University of Western Ontario

CS9668: Internet Algorithmics

Final Project Report

Simulator of an Asynchronous Distributed System

Author:
Daniel Servos
dservos5@uwo.ca
#250722713

Instructor:
Dr. Roberto Solis-Oba

April 28th, 2015

Abstract

This paper gives an overview of the creation and implementation of a new network simulator targeted at educational use. This simulator, entitled DANS (Dan's Asynchronous Network Simulator), uses a process-based approach to emulating basic network components (processors, links, etc.) while also providing a means of producing deterministic results. A versatile but easy to use GUI is provided in addition to a number of example algorithms and documentation describing their creation. Both the simulator's architecture and implementation are reviewed in detail and areas for future work are identified.

1 Introduction

In research, areas relating to computer networks and distributed systems it is often desired that new algorithms and methods provide detailed analysis of their performance and correctness in a realistic environment. However, the creation and maintenance of such hardware environments is often costly both in terms of time and the monetary expenses required. A common alternative to such hardware based test beds is the use of software based simulation methods to create virtual test beds that approximate the variables and conditions founds in their real hardware based counterparts. Outside the realm of research, network simulations provide tools for prototyping, developing and visualization protocols and algorithms that are helpful for both developers and students wishing to gain further insight into the inner workings of a distributed system.

A common approach to network simulation is to utilize discrete event simulation. In the discrete event method all possible changes to a systems state are considered to be part of events that occur at a specified instant of time. All pending future events that have not yet occurred are stored in an event set (commonly implemented as a priority queue) and accessed and applied to the network state at the time indicated in the event. Using this method it is possible to jump directly from the end of one event to the start of another without necessarily having to simulate the time elapsed between the two events (potentially reducing the time required to run lengthy simulations). For example if the transmission of some packet occurs at time T and arrives at its destination at time $T + \Delta T$, it is not necessary to simulate the time, ΔT, that it takes the packet to be transmitted assuming no other relevant state changes take place during that time.

An alternative approach, and the one utilized in this paper, is the process-based method. In the process-based method each activity or actor is modelled by an individual process or thread. Events created by processes both trigger changes to the system's state and invoke actions by other processes. For example, in a network simulation each processor/node might be modelled as a distinct thread and transmitting a packet would consist of one thread creating an event that both updates the network state and eventually wakes the thread corresponding to the receiving processor. This method can produce more modular and easier to understand code but introduces the complexities of concurrency and synchronization.

A large number of attempts have been made towards the development of network simulation tools but only a handful have gained widespread acceptance in both academic research and industry application. Perhaps the most popular of these is the ns line of simulators that started with the REAL (REalistic And Large) computer network simulator[1] upon which ns-1[2], ns-2[3] and ns-3[4] are based. These simulators use a discrete event model and aim to provide an open simulation environment for advancing networking research and education. ns-3, the most recent incarnation

of the simulator, supports both IP and non-IP based networks as well as wireless simulations (the primary focus of the majority of users). Other simulators include the Georgia Tech Network Simulator (GTNetS)[5], OPNET Modeler (now Riverbed Modeler)[6], OMNeT++[7], and NetSim[8] all of which use a discrete event model. GTNetS and OMNeT++ provide open solutions aimed at research applications while OPNET and NetSim are commercial solutions that are aimed towards prototyping, planning and network development.

While these efforts are perhaps adequate for research and industry application they are lacking for the domain of education, particularly for the studying of distributed algorithms. The added complexity required to realistically emulate the upper OSI layers and the physical structure of real networks tends to require users to commit a significant investment in both time and effort into learning the basic functions of the simulator despite only requiring a small subset of the features offered. The work described in this paper aspires to create a simplistic and largely GUI driven simulator for visualizing asynchronous distributed algorithms that is just sufficiently complex to support the most common algorithms that would be studied in an educational setting. Realistic modelling of the underlying OSI layers and physical network structure is sacrificed in favour of a more straightforward and high level representation.

The implemented simulator, referred to as DANS (Dan's Asynchronous Network Simulator), models a given network as a set of processors, links and a collection of global settings describing how the simulation will be conducted. Algorithms running on each processor are able to trigger send events that both affect the networks state and trigger subsequent events in receiving processors. A process-based method is used in which each algorithm on each processor is modelled as a thread that is awakened periodically (either by being triggered by a send event or after a set period of time) to update the network state and send/receive any messages left in its message queue. A versatile but easy to use GUI is provided to allow network editing and live visualization of current simulations.

The remainder of this paper is divided into the following sections; Section 2 details the specifications and objectives of the implemented network simulator, Section 3 gives a high level overview of the simulator's architecture and explains how communication between each component is accomplished, Section 4 discusses the tools and libraries used in implementation as well as how users may create algorithms for use in the simulator, and finally Section 5 provides concluding remarks and directions for future work. Additional documentation and usage instructions for the simulator can be found in the appendices.

2 Objectives & Specification

The main objective of DANS is to provide a simple and easy to use network simulator that is just sufficient in capabilities to accurately simulate the majority of the distributed algorithms studied in the CS9668: Internet Algorithmics course[1] taught at the University of Western Ontario. This includes algorithms that cover leader election, broadcasting, building search trees, finding a shortest path, consensus[2], and Chord[10]. The proceeding subsections detail the requirements and specifications given for the project as well as any additional features added.

[1] `http://www.csd.uwo.ca/faculty/solis/cs868b/2014/index.html`

[2] Not all consensus problems are solvable in an asynchronous environment[9].

2.1 Simulation Specification

Table 1 outlines the requirements for the simulator back-end (i.e. all components of the simulator other than the user interface). Requirements starting with "SIM_P" are primary requirements given in the projects description, while requirements starting with "SIM_E" are extra requirements added in addition to those given the project description.

Table 1: Simulation Requirements

Req#	**Requirement**	**Description**
SIM_P1	Input	The simulator should take as input a set of parameters including the number of processors, set of neighbours for each processor, the algorithm A that the processors must execute, speed of each processor, delay of each communication link, probability that each processor will fail, probability that each link will fail, whether Bizantine failures are allowed, etc.
SIM_P2	Processor speed & link delay	Each processor and link should have a mutable speed or delay setting that controls the speed at which the processor completes cycles of its main loop or the speed at which a link can transmit messages.
SIM_P3	Failure probability	Each link and processor should have a mutable failure probability setting that determines the likelihood of a processor or link failing during the execution of the simulation.
SIM_P4	Bizantine failures	Both clean and Bizantine failures should be supported for processors.
SIM_P5	Execution	The system must execute the specified distributed algorithm on each processor, taking care of delivering all messages that the processors exchange among themselves.
SIM_P6	Speed & delay change	There should be support for the speed and/or delay of the processors and links to change over time during the execution of the algorithm.
SIM_P7	Algorithm support	Simulator should support a variety of simple algorithms, similar to those discussed in CS9668.
SIM_E1	Link bandwidth	Each link should support a mutable bandwidth setting that controls the number of messages a link can transmit over a given period of time.
SIM_E2	Byte errors	Each link should should have a mutable byte error probability that determines if a given byte will contain an error (i.e. the byte will be changed to a different value when received).
SIM_E3	Different algorithms	Each processor should be able to run a different algorithm if desired. This would enable simulations that involve different algorithms communicating with each other (e.g. client/sever type situations).

Continued on next page

Table 1 – continued from previous page

Req#	Requirement	Description
SIM_E4	Multiple algorithms on one processor	Each processor should support running multiple algorithms simultaneously that have access to the same shared memory object. Algorithms running on different processors should not have access to the shared memory object of remote processors.
SIM_E5	Ports	Each algorithm running on a processor should be set to listen on a given port and be able to send messages to any remote port. Messages transmitted to a processor are only delivered to an algorithm listening on the remote port specified in the message.
SIM_E6	Pause, unpause, restart simulation	The simulation should be pauseable and restartable without having to restart the simulator application.
SIM_E7	Simulation statistics	The simulator should log and track basic statistics about the simulation including number of messages sent and number of cycles each processor has executed.
SIM_E8	Directional links	Support for both directional and bidirectional links between processors.
SIM_E9	Deterministic simulation	Given the same settings and initial network state, the simulator should return the same results (assuming the given algorithms are deterministic).

2.2 GUI Specification

Table 2 outlines the requirements for the GUI front-end (i.e. all components that make up the user interface). Requirements starting with "GUI_P" are primary requirements given in the projects description, while requirements starting with "GUI_E" are extra requirements added in addition to those given the project description.

Table 2: GUI Requirements

Req#	Requirement	Description
GUI_P1	Graphical interface	A graphical user interface is required for visualizing, controlling and setting up the simulation.
GUI_P2	Usability	The graphical interface and the simulator as a whole should be easy to use.
GUI_E1	Network editor	The graphical interface should support the creation and editing of simulated networks. The user should be able to add/remove processors and connect them with links.

Continued on next page

Table 2 – continued from previous page

Req#	Requirement	Description
GUI_E2	Runtime algorithm loading/switching	The user should be able to select and/or change which algorithm is being run on each processor at runtime.
GUI_E3	Common editing tools	The interface should support common editing tools and functions including copy, paste, cut, undo, redo, select all, select none, etc.
GUI_E4	Logging	The simulator should support logging to the command line, graphical interface, and to a file. Different logging levels should be available to control the logs verbosity.
GUI_E5	Zoom	The interface should support zooming in and out on the network graph.
GUI_E6	Full screen	The interface should have full screen support for presentations.
GUI_E7	Export network graph	The simulator should support saving and printing the network graph visualization as an image.

3 Architecture & Design

The DANS architecture is divided into two logical parts. A front-end consisting of all user interface components including the GUI, keyboard input and logging system. And a back-end consisting of components involved in the actual representation and simulation of a given network and algorithm. The back-end is designed to be completely independent of the front-end such that the front-end could be replaced without requiring changes to the back-end (e.g. the GUI could be replaced with a console based interface). The following subsections detail the architecture of both the front and back-end components in addition to describing the "Tick" system introduced to produced deterministic results.

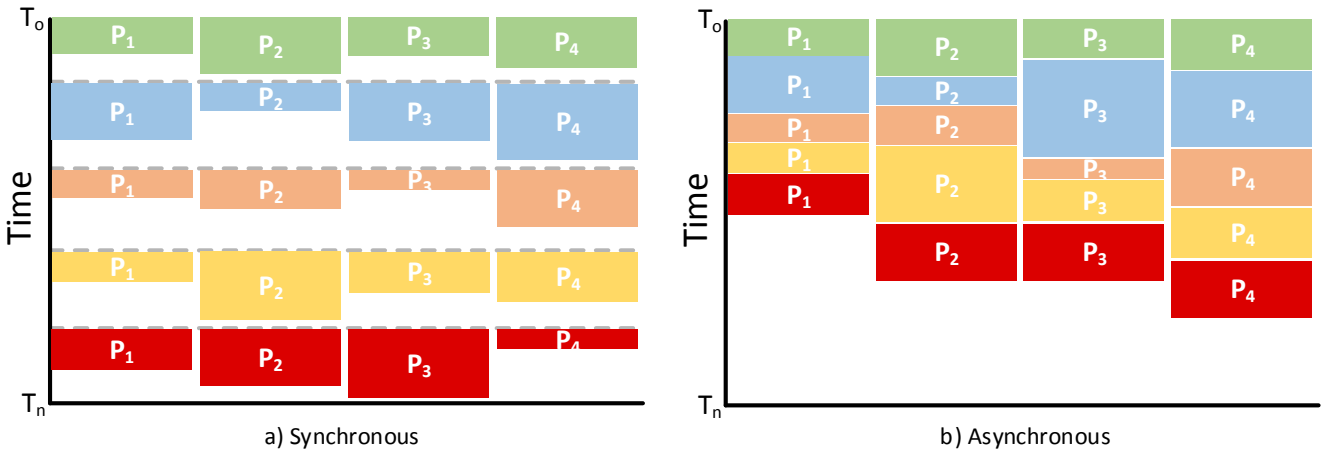


Figure 1: Synchronous v.s. asynchronous models of distributed algorithms.

