Department of Information and Computer Sciences.
Metropolitan State University
St. Paul, MN 55106

Graduate Project Report

Distributed Termination Detection: A Non-General Approach

Robert G. Jahn

Under the Guidance of
Brahma Dathan

Thursday July 12th, 2012

Submitted to the
Department of Information and Computer Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Acknowledgments

I would like to thank my family for their continued support throughout my studies. Their encouragement helps me to succeed in every avenue of life.

I also would like to thank the members of my graduate project committee as well as all of the professors at Metropolitan State: Brahma Dathan, Mike Stein, Jigang Liu, and KuoDi Jian. Thank you for all of the knowledge you have shared with me throughout my years here at Metropolitan State University.

Abstract

Consider a general distributed system. The main function of these systems is to perform a litany of useful computations. Although the computations to be executed perform a very wide range of functions, there is one property in common among many of these computations: In general, they must end. What does a distributed system need to observe in order for it to consider the system as terminated? Moreover, what must be observed to label an individual computation as terminated? Since the early 1980s, there have been many algorithms/papers with the goal of answering these questions in the general sense. Because there is a vast amount of possible computations, this general approach is logical. That is, because the approaches are designed to handle any type of computation, the type of computation cannot affect the functionality of the termination detection algorithm. However, when we consider individual computations, the ability to terminate obviously depends, to some degree, on what type of computation is being performed. Furthermore, the general termination algorithms introduced to this point tend to be quite involved.

In this paper, we shall explore the differences between the termination portions of specific problems (Leader election, consensus, database query, etc...). Based on these termination properties, we shall produce a new, but equivalent, definition of termination. Utilizing this new definition, we shall introduce a series of non-general approaches to the general termination problem. By following a non-general approach, we attempt to arrive at a simpler solution to the distributed termination detection problem. Because the approach is based upon a different definition of distributed termination, the main solution presented is unique when compared to popular algorithms proposed to this point.

The main approach discussed in the paper is dubbed the Final Work approach. It assumes an asynchronous communication protocol. Further, it assumes that it is not necessary that messages are processed in a First-In-First-Out order(FIFO). The Final Work approach compares adequately to other popular solutions to the DTD problem as its performance metrics are comparable to the metrics of the best solutions proposed to this point.

Table of Contents

Chapter 1
Introduction

The Distributed Termination Detection(DTD) problem has been on the forefront of distributed systems problems for over thirty years. In 1980, Nissim Francez, along with Dijkstra and Scholten, simultaneously introduced the problem to the scientific community. Now, for thirty years, those in the field have come up with many different approaches to solve the problem.

Briefly speaking, the distributed termination detection problem can be described as such: Given a distributed system running some sort of distributed computation, the question is how do we detect that the underlying computation has terminated? That is, how can we tell when the underlying computation has completed its work on all nodes throughout the distributed system? Although the problem does not sound at all difficult, it has certainly stood the test of time.

As we shall see in the upcoming section on related work, the range of approaches to solve the distributed termination detection problem is quite broad. That is, over these last thirty years, we have seen many different ideas and techniques used in order to solve the problem. To this point, the solution presented in this paper is no exception. It is quite different from any other popular solution presented to this point in terms of the approach used. However, as we shall see, we have borrowed some ideas from other approaches to help our approach achieve optimal performance.

After reading to this point, you most likely understand the concept of distributed termination detection(If not, don't worry! It shall be expounded upon in the background chapter). However, you may be puzzled by the concept of a non-general approach, as seen in the title of this report. What do we mean by a non-general approach? As shall be seen in the related work section, although the range of approaches is very broad, most algorithms do have something in common: They attempt to detect termination in a completely general fashion. This is where our solution diverts from the norm and explores a different type of approach.

To understand where this sense of 'non-generality' crops up in the approach, we must ask the following question. What exactly are the adjustable components with which we are dealing when discussing the DTD problem? That is, what are the components of the problem that can change from situation to situation? Well, we have the underlying distributed system as well as the computation being executed. Here, we are mainly concerned with the type of computation being executed. Back to the concept of the non-general approach, most other approaches use a completely general approach. That is, these solutions to the DTD problem do not consider the type of computation being executed. However, one can easily see that the termination properties of the computations shall differ from computation to computation.

