generated from the simulation stage are stored in the following format:

```
{
        "events": [{"eventType":(String), "timestep":(Long), ... (
           event specific information)}]
        "messages": [{"arrivedAt":(Long), "data":(data sent), "tag
           ":(tag sent), "sourceId":(Integer), "targetId":(Integer)
           }]
        "network":(Network definition)
}
```

E User Manual

E.1 Executing EasySim

EasySim is packaged as a runnable jar and can be executed on any machine with a copy of Java 1.6 and access to the javac compiler. Simple run the code as such:

```
        java -jar EasySim
```

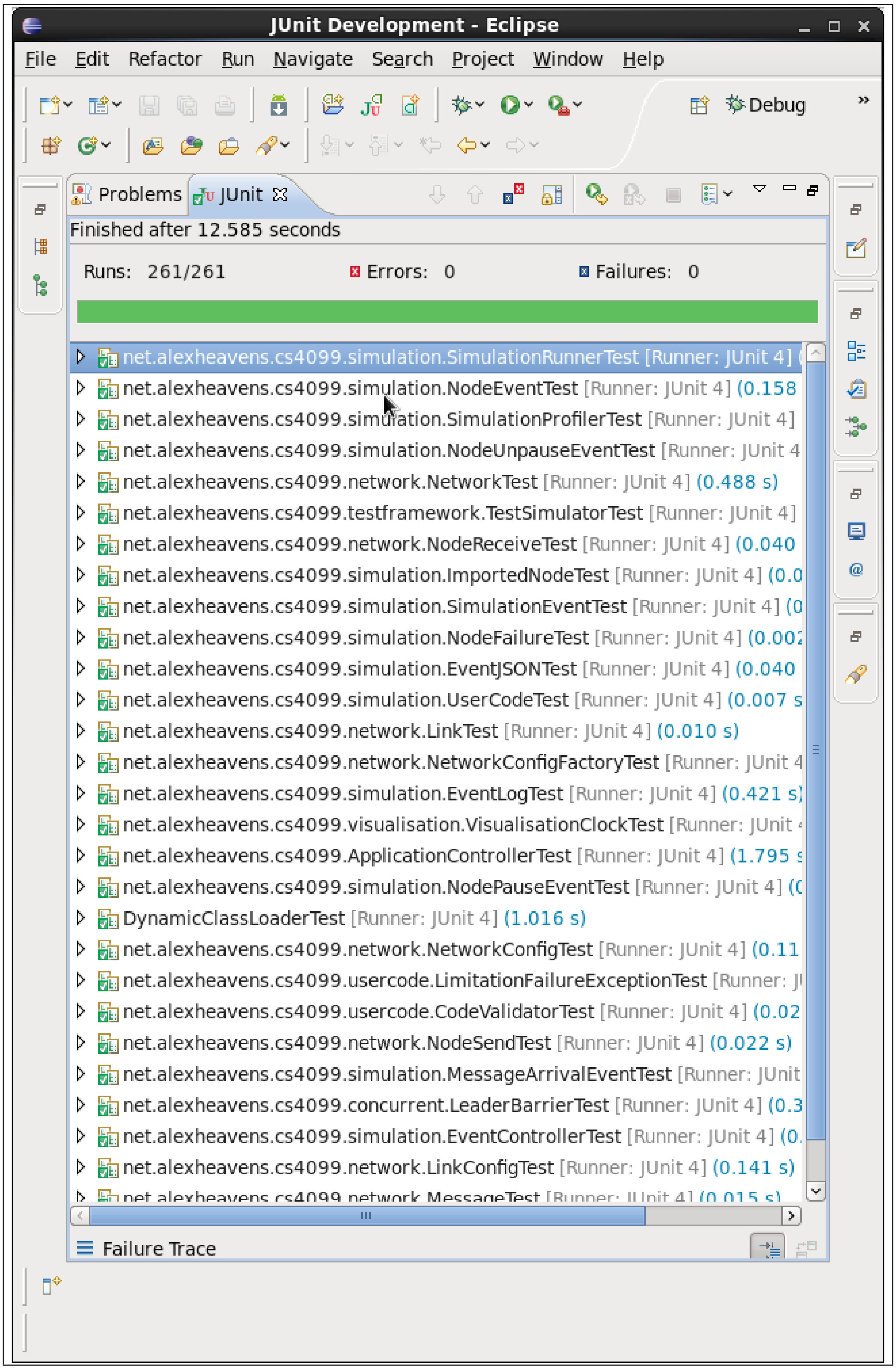
Examples of network configurations can be found in the *networks*, test user code examples can be found in *examples*, and sample results files can be found in *results*.

E.2 Viewing the source code

The EasySim source is contained within the Eclipse project *CS4099 SH - EasySim* and can be imported through:

```
        File > Import > Existing projects into workspace
```

Note that the provided *prefuse* project must also be imported to meet EasySim's build dependencies.