3.1 The Tick System

Traditionally, distributed algorithms assume either a synchronous or asynchronous model. In the synchronous model (shown in Figure 1.a), it is assumed that each processor pauses after executing a cycle or "round" of the distributed algorithm and waits for all other processors to get to the same point before continuing. Furthermore, the distributed algorithm is broken up into two phases, a message sending phase and a message receiving phase that must occur in the same order on each processor. This ensures that all messages are sent and received in a given round and that all processors are executing the same round at the same time. In this way the execution time of an algorithm can be discussed in terms of rounds (e.g. how many round does an algorithm take to terminate or what is the state of the network after i rounds).

In the asynchronous model (shown in Figure 1.b), no assumptions are made as to time each processor takes to complete a cycle of the algorithm and no restrictions are placed on when the algorithm can send or receive messages. In this way each processor immediately starts the next cycle of its algorithm after completing the last. No guarantee is given that each processor may be executing the same cycle of the algorithm or that any message will be sent or received in the same (or any) cycle. In terms of simulation this model can be problematic in that the current state of the network at a given time T_i may be different for each execution even if the same initial network state and settings are used (as shown in Figure 2). For example, if a given algorithm is executed on 4 processors and each cycle completes at the times shown in Figure 2.a a second execution of the same algorithm with the same settings might have different results (in terms of when cycles start and end) if an individual processor is delayed for some reason as shown in Figure 2.b. This could happen for a number of reasons including a thread waiting for a resource to unlock or CPU time being delegated differently on subsequent executions.

To meet the requirement that the results of the simulation be deterministic (requirement SIM_E9) a new "Tick" system is introduced. In this system, time is counted in "ticks" rather than rounds or traditional units of time. A tick is defined as a variable length of time that is sufficiently large enough that any algorithm being simulated can complete a single cycle. Each processor is then configured to take a set number of ticks to complete one cycle of the algorithm and each link is configured to take a set number of ticks to transmit a message. To allow the varying speeds and delays found in the asynchronous model a pseudorandomly determined amount of

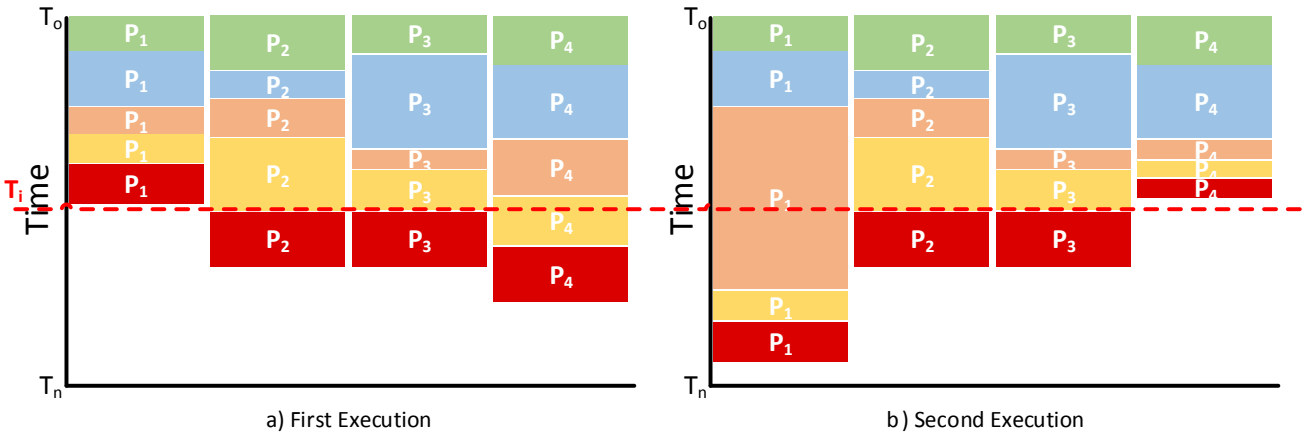


Figure 2: Nondeterministic nature of asynchronous simulation. Both a) and b) are execution of the same algorithm that result in different system states at time T_i.

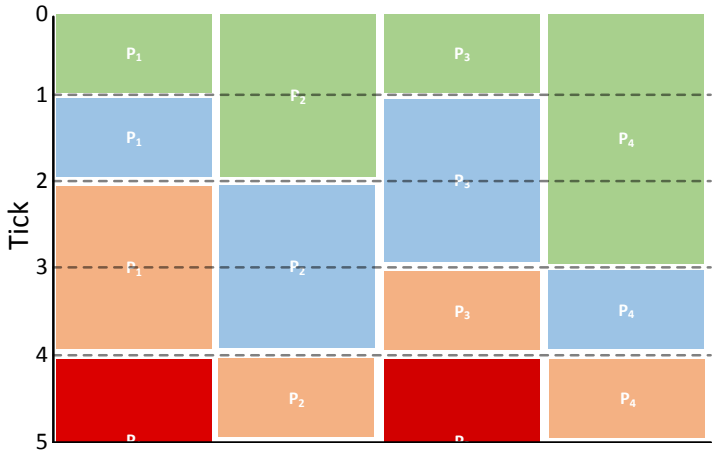


Figure 3: Algorithm execution using the Tick system. Only ticks 0 to 5 shown.

change in the speed/delay of each processor/link is allowed. The result remains deterministic as the seed for the pseudorandom function is provided by the user with the initial settings for the simulation such that the same changes are applied at the same times for the same seed. This style of execution is displayed in Figure 3.

DANS supports synchronous[3], asynchronous[4] and tick based execution. Users are allowed to specify a minimum tick length such that the length of a tick is:

Max = Maximum time required to complete one cycle of any algorithm being simulated.
Min = User define minimum tick speed
Tick Length = Min if Min > Max else Max

3.2 Simulation Back-End

The main elements of the back-end simulation are the processor, link, message and algorithm objects. Each of these objects is contained in a single network state object and coordinated by a single network manager thread. Each algorithm on each processor is contained in its own thread that is responsible for executing that algorithm. For example if two algorithms are ran on four processors eight threads would be used, one per algorithm per processor. Links are controlled by the network manager thread which periodically updates each link's state and ensures its messages are sent to the correct processor at the correct time. The following subsections detail the design behind each of these components.

3.2.1 Processors

Processors represent the distributed systems the algorithms will be executed upon and nodes in the network graph. The processor object (shown in Figure 4.b) contains a list of settings and statistics stored in a concurrent hash map[5] indexed by the name of the setting or statistic (i.e. the name is the key of the entry in the hash map). Settings determine how the processor will function in the simulation (possible settings are described in Table 7 in Appendix B) and statistics keep track of various figures related to the execution of the algorithms on the processor during the

[3]If all processors are given a delay of 1 and links a delay of 0.
[4]If all processors are given a delay of 0.
[5]`https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html`

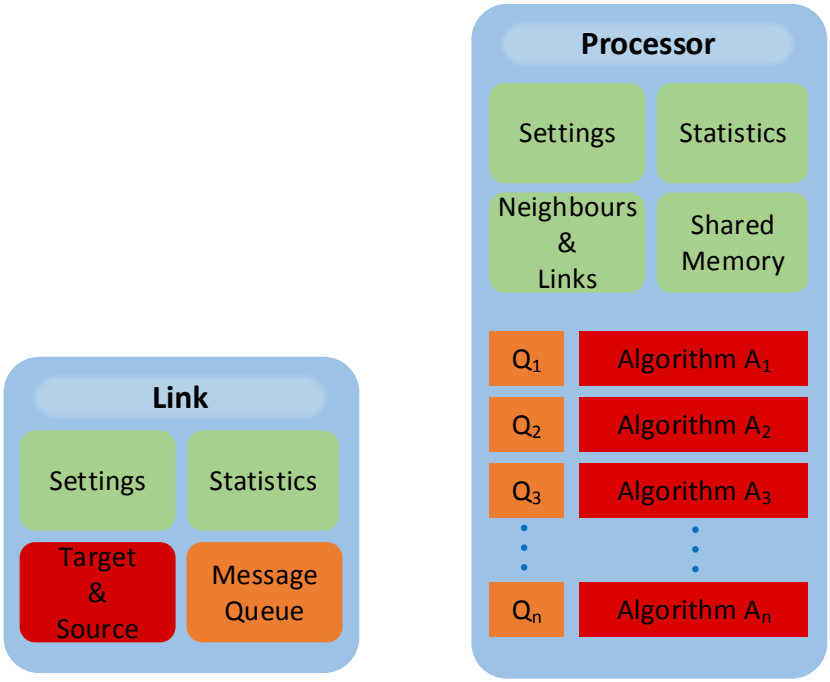


Figure 4: Processor and Link objects. Green boxes denote a collection (ConcurrentHashMap), red boxes denote a reference to another object, and orange boxes represent a queue object.

simulation (possible statistics are described in Table 5 in Appendix B). A list of the processors neighbours and links connected to the processor are also stored in a concurrent hash map.

Each processor is able to run 1 to n algorithms and contains n output queues and n algorithm object, one for each algorithm. Every algorithm registered to a processor must "listen" on a unique port number such that its queue only receives messages sent to its specific port. The output queue represents the message buffer for the given algorithm and messages are removed in order upon calling the receive command. Each output queue is a custom object but essentially provides functions that indirectly access a concurrent linked queue[6].

All algorithms running on a given processor have access to a shared memory object represented by a concurrent hash map. It is up to the algorithms to determine how the shared memory is used and the keys used to index any values stored in the map. Also, while the hash map its self is thread safe, it is up to the algorithm's implementation to ensure the values stored in the map are properly synchronized if they are not primitive types.

3.2.2 Links

Links represent directional connections between processors over which messages can be transmitted and the edges of the network graph. Like processors, the link object (as shown in Figure 4.a) uses a concurrent hash map to store settings and statistics about the individual link (a description of each possible setting and statistic is given in Tables 8 and 6 respectively in Appendix B). Links store a reference to their target and source processors as well as an input queue that contains messages being transmitted over the link. As with the output queues used by the processor object, a links input queue is a custom object that utilizes a concurrent linked queue to store messages.