This is the driving force, and also the reasoning behind the title, of our non-general approach. Essentially, it is built on the assumption that individual nodes can retain some information on the actual computation being executed. Using this information, we are able to build an efficient approach to solve the DTD problem.

1.1 Contribution Summary

We have made ample contributions to the problem throughout this project. First, we have

observed specific distributed computations in order to determine exactly what is needed by these computations in order for them to terminate. Essentially, this process consists of describing the algorithm and then focusing on termination properties of said computation. By doing this, we can determine the feasibility of our approach if we are able to accurately generate termination criteria from these computations.

Our second contribution is the creation of a new definition of termination on a distributed system. With the previously acquired termination properties in mind, we develop a new, but essentially equivalent, definition of termination on a distributed system. By creating this new definition, it gives us a new base from which to develop our approach. Specifically, we take the generally accepted global definition of termination and define it in the local sense. This allows us to fully work from our non-general approach.

For our next contributions, with this new definition at hand, we develop a series of algorithms in order to solve the DTD problem. Although the first couple of algorithms are naive, they serve as a stepping stone to the final approach of the paper: The Final Work approach. This approach is then fully expounded upon and described in great detail. Further, we also discuss non-theory issues, such as implementation strategies.

Finally, our last major contribution is the development of a JAVA simulation environment. This environment is to be used in the testing of our final work approach. In this report, we discuss the design behind the project, discuss the issues discovered during the creation of the project, and also run a few test scenarios.

Note that this latest section was simply a summary of our contributions. We shall go into much greater detail regarding these contributions within the body of the report.

1.2 Project Organization

The remainder of this project report shall proceed as such. First, we shall go into some depth regarding the background of the distributed termination detection problem. Specifically, we shall observe some basic introductory material as it relates to distributed systems in general. After we have achieved this breadth of understanding, we shall observe some terminology and concepts as it relates to the distributed termination detection problem specifically. Finally, to wrap up this background section we shall observe some of the related work in the field. Although there are literally hundreds of distributed termination detection algorithms, we shall mainly discuss the papers that are most closely related to our work. Otherwise, we also discuss some landmark papers as we consider them important in the sense they were responsible for introducing some important approaches to the DTD problem.

Second, we shall observe some basic computations and specific examples of these computations. The reasoning behind this is to try to abstract some termination properties as they relate to specific computations. By doing so, we shall have extra quantifying information at our disposal during the creation of our approach to the problem. Specifically, in this section, we shall only observe the following types of computations: Leader Election, Breadth-First Search, Consensus, Edge-Coloring, and Distributed Database Queries. These have been chosen as we feel that they provide a decent scope in terms of possible distributed computations.

Next, with this previously attained quantifying information at hand, we introduce an initial sequence of algorithms to be used in order to solve the distributed termination detection problem.

Initially, we introduce a very naive algorithm, which we actually have no intention of actually using as a solution to the problem. It simply serves as a base solution to be used in certain restricted situations. Using this naive algorithm as a base, we introduce a second algorithm; however, this second algorithm is also quite naive in the sense that it requires too much information about the underlying computation to be feasible. Finally, from this second algorithm, we introduce the third and fourth algorithms. The third algorithm is based on having an accurate measure of the statistics behind given computations. Finally, the fourth algorithm, which is the main approach described in this paper, is introduced. After a small discussion on the potential feasibility of this approach, the Final Work approach, we proceed to the next section in order to expand upon the approach to a greater degree.

Next, we introduce the real meat of the project by describing how individual computations could be accurately defined under the final work approach. From this information, we attempt to make a sort of categorization of different types of final work experienced by different computations. As can be seen in this section, we have success in accurately defining the final work properties for all but one of the types of distributed computations we had chosen from the previous chapter in which we observed the termination properties of specific computations. To end this chapter, we discuss some general concerns and summarize the feasibility of the Final Work approach.