JUnit Development - Eclipse
File Edit Refactor Run Navigate Search Project Window Help
Debug
Problems
JUnit
Failure Trace

```
Finished after 12.585 seconds
Runs:  261/261          Errors:  0          Failures:  0

net.alexheavens.cs4099.simulation.SimulationRunnerTest [Runner: JUnit 4] (
net.alexheavens.cs4099.simulation.NodeEventTest [Runner: JUnit 4] (0.158
net.alexheavens.cs4099.simulation.SimulationProfilerTest [Runner: JUnit 4]
net.alexheavens.cs4099.simulation.NodeUnpauseEventTest [Runner: JUnit 4
net.alexheavens.cs4099.network.NetworkTest [Runner: JUnit 4] (0.488 s)
net.alexheavens.cs4099.testframework.TestSimulatorTest [Runner: JUnit 4]
net.alexheavens.cs4099.network.NodeReceiveTest [Runner: JUnit 4] (0.040
net.alexheavens.cs4099.simulation.ImportedNodeTest [Runner: JUnit 4] (0.0
net.alexheavens.cs4099.simulation.SimulationEventTest [Runner: JUnit 4] (0
net.alexheavens.cs4099.simulation.NodeFailureTest [Runner: JUnit 4] (0.002
net.alexheavens.cs4099.simulation.EventJSONTest [Runner: JUnit 4] (0.040
net.alexheavens.cs4099.simulation.UserCodeTest [Runner: JUnit 4] (0.007 s
net.alexheavens.cs4099.network.LinkTest [Runner: JUnit 4] (0.010 s)
net.alexheavens.cs4099.network.NetworkConfigFactoryTest [Runner: JUnit 4
net.alexheavens.cs4099.simulation.EventLogTest [Runner: JUnit 4] (0.421 s)
net.alexheavens.cs4099.visualisation.VisualisationClockTest [Runner: JUnit
net.alexheavens.cs4099.ApplicationControllerTest [Runner: JUnit 4] (1.795 s
net.alexheavens.cs4099.simulation.NodePauseEventTest [Runner: JUnit 4] (0
DynamicClassLoaderTest [Runner: JUnit 4] (1.016 s)
net.alexheavens.cs4099.network.NetworkConfigTest [Runner: JUnit 4] (0.11
net.alexheavens.cs4099.usercode.LimitationFailureExceptionTest [Runner: J
net.alexheavens.cs4099.usercode.CodeValidatorTest [Runner: JUnit 4] (0.02
net.alexheavens.cs4099.network.NodeSendTest [Runner: JUnit 4] (0.022 s)
net.alexheavens.cs4099.simulation.MessageArrivalEventTest [Runner: JUnit
net.alexheavens.cs4099.concurrent.LeaderBarrierTest [Runner: JUnit 4] (0.3
net.alexheavens.cs4099.simulation.EventControllerTest [Runner: JUnit 4] (0.
net.alexheavens.cs4099.network.LinkConfigTest [Runner: JUnit 4] (0.141 s)
net.alexheavens.cs4099.network.MessageTest [Runner: JUnit 4] (0.015 s)
```

Figure 11.1: Results from unit tests.

Bibliography

[1] A. Almiray. *Welcome to Json-lib*. Dec. 2010. URL: `http://json-lib.sourceforge.net/`.

[2] Cisco. *Cisco Packet Tracer*. Mar. 2012. URL: `http://www.cisco.com/web/learning/netacad/course_catalog/PacketTracer.html`.

[3] World Wide Web Consortium. *Extensible Markup Language (XML)*. Jan. 2012. URL: `http://www.w3.org/XML/`.

[4] D. Crockford. *JSON*. Mar. 2012. URL: `http://json.org/`.

[5] C. Evans. *The Official YAML Web Site*. Mar. 2012. URL: `http://yaml.org/`.

[6] B. Fry and C. Reas. *Overview \Processing.org*. Mar. 2012. URL: `http://processing.org/about/`.

[7] EDC Group et al. *Sinalgo-simulator for network algorithms*. 2008.

[8] J. Heer. *visulisation gallery*. Aug. 2007. URL: `http://prefuse.org/gallery/`.

[9] J. Heer, S.K. Card and J.A. Landay. “Prefuse: a toolkit for interactive information visualization”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 2005, pp. 421–430.

[10] T.R. Henderson et al. “Network simulations with the ns-3 simulator”. In: *SIGCOMM demonstration* (2008).

[11] IBM. *Teleprocessing Network Simulator*. Mar. 2012. URL: `http://www-01.ibm.com/software/network/tpns/`.

[12] J. Janitor, F. Jakab and K. Kniewald. “Visual Learning Tools for Teaching/Learning Computer Networks: Cisco Networking Academy and Packet Tracer”. In: *Networking and Services (ICNS), 2010 Sixth International Conference on*. IEEE. 2010, pp. 351–355.

[13] A.M. Law and W.D. Kelton. *Simulation modeling and analysis*. McGraw-Hill series in industrial engineering and management science. McGraw-Hill, 2000. ISBN: 9780070592926.

[14] Loskutov A. Pugh B. *Findbugs*. Dec. 2011.

[15] Z.A. Qun and W. Jun. "Application of NS2 in Education of Computer Networks". In: *Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on*. IEEE. 2008, pp. 368–372.

[16] A. Smith and C. Bluck. "Multiuser collaborative practical learning using packet tracer". In: *Networking and Services (ICNS), 2010 Sixth International Conference on*. IEEE. 2010, pp. 356–362.

[17] Sun. *Cyclic Barrier*. [Accessed 21/02/2012]. Dec. 2006. URL: `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/CyclicBarrier.html`.