[6] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html`

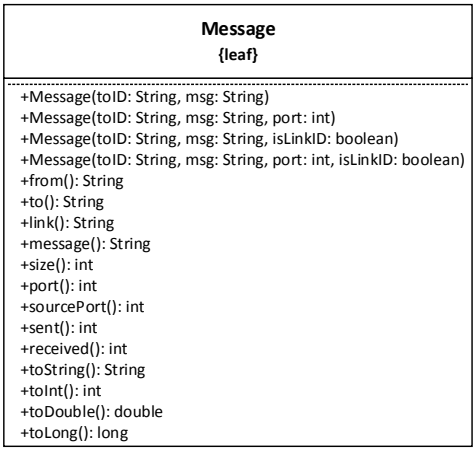


Figure 5: Message class UML diagram.

3.2.3 Messages

Messages are strings of variable length that are contained in a message object that includes additional meta data about the message including its destination, origin, source port, target port, size, etc. A UML class for the message class is given in Figure 5. Messages are sent by processors (on behalf of the algorithms they run) by adding them to the appropriate link which in turns tags the message with the time (in ticks) it was received and stores it in its output queue. Periodically (at least once per tick, but often more), the network manager will invoke the transfer method on each link object. The transfer method peeks at the top of the link's message output queue and compares the time the message was sent with the current time (in ticks) and determines if the message should be removed from the queue and sent to the appropriate processor (this determination is based on the delay and bandwidth of the link). If a message is removed, the transfer method is called again on the link until no more message can be sent. When a processor

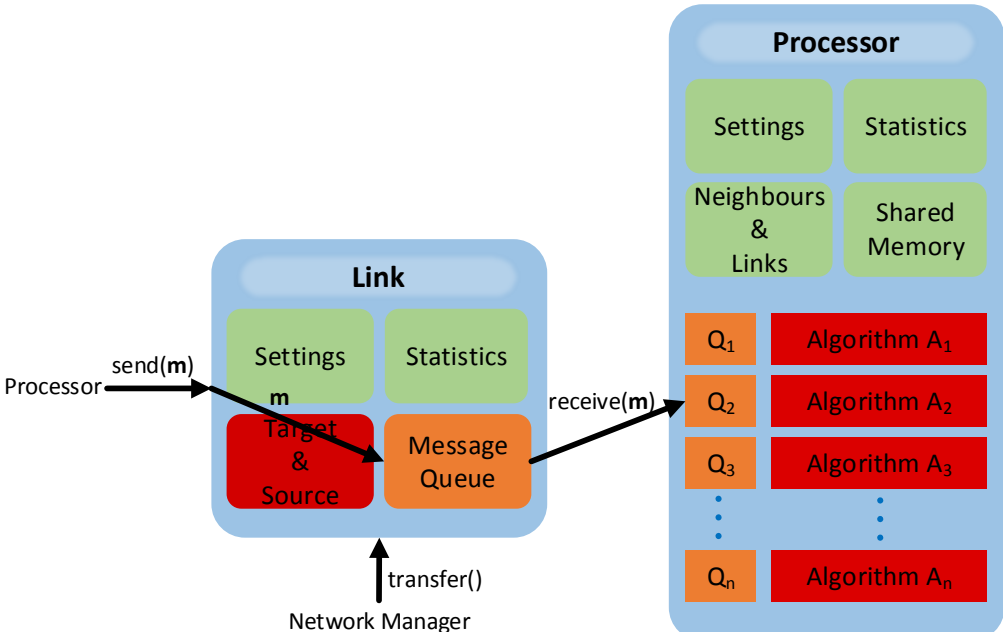


Figure 6: Process of sending some message, m, to the port algorithm, A_2, is listening on.

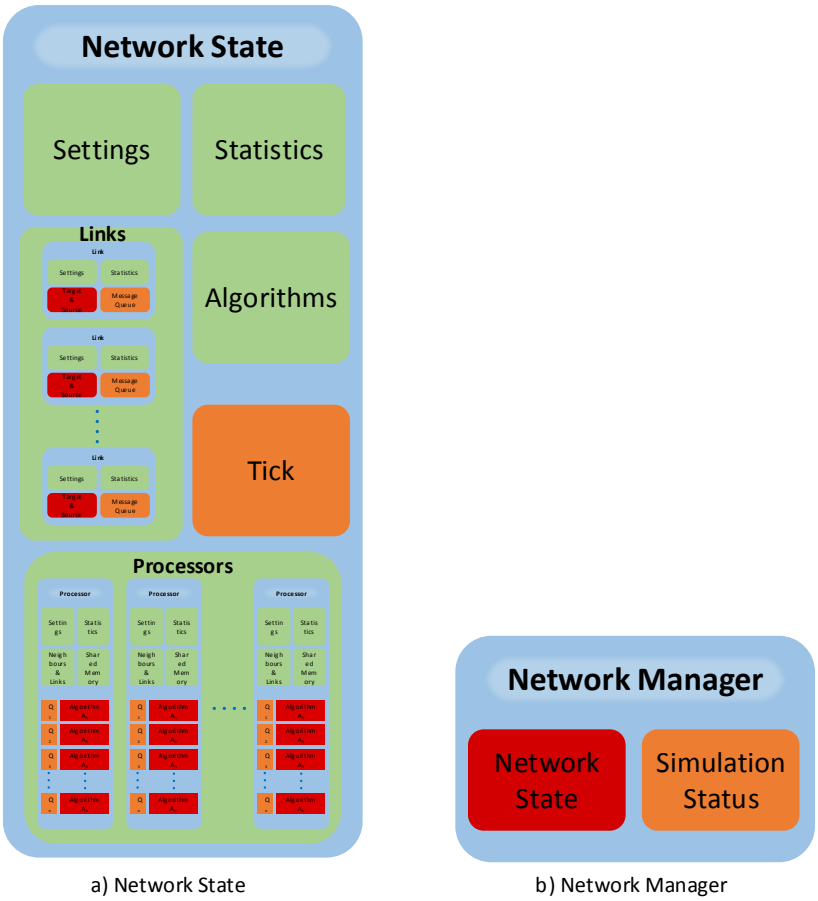
receives a message from a link it adds it to the queue for the algorithm matching the target port listed in the message. This process is shown in Figure 6.

3.2.4 Algorithms

Algorithms are user defined programs that have limited access to the simulator back-end and are restricted to only using methods defined in the algorithm class (detailed in Section 4.3). The algorithm class isolates algorithms to only accessing elements of the processor they are executed on, enforcing the distributed system model. All algortihms running on the same processor share the same settings, statistics and shared memory object. User defined algorithms should contain an event loop that receives/sends messages and composes the main logic of the algorithm. One cycle of this loop will take a specified number of ticks (as defined in the processor's settings) with a minimum of one cycle per tick.

3.2.5 Network State

The network state maintains the current state of the simulation at a given point in time. It contains all elements of the simulation and their individual settings, statistics, and states. The network state object (shown in Figure 7.a) uses a concurrent hash map to store settings, statistics,



a) Network State b) Network Manager

Figure 7: Network state and network manager objects. Green boxes denote collections (ConcurrentHashMap), red boxes denote a reference to another object, and orange boxes denote a primitive type or built in Java object.

the set of processors, the set of links and the set of algorithms used in the current simulation. The settings and statistic collections contain global properties and defaults that effect or describe the whole network being simulated (and are described in detail in Appendix B in Tables 4 and 9). The links and processor collections are indexed by the unique identifier assigned to each link and processor allowing for efficient lookups. The algorithms collection contains the set of user created algorithm classes that have been loaded at run time, indexed by the name of the class. Finally a primitive integer type is used to keep track of the current tick. In addition to the concurrency protections offered by the ConcurrentHashMap type, locks are used to ensure that all operations on the network state are thread safe.

3.2.6 Network Manager

The network manager (shown in Figure 7.b) is the main thread of the simulator and is responsible for creating, waking and terminating the algorithm threads when appropriate. It contains a reference to the network state and the current simulation status (e.g. running, stopped, paused, etc.). Additionally, it is responsible for incrementing the tick value in the network state and periodically calling the transfer method on each link to ensure messages are delivered to the algorithms' input queues at the correct time (account for the links bandwidth and delay).

3.3 GUI Front-End

The GUI front-end consists of a GUI object and handlers for the keyboard and user interface. The GUI object (shown in Figure 8.a) is responsible for configuring and initializing the graphical interface and contains the Java swing GUI components and the JGraph graph visualization/editor. The GUI Handler (shown in Figure 8.b) deals with all logic regarding events caused by a user's interactions with the graphical interface and handles all communication with the simulator back-end. Similarly, the Keyboard Handler deals with all keyboard based input and translates keyboard short cuts into GUI actions that the GUI handler can deal with.

All GUI events and updates are handled by special event dispatch thread (EDT) that is solely allowed to change the state of the GUI. This is both due to the built in swing GUI components not being thread safe and to maintain the separation between the back-end and front-end. As such no back-end thread is allowed to directly edit, update or otherwise alter any of the GUI components. Instead the network state, processors and links implement an observer software pattern that allows any object implementing the correct interface to register its self and be automatically informed

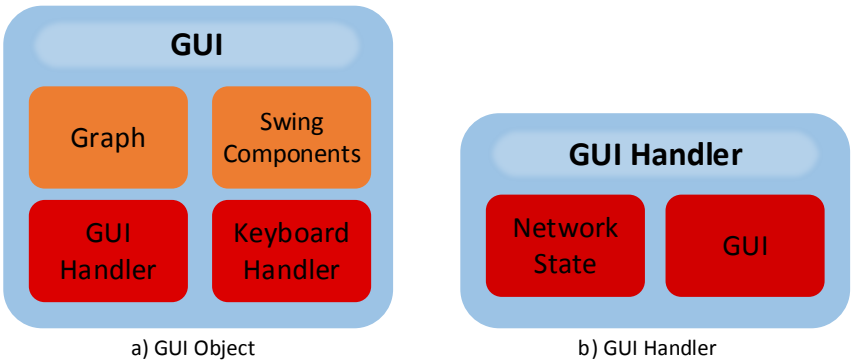


Figure 8: GUI object and GUI Handler. Orange boxes represent swing based GUI components and red boxes represent references to other custom objects.