In the next chapter, we discuss some of the specific mechanics of the Final Work approach. First, we discuss exactly how an individual node would detect its individual termination. Included in this discussion would be the exact mechanics of impressing the notion of final work onto an individual node. Next, we discuss a few different ways to detect the overall termination of the system. That is, given the notion of local node termination at each node, how do we become informed of the overall termination detection of the system. Here, we present a few different methods to be used depending on different scenarios. Further, we discuss the optimality of the different approaches. Next, we discuss the difference between solving the overall termination of the system and solving the termination of a single given computation. Finally, we discuss a simple method to allow different computations to be executed in certain scenarios while adhering to the notion of mutual exclusion.

Next, we discuss the performance of the Final Work approach. In doing this discussion, we compare the final work approach to the most popular distributed termination detection algorithms conceived over the last 30 years. We essentially consider two major metrics, message complexity and the time delay from achieving termination and detecting termination.

Following this performance discussion we present the associated JAVA project used to display the performance of the Final Work approach. This presentation includes a discussion on the overall design and reasoning behind certain design decisions, hardships found during the coding/testing processes, as well as a few scenarios executed using the software.

Finally, we present a discussion on importing the approach to an imperfect distributed environment. Mainly, we touch on the troubles that arise in an asynchronous and faulty environment. Further, we present solutions on how to handle these issues when considering our approach.

Chapter 2
Background

Before delving into the real meat of the problem and the solutions to follow, we shall first discuss the important elements and inner-workings of the distributed system. With this breadth of information at hand, we shall then observe the problem of distributed termination detection and the related work as it pertains to the problem of distributed termination detection.

2.1 Background Information

What is a distributed system? What are its key components? Essentially, there are two key physical items to consider in any distributed system: Nodes and Links.