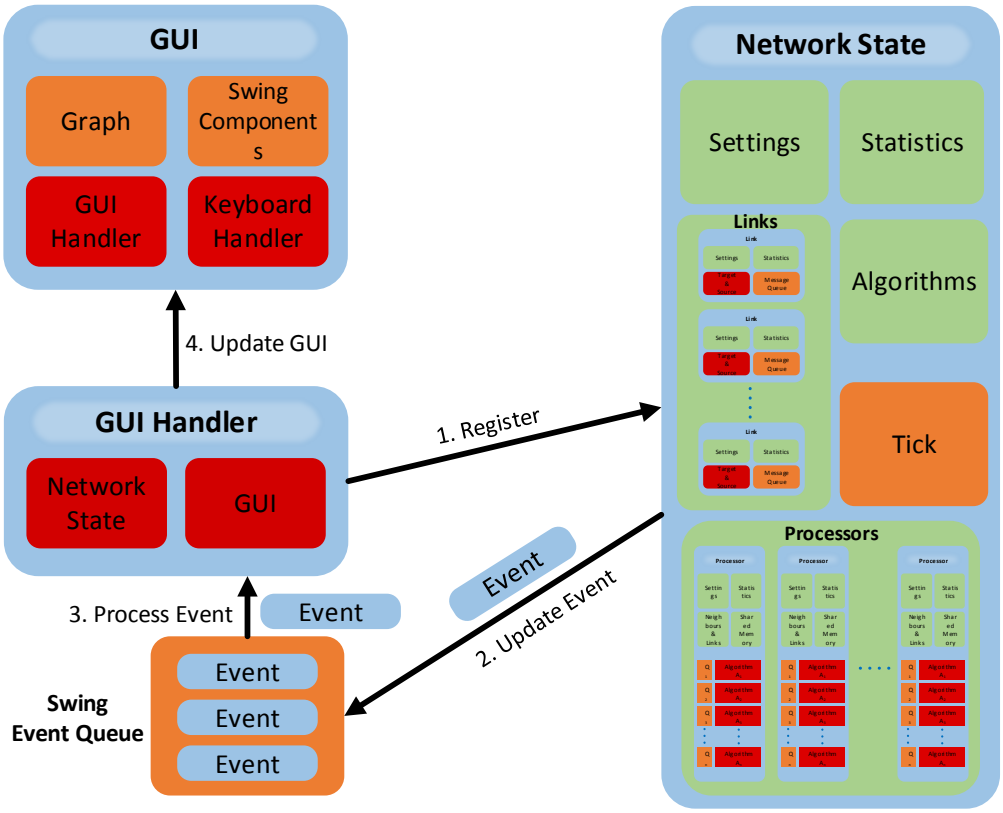


Figure 9: Process by which network state update events are sent from the back-end to the front-end GUI.

of any changes to the network state. In the case of the GUI Handler update events from the network state are added to the swing event queue (via the SwingUtilities.invokeLater method) and processed in a FIFO order in the EDT. This process is shown in Figure 9 and allows both the back-end to require no knowledge of the front-end and avoids any currency issues as all GUI updates are handled in a single thread.

As the network state object is fully synchronized and thread safe, the GUI handler is able make direct calls from the EDT to update settings and values in the network state so long as these calls do not block or delay the execution of the EDT. For example, the GUI handler could remove a processor from the network state on a mouse click, but it could not wait or do a loop until some property of the network state has changed as this would block the EDT and cause the GUI to become unresponsive.

4 Implementation Details

The following subsections give additional details about the implementation of the simulator including the tools used, description and properties of the codebase, and further details about the implementation of user defined algorithms.

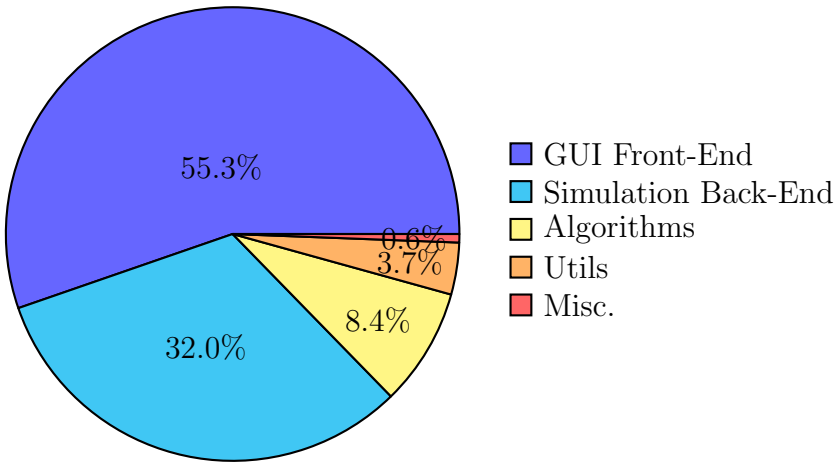


Figure 10: Physical executable lines of code per component.

4.1 Tools & Libraries

The simulator back and front-ends are implemented in Java 8 and is not backwards compatible with older versions of Java. The GUI front-end uses the built in swing widget toolkit for creating GUI elements and utilizes the JGraphX[7] library for graph visualization and editing. Several classes from the JGraphX library are extended to provide additional features or changes required for the simulator. Extended classes include mxCell, mxConnectPreview, mxGraph and mxGraphLayout. Finally, toolbar and menu icons from the Java Look-and-Feel Graphics Repository (JLFGR)[8] library are also used in the GUI.

4.2 Codebase

The simulator codebase is divided into four parts, the GUI front-end (package *dans.GUI*), the simulation back-end (package *dans.network*), utilities (package *dans.util*) and algorithms (package *dans.algorithm*). Packages *dans.GUI* and *dans.network* contain the front and back-end components as described in the previous sections. Package *dans.util* contains utility classes that provide simple services for the other packages (including a logging service). Package *dans.algorithm* contains the abstract algorithm class for users to extend (as described in Section 4.3) as well as several

[7] `https://github.com/jgraph/jgraphx`
[8] `http://www.oracle.com/technetwork/java/index-138612.html`

Table 3: Codebase Metrics

Metric	**Count**
Source Files	54
Directories	8
Lines of Code	10715
Blank Lines of Code	1635
Physical Executable Lines of Code	7933
Logical Executable Lines of Code	6253

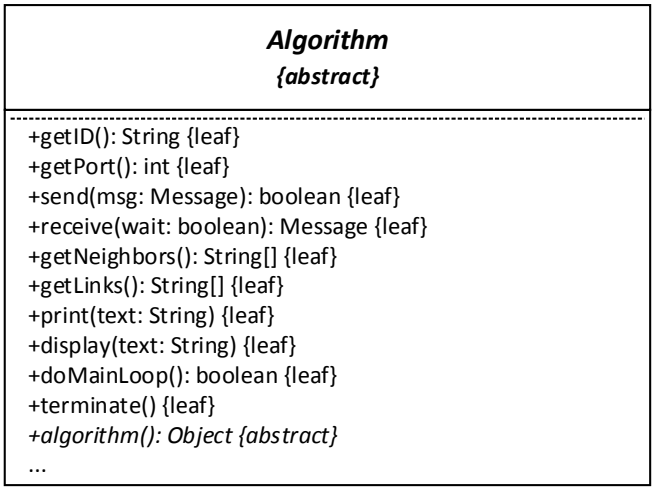


Figure 11: UML class diagram for abstract algorithm class. Some methods omitted for space reasons.

example algorithms. A break down of the number of executable lines of code per division is given in Figure 10 and overall counts are given in Table 3.

4.3 Algorithm Creation

As described in Section 3.2.4 users may create algorithms by extending the abstract Algorithm class (UML diagram shown in Figure 11). This class provides methods that grant limited access to the properties of the processor the algorithm is executed on as well as the ability to send and receive messages. All user created algorithms should follow the template given in Listing 1, such that they contain a single event loop in the *algorithm* method that handles the sending and receiving of all messages. This loop should run until the value of *doMainLoop()* returns false or the algorithm has finished running (it is acceptable to break or return out of the loop if the algorithm should terminate). The *algorithm* method may return a single value that represents the result of

Listing 1: Algorithm Template

```
1  import dans.algorithm.Algorithm;
2  import dans.algorithm.Message;
3
4  public class MyAlgorithm extends Algorithm {
5
6       @Override
7       public Object algorithm() {
8            //Do algorithm setup here
9
10           while(doMainLoop())  {
11                //Main algorithm code
12                //Break, return, or call terminate() when done
13           }
14
15           //Code to run before termination
16           return MyResults;
17      }
18 }
```

the execution which is displayed when the processor terminates.

A *receive* method is provided that will either immediately return the next message in the input queue (or null) or wait for the next message to arrive depending on the arguments given. If the algorithm waits for a message it will not block the execution of any other algorithm (i.e. the other algorithms running will not wait for it to finish its cycle before continuing and the tick value will be incremented normally). The *send* method returns a boolean value indicating if the message was added to the links output queue successfully and will return immediately in either case (i.e. the send method does not wait for the message to be delivered).

Complete documentation of the Algorithm and Message classes are given in the JavaDoc found at `http://cs1.ca/cs9668/async`. Example algorithms for leader election, broadcasting, and doing simple calculations using a BFS tree are given in Appendix C.

5 Conclusion

DANS provides a simple and easy to use network simulator for visualizing and prototyping basic distributed algorithms that is aimed at educational use. While both traditional synchronous and asynchronous modes of operation are supported (by configuring the delays in a certain way as described in Section 3.1) a new Tick based system is introduced that allows for deterministic results while maintaining most properties of an asynchronous simulation. A detailed description of the DANS architecture and design is given as well as details about the Java based implementation. Additional documentation for the simulator can be found in the Appendixes (including example algorithms) and complete documentation of the Algorithm and Message classes (the two classes users would interact with) can be found at `http://cs1.ca/cs9668/async`.

There are a number of directions for future work beyond simply fixing the known bugs and limitations listed in Appendix D. Some possible features include adding support for wireless networks, increasing the realism of the simulation (e.g. more closely modelling the OSI layers), adding more network editing tools, adding additional simulation statistics, allowing for editing while the simulation is paused, adding addtional documentation and examples, automatic generation of different kinds of networks, automatic randomization of processor and link settings and support for automatic network graph layouts.

References

[1] Srinivasan Keshav. *REAL: A network simulator*. University of California, 1988.

[2] Steven McCanne and Sally Floyd. Ns network simulator, 1995.

[3] Network Simulator. ns-2. `http://www.isi.edu/nsnam/ns/`, 1989.

[4] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and JB Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 15:17, 2008.

[5] George F Riley. The georgia tech network simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12. ACM, 2003.

[6] Riverbed Technology. Riverbed modeler. `http://www.riverbed.com/products/performance-management-control/network-performance-management/network-simulation.html`.

[7] András Varga et al. The omnet++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESM2001)*, volume 9, page 65. sn, 2001.

[8] Tetcos. Netsim. `http://tetcos.com/`, 2002.

[9] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[10] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

Appendix

Appendix A: Simulator Manual

Running the Simulator

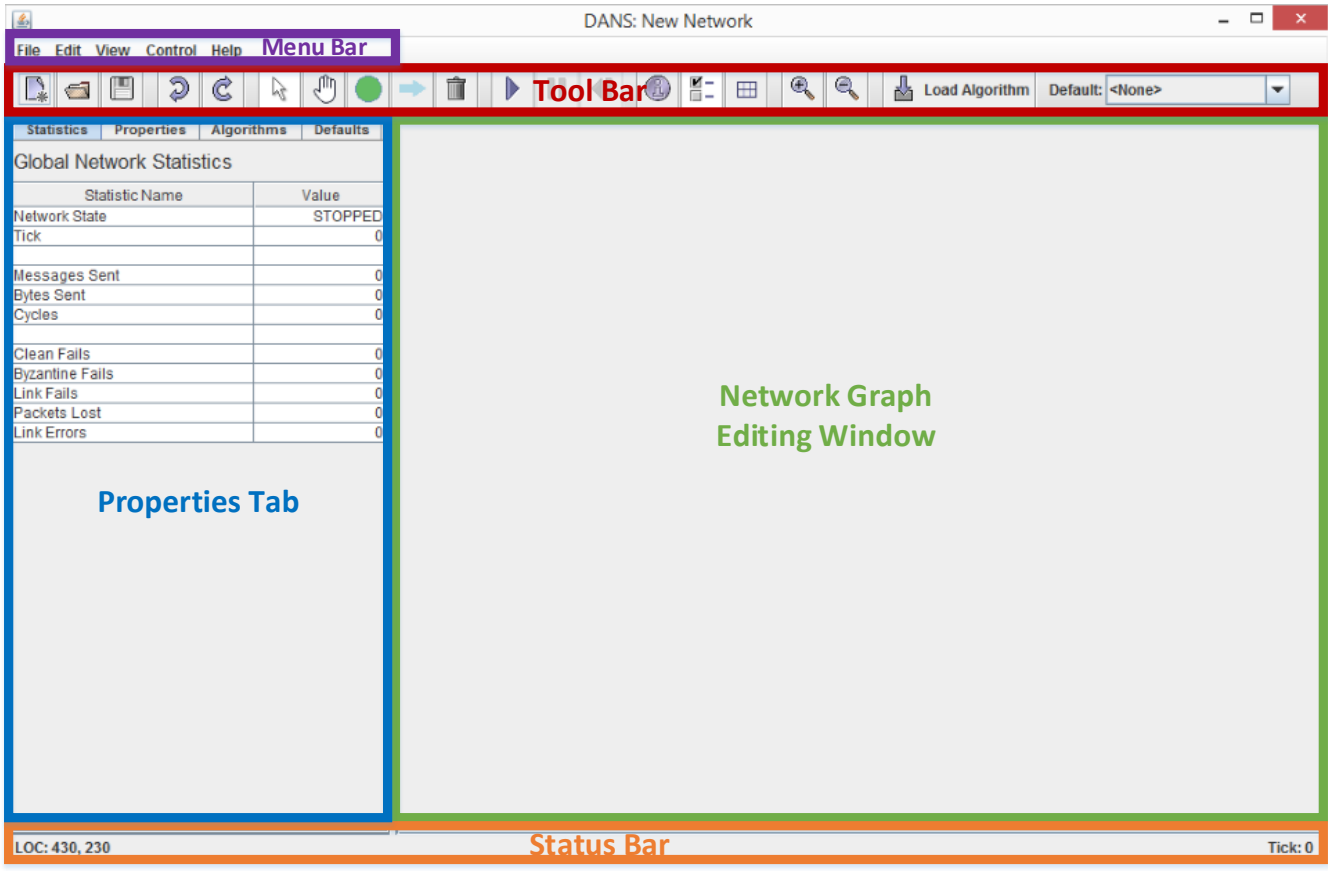
DANS requires a Java version of 8 or higher to function properly. The JGraphX and JLFGR libraries are also required but should be embedded in the jar build of DANS. To run the simulator ensure DANS.jar is in the current directory and type the following in to the console/command line:

```
java -jar DANS.jar
```

DANS will only function correctly in desktop environments that support the swing GUI (i.e. it will not work in console only environments).

Using the GUI

GUI Components:



Network Graph Editing Window: This window displays the current visualization of the network graph and allows editing of links and processors while the simulation is stopped. Selecting a processor or link in this window will change what properties are displayed in the

properties tab (e.g. if a processor is selected, settings for that processor will be shown in the properties tab).

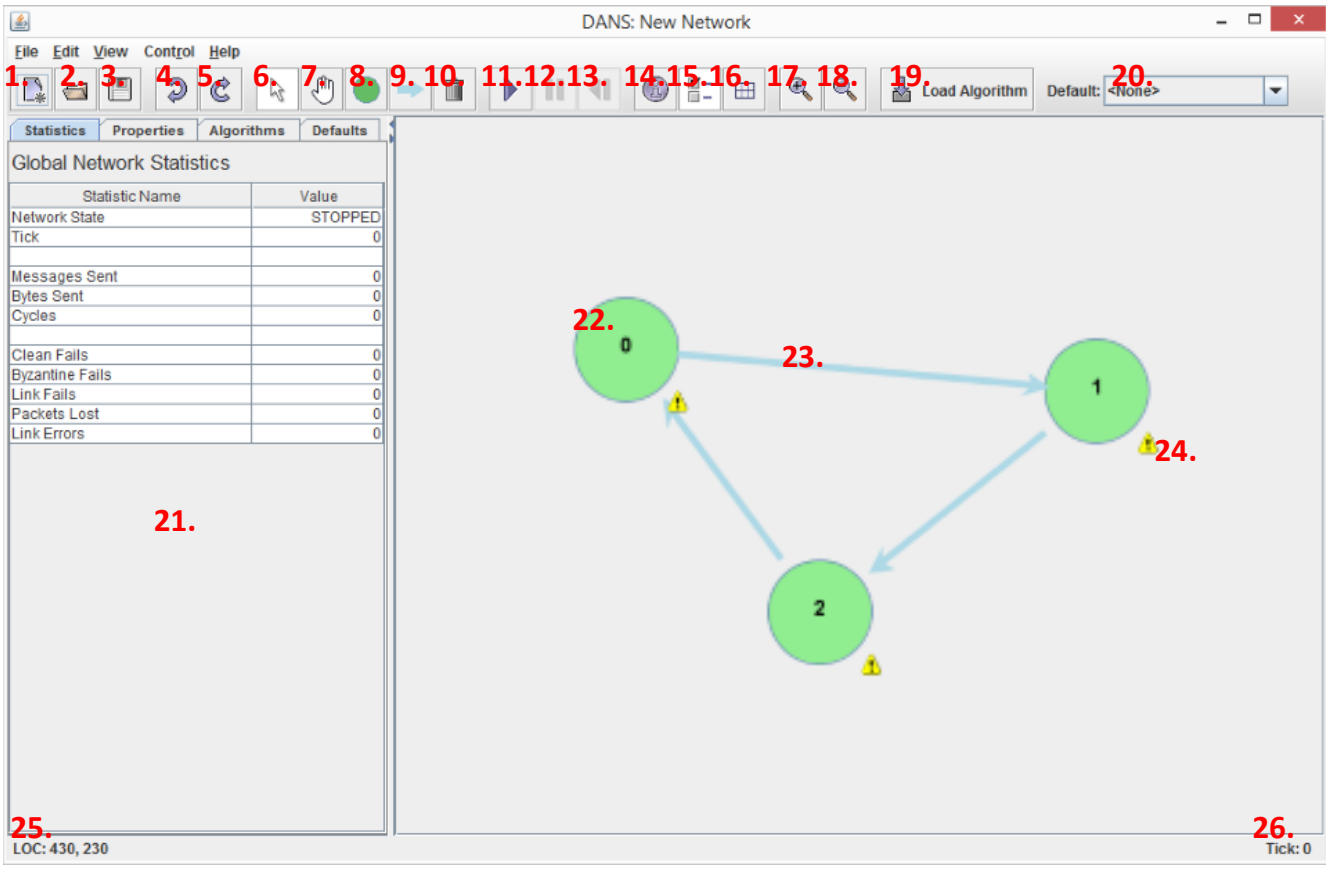
Properties Tab: This window displays statistics and settings for the currently selected object in the network graph editing window. The tabs change what kind of settings or statistics are displayed (e.g. the algorithms tab displays details about the algorithms registered with the currently selected processor). A detailed description of each setting and statistic is given in Appendix B.

Status Bar: The status bar displays the current tick the simulation is on as well as the current location of the mouse relative to the network graph (and accounting for zoom levels).

Tool Bar: The tool bar displays buttons that activate different tools or editing modes. See the next subsection for details on each button's function.

Menu Bar: The menu bar displays additional features not necessarily displayed on the tool bar.

Button functions & Other GUI Elements:



1. **New:** Create a new simulation. Any unsaved work will be lost.
2. **Open:** Open a previously saved simulation. Any unsaved work will be lost.

3. **Save:** Save the current simulation to a file.

4. **Undo:** Undo the last network graph editing action (does not effect settings changed in the properties tab).

5. **Redo:** Redo the last network graph editing action (does not effect settings changed in the properties tab).

6. **Selection:** Activates the selection editing mode. In the selection editing mode, clicking on a processor or link will select them and allow you to hold down the mouse button to move the object.

7. **Pan:** Activates the panning editing mode. If the network graph is zoomed in to the point that scroll bars are shown, the panning editing mode will allow you to move your view of the graph. This has no effect if the scroll bars are not shown.

8. **Add Processor:** Activates the processor editing mode. In this editing mode clicking in any blank space in the network graph editing window will create a processor.

9. **Add Link:** Activates the link editing mode. In this editing mode clicking on a processor will allow you to create a link to the processor that is clicked next. Clicking on a blank space will remove the incomplete link.

10. **Delete:** Activates the delete editing mode. Any processor or link clicked on while in this editing mode will be deleted.

11. **Play:** Start the simulation. While the simulation is running most editing features will be disabled.

12. **Pause:** Pause the currently running simulation. A simulation must be paused before it can be restarted.

13. **Restart:** If the simulation has terminated or is currently paused you may restart the simulation. Restarting the simulation will return the simulation to an editable state like it was before the play button was pressed. A running simulation must be paused before it may be restarted.

14. **Toggle Log:** Display or hide the logging window.

15. **Toggle Properties:** Display or hide the properties tab.

16. **Toggle Grid:** Display or hide the grid.

17. **Zoom In:** Zoom in on the network graph.

18. **Zoom Out:** Zoom out on the network graph.

19. **Load Algorithm:** Displays a file chooser window so that you can select an algorithm to load. The first time an algorithm is loaded it is automatically set as the default algorithm.

20. **Select Default Algorithm:** Sets the default algorithm for processors to run. When selected, all algorithms registered as listening on the default port are changed to the selected default algorithm and processors created in the future will have the default algorithm registered on the default port.

21. **Properties Tab:** See Appendix B for details on the settings and statistics displayed in the properties tab.

22. **Processor:** Circles in the network graph editing window represent processors. Processors that are green are in an OK or running state. Processors that are red are terminated and/or have encountered an exception. Processors that are orange have experienced a failure as a result of the failure probability setting in the processor's settings. Double clicking on a processor will allow you to edit its current ID.

23. **Link:** Arrows in the network graph editing window represent links between processors. Links that are blue are in an OK or not transmitting state. Links that are green are actively transmitting a message that should be displayed in text beside the link (blue text for messages going into the link and green for messages leaving it). Links that are orange have experienced a failure as a result of the failure probability setting in the link's settings. Clicking on either end of a link will allow you to drag it to a new processor to reassign it.

24. **Warning:** A flashing yellow exclamation mark icon indicates that an algorithm has not yet been assigned to the processor. If the simulation is run without assigning an algorithm to a processor, the processor will terminate immediately.

25. **Mouse Location:** The current mouse location relative to the network graph (accounting for zoom levels).

26. **Tick:** The current tick value of the simulation.

Keyboard Shortcuts:

Delete: Delete the currently selected object in the network graph editing window.

Ctrl-a: Select all objects in the network graph editing window.

Ctrl-d: Select none.

Ctrl-x: Cut.

Ctrl-c: Copy.

Ctrl-p: Paste.

Ctrl-=: Zoom in.