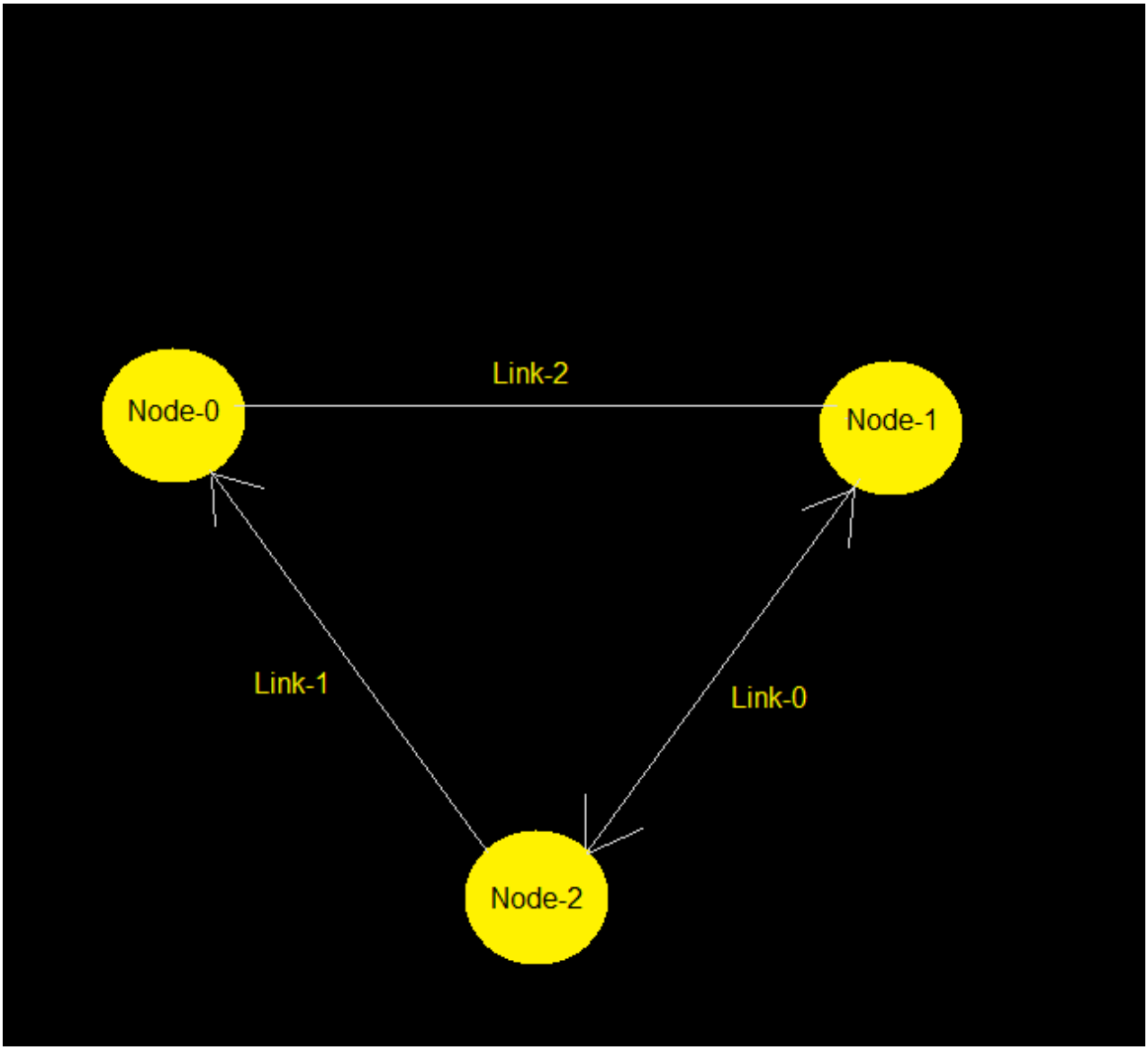


Figure 1

Nodes, also known as computing centers or processors, are just that: Individual machines, capable of running instructions autonomously from other nodes. In a distributed system, the nodes are responsible for the majority, if not all, of the instruction execution. Refer to figure one. In this figure, we have three nodes: Node-0, Node-1, and Node-2.

Links, also known as edges, are responsible for the communication between nodes. Nodes, whether through a physical wired link, or a wireless link, need to be able to send messages to each other. Generally, we shall associate a delay on these links where the delay is defined as the average amount of time it shall take for a single message to travel from one node to another node across a given link. Refer to figure one. Here, we have three links: Link-0, Link-1, and Link-2. We can plainly see that Link-0 connects a pair of nodes: Node-1 and Node-2. Similarly, Link-1 connects Node-2 and Node-0.

In general, we can expect two types of links to be present in a distributed system. The link could be classified as bidirectional. This means that the link can be traversed in both directions. For instance, referring to figure one, we can see that Link-0 is bidirectional as the arrows point in both directions. This means that Node-1 can send to Node-2 and conversely Node-2 can also send to Node-1. Also note that Link-2, which has no arrows, is also considered to be bidirectional. This shall be true when we have a link that has no arrows on either end.

A link could also be classified as unidirectional. This means the link only allows traffic in one direction. For instance, referring to figure one, we can see that Link-1 is unidirectional as there is only one arrow on the link. This means that Node-2 can send to Node-0; however, Node-0 cannot send a message to Node-2 directly. In this scenario, Node-0 would have to send the message to Node-1 first. Then, Node-1 would relay the message to Node-2. For our purposes, unless specified otherwise, we shall assume that all links are bidirectional.

In general, we shall consider two types of messaging environments in which our distributed systems exist: Synchronous versus Asynchronous environments. A synchronous environment is one that proceeds in a lock-step fashion, round by round. All nodes within the system are strictly synchronized such that there is predictability as to when a message should be received by a particular node. On the other hand, an asynchronous communication protocol is sporadic. Messages are not sent and received on a round by round basis. Messages can be delayed indefinitely. Further, message order is not guaranteed to be received in a First-In-First-Out(FIFO) fashion. Instead messages are assumed to be processed randomly(Although we could 'predict' in what order messages are processed, this is not necessarily helpful when it comes to our approach). Because creating an algorithm for use in a synchronous environment would not be of much use in a real-world setting, we shall build our solution with an asynchronous communication environment in mind.