Ctrl–: Zoom out.

=: Reset zoom.

Ctrl-z: Undo.

Ctrl-y: Redo.

F1: Selection edit mode.

F2: Pan edit mode.

F3: Processor edit mode.

F4: Link edit mode.

F5: Delete edit mode.

F12: Toggle fullscreen mode.

Appendix B: Properties Tab Settings & Statistics

Statistics

Table 4: Global Network Statistics

Statistic	Description
Network State	The current state of the network (e.g. running, stopped, paused, etc.).
Tick	The current tick the simulation is on.
Messages Sent	The number of messages in total that have been sent up to this point in the simulation.
Bytes Sent	The number of bytes in total that have been sent up to this point in the simulation.
Cycles	The total number of cycles all processors have completed.
Clean Fails	The total number of processors that have encountered a clean failure.
Byzantine Fails	The total number of processors that have encountered a Byzantine failure.
Link Fails	The total number of links that have encountered a failure.
Packets Lost	The total number of messages that all links have dropped.
Link Errors	The total number of bytes that have had errors in all messages sent.

Table 5: Processor Statistics

Statistic	Description
Status	The status of the selected processor (e.g. running, terminated, failed, etc.).
Messages	The number of messages this processor has sent (includes totals from all algorithms running on that processor).
Messages in Bytes	The number of bytes this processor has sent (includes totals from all algorithms running on that processor).
Cycles	The number of cycles completed by all algorithms running on this processor.

Table 6: Link Statistics

Statistic	Description
Status	The status of the selected link (e.g. inactive, active, failed, etc.).
Messages	The number of messages that have been sent through the selected link.
Messages in Bytes	The number of bytes that have been sent through the selected link.
Messages per Tick	The number of messages this link sends per tick on average.
Packets Lost	The number of messages that have been dropped by this link.
Packets Errors	The number of bytes that have contained errors due to faults in this link.

Properties

Table 7: Processor Settings

Setting	Description
ID	A unique identifier assigned to the selected processor. The ID is used to identify what processors to send messages to in user created algorithms.
Delay	The time in ticks it will take the processor to complete a cycle. If the value is set to 0, the processor will ignore the tick system and the next cycle will start immediately after the last cycle finished. If all processors have a delay of 1 and all links have a delay of 0 the simulation will be synchronous.
Speed Change Method	The method used for determining the change in the processors delay. NONE will keep the delay constant (no change). UNIFORM will choose a uniformly distributed pseudo random number each cycle to be added to the delay (this number can be negative). NORMAL will choose a normally (Gaussian) distributed pseudo random number each cycle to be added to the delay (this number can be negative). CONSTANT a constant value will be added to the delay each cycle (can be negative).
Change Seed	The seed used for generating pseudo random changes to the delay.
Change Min	The minimum value the processor's delay can fall to. Should be 1 or more.
Change Max	The maximum value the processor's delay can rise to. Should be greater than min.
Normal Change Mean	Sets the mean of the normal distribution if a normal change method is being used.
Normal Change STDV	Sets the standard deviation of the normal distribution if a normal change method is being used.
Uniform Change Min	Sets the minimum random value chosen if the uniform change method is being used (can be negative).
Uniform Change Max	Sets the maximum random value chosen if the uniform change method is being used. Can be negative but should be greater than the minimum change.
Constant Change Amount	Sets the constant amount the delay changes by if the constant change method is being used (can be negative).
Clean Failure Rate	The probability (in percent) that the processor will fail in any given cycle. Should be a value between 0 and 100. If the processor fails using this method, all message will be sent before it terminates.
Byzantine Failure Rate	The probability (in percent) that the processor will fail after sending any message. Should be a value between 0 and 100. No guarantee is given that all messages in a given cycle will be sent.

Table 8: Link Settings

Setting	Description
ID	A unique identifier assigned to the selected link. The ID can be used to identify what link to send messages on in user created algorithms.
Delay	The time in ticks it will take for the selected link to transmit its message to the target processor. If a delay of 0 is given, messages will skip the link output queue and be directly placed in the target algorithm's input queue.
Bandwidth	The bandwidth of the selected link measured in messages per tick. This value is a double and can be less than one (but should be more than 0). A bandwidth of 0 disables restricting the links bandwidth.
Speed Change Method	The method used for determining the change in the links delay. NONE will keep the delay constant (no change). UNIFORM will choose a uniformly distributed pseudo random number each cycle to be added to the delay (this number can be negative). NORMAL will choose a normally (Gaussian) distributed pseudo random number each cycle to be added to the delay (this number can be negative). CONSTANT a constant value will be added to the delay each cycle (can be negative).
Change Seed	The seed used for generating pseudo random changes to the delay.
Change Min	The minimum value the link's delay can fall to. Should be 1 or more.
Change Max	The maximum value the link's delay can rise to. Should be greater than min.
Normal Change Mean	Sets the mean of the normal distribution if a normal change method is being used.
Normal Change STDV	Sets the standard deviation of the normal distribution if a normal change method is being used.
Uniform Change Min	Sets the minimum random value chosen if the uniform change method is being used (can be negative).
Uniform Change Max	Sets the maximum random value chosen if the uniform change method is being used. Can be negative but should be greater than the minimum change.
Constant Change Amount	Sets the constant amount the delay changes by if the constant change method is being used (can be negative).
Failure Rate	The probability (in percent) that the link will fail after sending any message. This value should be between 0 and 100. Messages left in the links queue after a failure will not be sent.
Packet Loss Rate	The probability (in percent) that any given message will be dropped by the link. This value should be between 0 and 100.
Byte Error Rate	The probability (in percent) that any given byte in a message transmitted by the link will contain an error. If a byte contains an error its value will be randomly altered to that of a different printable character. This value should be between 0 and 100.

Defaults

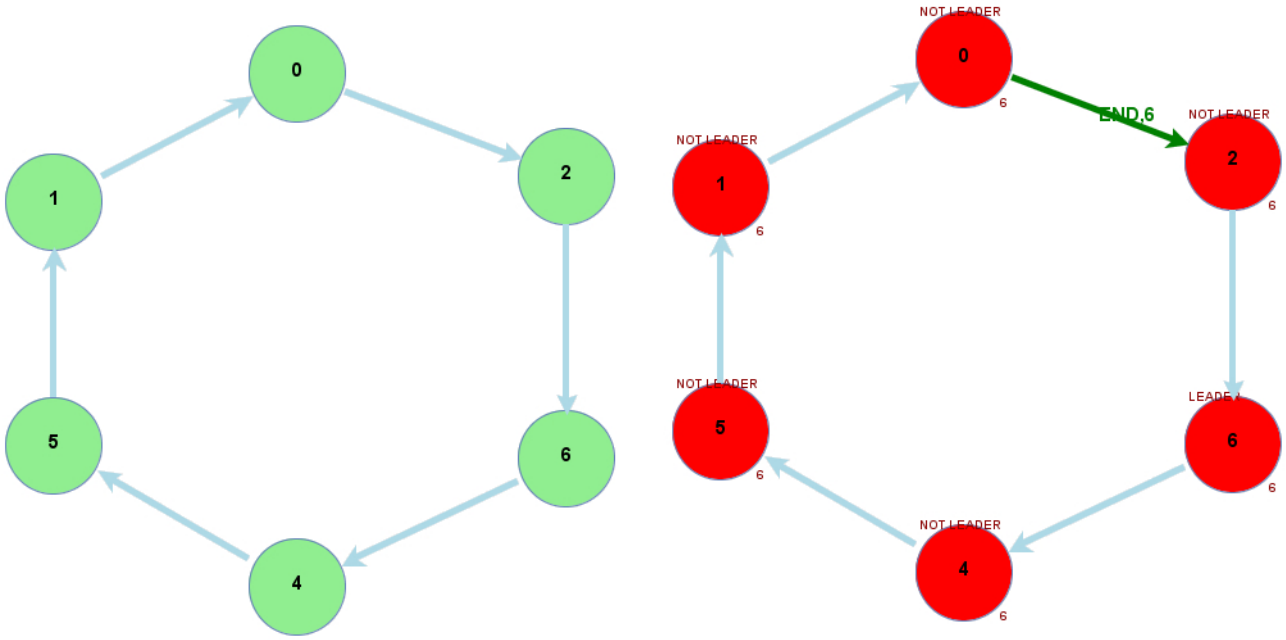
Table 9: Global Default Settings

Default Setting	Description
tickSpeed	The minimum time in milliseconds that a tick will last.
loggingLevel	The verbosity of the log.
defaultPort	The port that will be considered the default port. New processor will automatically have the default algorithm assigned to this port. When the default algorithm is changed, the algorithm listening on this port will also be changed.
defaultSpeedChangeSeed	The default speed for the processor and link delay change methods.
defaultProcSpeed	Default delay setting for new processors.
defaultProcSpeedChangeMethod	Default delay change method for new processors.
defaultProcSpeedChangeNormMean	Default normal mean for new processors.
defaultProcSpeedChangeNormSTDV	Default normal standard deviation for new processors.
defaultProcSpeedChangeUniMin	Default uniform minimum for new processors.
defaultProcSpeedChangeUniMax	Default uniform maximum for new processors.
defaultProcSpeedChangeMin	Default minimum change amount for new processors.
defaultProcSpeedChangeMax	Default maximum change amount for new processors.
defaultProcSpeedChangeConstant	Default constant change amount for new processors.
defaultLinkSpeed	Default link delay for new links.
defaultLinkSpeedChangeMethod	Default delay change method for new links.
defaultLinkSpeedChangeNormMean	Default normal standard deviation for new links.
defaultLinkSpeedChangeNormSTDV	Default normal standard deviation for new links.
defaultLinkSpeedChangeUniMin	Default uniform minimum for new links.
defaultLinkSpeedChangeUniMax	Default uniform maximum for new links.
defaultLinkSpeedChangeMin	Default minimum change amount for new links.
defaultLinkSpeedChangeMax	Default maximum change amount for new links.
defaultLinkSpeedChangeConstant	Default constant change amount for new links.
defaultProcCleanFailRate	Default clean failure rate for new processors.
defaultProcByzantineFailRate	Default Byzantine failure rate for new processors.
defaultLinkBandwidth	Default bandwidth for new links.
defaultLinkFailRate	Default failure rate for new links.
defaultLinkLossRate	Default packet loss rate for new links.
defaultLinkErrorRate	Default byte error rate for new links.

Appendix C: Example Algorithms

Leader Election

The algorithm in Listing 2 elects a leader in a directional ring network as shown in Figure 12a. It is assumed that a directional ring network of at least two processors exists and that there is only one link from each processor. It is required that all processors in the network have numerical IDs.



(a) Example directional ring network.

(b) Example output.

Figure 12: Example input and output for leader election algorthim.

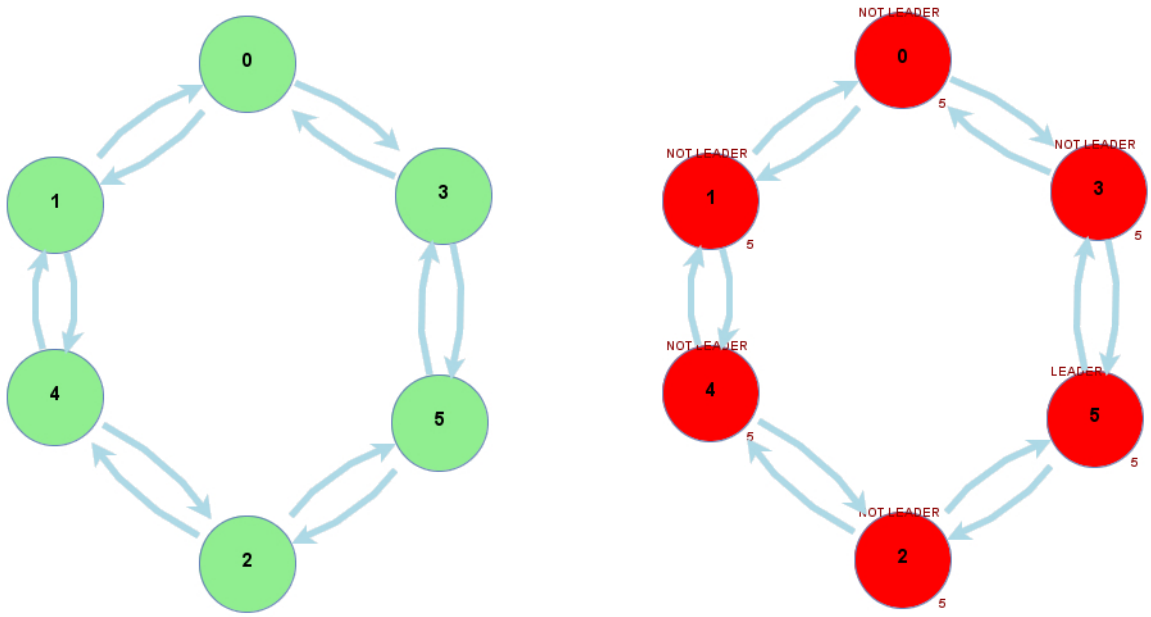
Listing 2: Directional Ring Leader Election

```
1  package dans.algorithm.examples;
2
3  import dans.algorithm.Algorithm;
4  import dans.algorithm.Message;
5
6
7  public class LeaderElection extends Algorithm {
8
9      @Override
10     public Object algorithm() {
11         int id = Integer.parseInt(getID());
12         String msg = getID();
13         String status = "UNKOWN";
14         String neighbors[] = getNeighbors();
15         int leader = -1;
16
17         while(doMainLoop()) {
18             if (msg ≠ null) {
19                 send(neighbors[0], msg);
20
21                 if(msg.startsWith("END")) {
```

```
22                      return leader;
23                  }
24              }
25
26              msg = null;
27
28              Message m = receive(false);
29              if(m ≠ null) {
30                  if(m.message().startsWith("END")) {
31                      leader = Integer.parseInt(m.message().split(",")[1]);
32
33                      if(neighbors[0].equals(leader+"")) {
34                          return leader;
35                      } else {
36                          msg = m.message();
37                      }
38                  } else {
39                      int i = m.toInt();
40                      if(i > id) {
41                          status = "NOT LEADER";
42                          msg = m.message();
43                      } else if(id > i) {
44                          msg = null;
45                      } else {
46                          leader = Integer.parseInt(getID());
47                          status = "LEADER";
48                          msg = "END," + leader;
49                      }
50                  }
51              }
52
53              display(status);
54          }
55
56          return leader;
57      }
58
59  }
```

Bidirectional Leader Election

The algorithm in Listing 3 elects a leader in a bidirectional ring network as shown in Figure 13a. It is assumed that a bidirectional ring network of at least two processors exists and that each processor has a link to the neighbour on its left and right in the ring. It is further assumed that the processors do not know which neighbour is their left neighbour and which is their right neighbour just that they know the IDs of two neighbours. It is required that all processors in the network have numerical IDs.



(a) Example bidirectional ring network.

(b) Example output.

Figure 13: Example input and output for leader bidirectional election algorthim.

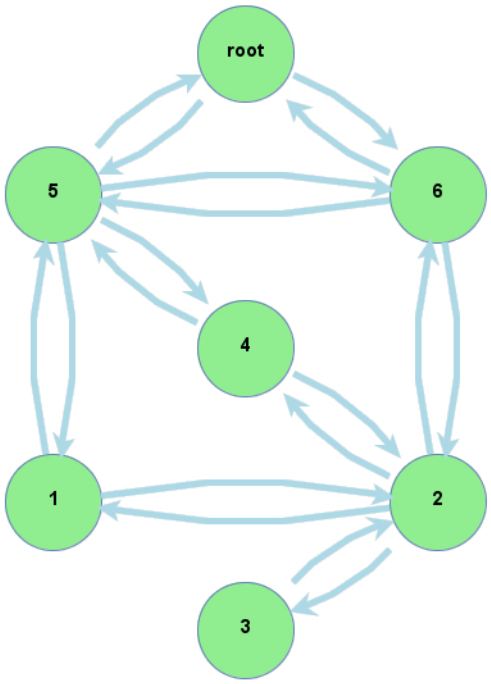
Listing 3: Bidirectional Ring Leader Election

```
1  package dans.algorithm.examples;
2
3  import dans.algorithm.Algorithm;
4  import dans.algorithm.Message;
5
6
7  public class BidirectionalLeaderElection extends Algorithm {
8
9      @Override
10     public Object algorithm() {
11         int id = Integer.parseInt(getID());
12         String msgA = getID();
13         String msgB = getID();
14         String status = "UNKOWN";
15         String neighbors[] = getNeighbors();
16         int leader = -1;
17
18         while(doMainLoop()) {
19             if(msgA ≠ null) {
20                 send(neighbors[0], msgA);
21             }
22
23             if(msgB ≠ null) {
```

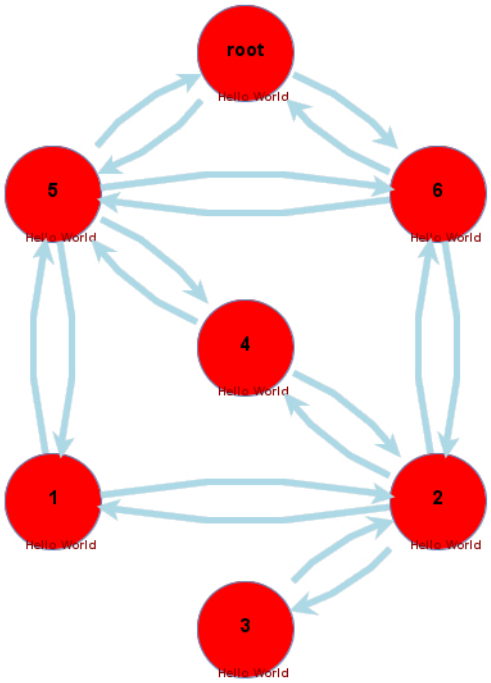
```
                send(neighbors[1], msgB);
            }

            if(leader >= 0) {
                return leader;
            }

            msgA = null;
            msgB = null;

            Message in;
            while((in = receive()) ≠ null) {
                String m = in.message();
                String from = in.from();
                String splited[] = m.split(",");

                if(m.startsWith("END")) {
                    leader = Integer.parseInt(splited[1]);

                    if(from.equals(neighbors[0])) {
                        if(leader == Integer.parseInt(neighbors[1])) return leader;
                        msgB = m;
                    } else {
                        if(leader == Integer.parseInt(neighbors[0])) return leader;
                        msgA = m;
                    }
                } else {
                    int i = in.toInt();

                    if(i > id) {
                        status = "NOT LEADER";

                        if(from.equals(neighbors[0])) {
                            msgB = m;
                        } else {
                            msgA = m;
                        }
                    } else if(i == id) {
                        status = "LEADER";
                        leader = id;
                        msgA = "END," + id;
                    }
                }

                display(status);
            }
        }

        return leader;
    }

}
```

Broadcast

The algorithm in Listing 4 broadcasts the message "Hello World" from the processor with the ID "root" to all other processors in the network. It is assumed that all links are bidirectional and that the network graph is strongly connected. It is required that one and only one processor be given the ID "root". An example network and corresponding output are shown in Figure 14.



(a) Example network for broadcasting.



(b) Example output.

Figure 14: Example input and output for asynchronous broadcasting algorthim.

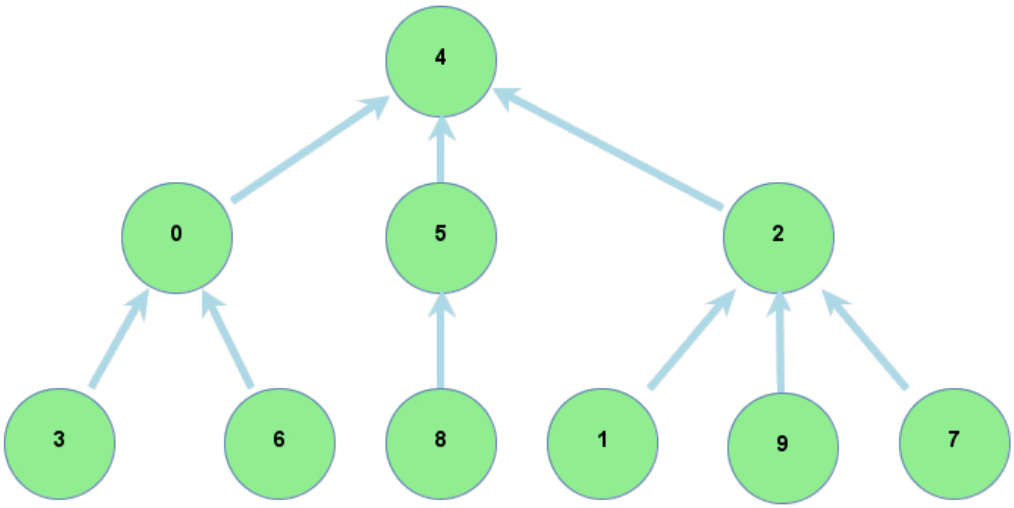
Listing 4: Asynchronous Broadcast

```
1  package dans.algorithm.examples;
2
3  import dans.algorithm.Algorithm;
4  import dans.algorithm.Message;
5  import java.util.ArrayList;
6
7  public class Broadcast extends Algorithm {
8      @Override
9      public Object algorithm() {
10         String parent = null;
11         String message = null;
12         int numNeighbors = getNeighbors().length;
13         ArrayList<String> acked = new ArrayList<>();
14
15         if(getID().equalsIgnoreCase("root")) {
16             message = "Hello World";
17             for(String nid : getNeighbors()) {
18                 send(nid, "BROADCAST," + message);
19             }
20         }
21
22         while(doMainLoop()) {
23             Message m;
24             while((m = receive()) ≠ null) {
```