Distributed systems are also vulnerable to certain types of faults and failures. A node fault is a situation where a given node dies or locks up. This node might be able to reboot itself or it might be down indefinitely. While it is down, it can perform no local execution, send any message, or receive any message. Messages sent to a faulty node are simply dropped or lost. The other major type of fault is known as a link failure. A link failure is a situation where a given link drops a message for some reason. It could be the link has died or a message was simply lost in transit. During the creation of our solution, we shall not assume the presence of either of these types of failures. However, we shall build the solution with the possibility of failures in mind so that certain decisions might be made to avoid issues with faults down the road.

Now that we have discussed the physical components of the distributed system, let us consider

the software components. Specifically, we are concerned with the inner-workings of an individual node. As mentioned previously, a node is capable of executing a certain range of instructions. What type of instructions with which are we dealing? Essentially, we want to execute distributed computations on these nodes. That is, we desire to execute decentralized algorithms that each node must individually utilize in order to execute the computation over the entire set of nodes. An example of a distributed computation would be a simple leader-election algorithm. In this type of algorithm, the goal is for each node to decide on which of the nodes should be the leader. In order to make this decision, each node must pass along information to every other node in the system currently running the computation. Once each node knows of every other node's information, each node can individually make a decision as to which node should be elected the leader(Hopefully, the algorithm is correct and they all choose the same node!). In general, regardless of the end goal of the computation, the process of a distributed computation shall consist of some amount of message passing as well as individual processing local to each node.

2.2 Distributed Termination Detection

Now that we have developed some knowledge about the distributed system, let us consider the problem at hand.

As mentioned previously, a distributed computation generally consists of some amount of message passing, some amount of internal processing of information at each node, and finally each node makes some sort of decision. Some time after this final decision is made, the computation terminates. The goal of distributed termination detection is to detect when the underlying computation has fully terminated. That is, we want to detect when the underlying computation has completed all of its tasks and no node used in the computation shall perform any more tasks as it relates to the computation in question. Hopefully, we can make such a detection as quickly as possible from the point of actual termination to the detection of said termination. Also, we would hope to keep the number of added messages to a minimum.

There are two broad definitions when it comes to determining the termination status of a given distributed system. First, many different authors have used the following definition of distributed termination detection: A *distributed system* is said to have terminated once all processes have become passive/idle and when all channels between all nodes are empty. Alternatively, the distributed termination detection problem can be broken down further. This alternative definition defines the distributed termination detection problem as follows: A *specific computation* can be considered terminated once all nodes are passive with respect to the *given computation* and there exist no messages related to the *given computation* within any channel in the system.

This brings a peculiar difference in the two definitions of termination to light. We must ask the following question. Are we only concerned with the termination status of a given single computation being executed on a given system? Or, are we concerned with the general termination status of the distributed system itself? That is, are we concerned termination status of a single computation or are we concerned with the termination statuses of every separate computation currently being executed on the system? At the end of the day, it really does not matter which question we are concerned with answering. We shall be answering the question in the singular sense (Discovering the termination status of a single computation) and then we shall extend this solution to answer the general question (Discovering the termination status of the entire distributed system).

However, it is important to understand the difference between the two queries. One can easily

imagine a scenario for which an adequate solution would be desired for each problem. For instance, consider a very large distributed system, System-A. System-A is constantly executing computations concurrently over different sets of nodes contained within the system: So much so, that there is exactly zero downtime for the system. That is, there is always a computation being executed on the system. In this scenario, we would probably not need a solution to the general problem as we already know that the system shall always be executing some computation at any given time. However, we could imagine that a solution to the singular problem would be quite useful: Computation scheduling for instance.