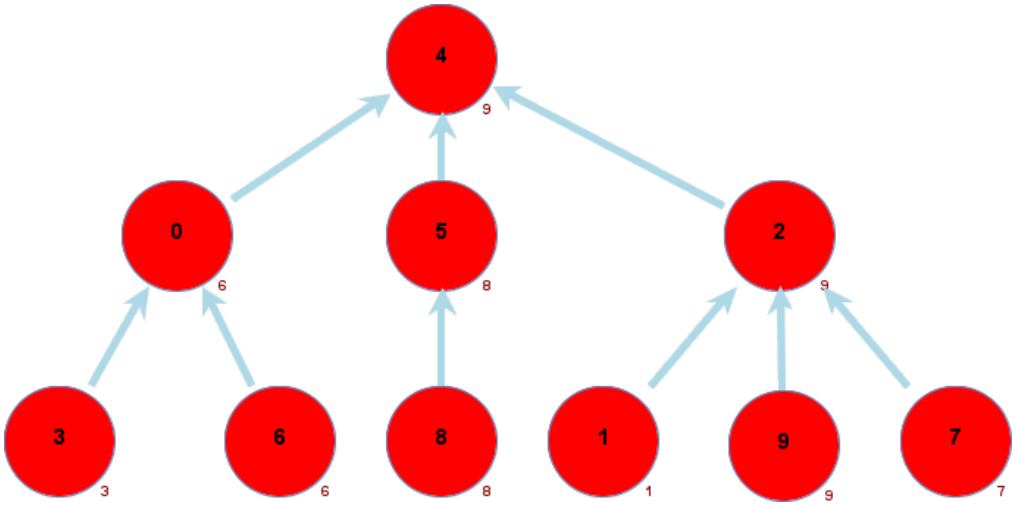
```
25                  if(m.message().startsWith("BROADCAST,")) {
26                      if(message == null) {
27                          message = m.message().split(",")[1];
28                          parent = m.from();
29                          for(String nid : getNeighbors()) {
30                              if(!nid.equals(parent)) {
31                                  send(nid,  m.message());
32                              }
33                          }
34                      } else {
35                          send(m.from(),"ACK");
36                      }
37                  } else if(m.message().equals("ACK")) {
38                      acked.add(m.from());
39                  }
40              }
41
42              if(getID().equalsIgnoreCase("root")) {
43                  if(acked.size() >= numNeighbors) {
44                      return message;
45                  }
46              } else {
47                  if(parent ≠ null && acked.size() >= numNeighbors - 1) {
48                      send(parent, "ACK");
49                      return message;
50                  }
51              }
52
53          }
54
55          return message;
56      }
57
58  }
```

Largest ID

Given a BFS tree of a network, the algorithm in Listing 5 finds the largest ID of any processor in the network. It is assumed that the BFS tree is represented by laying the network graph out in a tree with only one directional link leaving each processor (other than the root) as shown in Figure 15a. Only the root processor will find the correct largest ID, any other processor will simply have the largest ID between its self and its children. It is required that all processor IDs be numerical.



(a) Example tree network for finding largest ID.



(b) Example output.

Figure 15: Example input and output for find largest ID algorthim.

Listing 5: Find Largest ID

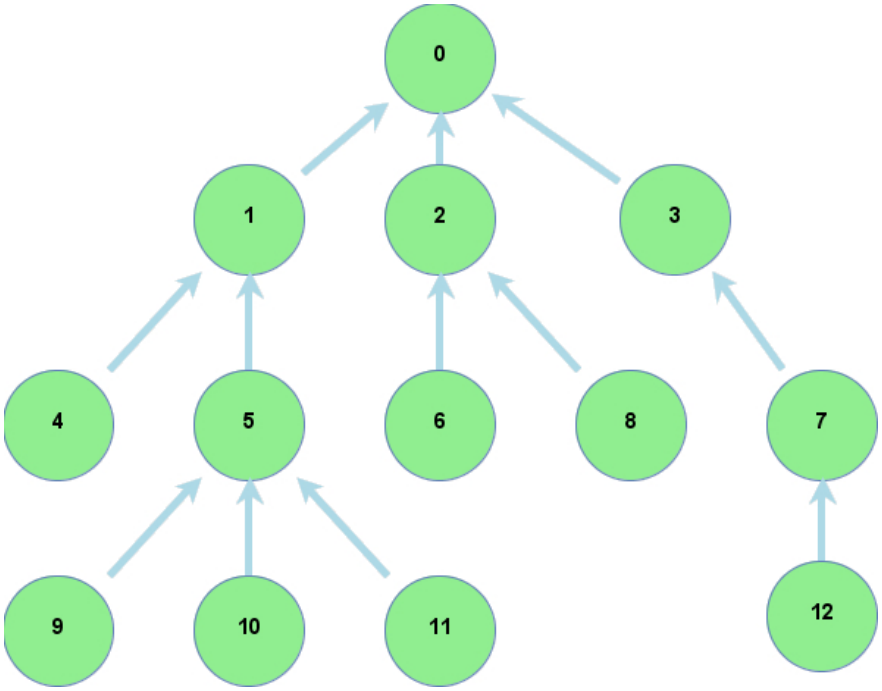
```
package dans.algorithm.examples;

import dans.algorithm.Algorithm;
import dans.algorithm.Message;


public class TreeLargestID extends Algorithm {

    @Override
    public Object algorithm() {
        String parents[] = getNeighbors();
        String childern[] = getSources();
        int id = Integer.parseInt(getID());
        String msg = null;
        int largest = id;
        int heardfrom = 0;


        if(childern.length == 0) {
            msg = getID();
        } else if(parents.length == 0 && childern.length == 0) {
            return largest;
        }

        while(doMainLoop()) {
            if(msg ≠ null) {
                for(String pid : parents) {
                    send(new Message(pid, msg));
                }

                return largest;
            }

            msg = null;

            Message in;
            while((in = receive()) ≠ null) {
                int i = in.toInt();

                if(i > largest) {
                    largest = i;
                }

                heardfrom++;
            }

            if(heardfrom  >= childern.length) {
                msg = largest + "";
            }
        }

        return largest;
    }

}
```

ID Sum

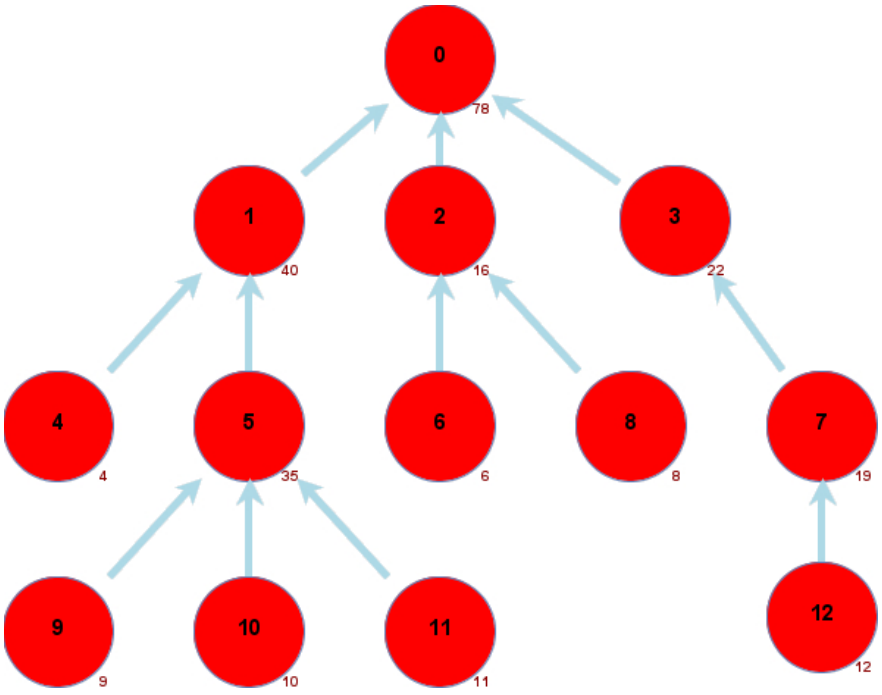
Given a BFS tree of a network, the algorithm in Listing 6 finds the sum of the processor's IDs in the network. It is assumed that the BFS tree is represented by laying the network graph out in a tree with only one directional link leaving each processor (other than the root) as shown in Figure 16a. Only the root processor will find the correct sum, any other processor will simply have the sum of its self and its children. It is required that all processor IDs be numerical.

Listing 6: Find Sum of IDs

```
1  package dans.algorithm.examples;
2
3  import dans.algorithm.Algorithm;
4  import dans.algorithm.Message;
5
6
7  public class TreeSumID extends Algorithm {
8
9      @Override
10     public Object algorithm() {
11         String parents[] = getNeighbors();
12         String childern[] = getSources();
13         int id = Integer.parseInt(getID());
14         String msg = null;
15         int sum = id;
16         int heardfrom = 0;
17
18
19         if(childern.length == 0) {
20             msg = getID();
21         } else if(parents.length == 0 && childern.length == 0) {
22             return sum;
23         }
24
25         while(doMainLoop()) {
26             if(msg ≠ null) {
27                 for(String pid : parents) {
28                     send(new Message(pid, msg));
29                 }
30
31                 return sum;
32             }
33
34             msg = null;
35
36             Message in;
37             while((in = receive()) ≠ null) {
38                 int i = in.toInt();
39                 sum += i;
40
41                 heardfrom++;
42             }
43
44             if(heardfrom  >= childern.length) {
45                 msg = sum + "";
46             }
47         }
48
49         return sum;
50     }
51
52 }
```



(a) Example tree network for finding the sum of IDs.



(b) Example output.

Figure 16: Example input and output for find sum of IDs algorthim.

Appendix D: Known Bugs & Limitations

Know Bugs:

1. Adding two or more links between two processors in the same direction will cause a display bug in which a "loop" will be created in the edge connecting the processors. This has no effect on the simulation.

2. If two processors share a bidirectional link (a link in both directions) and one of the links is dragged to a new processor a display issue can occur in which the edges are not drawn correctly. Moving the processor will fix the issue. This has no effect on the simulation.

3. The print function does not correctly size the network graph to the paper it is printed on and can be cut off or sized incorrectly.

4. In some cases when in full screen mode the file chooser and other dialogue boxes are not shown. Algorithms should be loaded before entering fullscreen mode.

5. The properties tab only allows integers to be input for a processor's ID. It should allow any unique string. A work around is to set the processor's ID by double clicking on it in the network editor window.

6. Some keyboard shortcuts that should be disable are still enabled while running the simulation.

Know Limitations:

1. Only processors can be copied, cut or and pasted.

2. Undo/redo only work on changes to the network graph. They do not undo/redo changes to settings or properties.

3. The simulation visualization can only display a certain number of messages per tick. If multiple messages are sent down the same link during the same tick, not all will be displayed. This has no effect on the actual simulation.

4. If the display command is called multiple times in one tick, only the most recent text will be displayed. This has no effect on the actual simulation.

5. Panning only works if scroll bars are displayed in the network graph editing window (i.e. when the windowed is zoomed in on the graph).

6. Having a large number of processors send many messages every tick will cause the GUI to become unresponsive or even crash.

7. Statistics are combined for all algorithms running on the same processor, there is currently no way to view per algorithm statistics.

8. No distinction is made in the visualization between messages sent on different ports. If multiple algorithms on the same processor are running simultaneously and sending messages down the same link on the same tick, not all messages will be displayed (though they will be sent, just not shown).

9. Link IDs can not be changed after they are created.