Now, consider a somewhat small system, System-B. This system often experiences some downtime where there are no computations being executed. Here, a solution to the general version of the problem would be desirable as we can imagine a scenario where we would be interested in being informed as soon as the entire system has reached a terminated status: Scheduling for a down time for instance.

In terms of the DTD problem, we have some specialized notation to consider. First, in any solution to the DTD problem we shall introduce extra messages to the distributed system(Although sometimes these added messages can be piggybacked to the messages of the underlying computation). These messages, which are used to help detect the termination status of the system, are generally referred to as *control* messages. The name comes from the fact that the messages are used to control the termination detection of the underlying computation. The amount of control messages added to the underlying computation is one of the metrics with which we are concerned. Hopefully, we do not add many additional messages to the underlying computation. We shall refer to this metric as *message complexity.*

The other metric with which we are concerned is known as the *detection delay.* The *detection delay* is defined as the delay from actual termination to the detection of said termination. Once again, we try to design solutions such that this delay is as small as possible.

Now, we shall discuss some of the related work as it pertains to the distributed termination detection problem. As we shall see, there have been many types of different solutions, utilizing different approaches, and yielding unique solutions.

2.3 Related Work

As mentioned previously, the distributed termination detection problem has been on the forefront of computer science for over 30 years. It should come as no surprise that there have been many different types of solutions created during this time. In this section, we shall discuss the most prominent approaches as well as the approaches that most closely served as inspirations for our approach, to be discussed later. However, it should be noted that the approach given in this paper is quite unlike any of the other approaches given in this section. Further, we shall also discuss some additional papers that do not necessarily introduce a solution to the DTD problem, but rather contains some auxiliary information as it pertains to the DTD problem.

2.3.1 Wave-based Approach

The first major approach we shall discuss is generally referred to as the Wave approach. We shall discuss this approach first since it makes up the largest subset when one considers the entire set of solutions to the DTD problem. The descriptive name, 'Wave', is taken from the behavior of the approach. Simply put, given the nodes in the system, either a single node, or a set of nodes, are chosen

to be initiating nodes. These initiating nodes, or initiators, are responsible for determining the detection status of the underlying computation(which is being executed on the same nodes). They make this determination through the use of message sending. Basically, the initiator, or initiators, send messages to every node in the entire system. Information from these nodes is then added to the message body of each original message before being sent back to the original initiator(s). How the initiators handle this information varies from approach to approach, however, this notion of a 'wave' of messages traveling across a system before traveling back to the original initiator(s) shall be the driving concept behind any approach described as a Wave approach. Basically, the initiators are constantly requesting and receiving status updates from other nodes in the system until they discover that the computation has globally terminated.

The main approach presented in this report, the Final Work approach, shares some similarities with the Wave subset of DTD algorithms; however, it cannot be defined as a Wave approach. In the Final Work approach, we shall see that there really is no concept of an initiator. Instead, each node in the computation is solely responsible for determining its own termination status. This shall be discussed in greater detail when the approach is developed later on in the report.

Now, we shall observe a few examples of Wave-based algorithms. We shall discuss them in enough detail to understand how the approach works; however, please refer to the actual papers for a complete understanding of the approaches discussed.

In the 1983 paper, [29], S.P. Rana presents an early, wave-based solution to the termination detection problem. This algorithm, like the Dijkstra-Scholten algorithm, serves as an important base for the approach presented in our paper. The algorithm is dependent on the fact that all processes used in the computation are connected by a Hamiltonian cycle, and that all communication travels in a single direction. Essentially, the paper discusses the completion of a local predicate at each node. Through the use of synchronized clocks at each node, the individual nodes shall each mark down the time when they have completed their local predicates. Upon marking this local time, the node shall send a detection message to the next node in the cycle with both the predicate completion time and an incremented counter. If the detection message is received by an active node, it is simply purged. However, if it is received by a node in a passive state, and if it passes a test to make sure the predicate completion time is greater than its own completion time, then it increments the counter, attaches its predicate completion time, and sends it down to the next node in the cycle. If, at any point, we have a passive node receive a message with a counter equal to the amount of nodes used in the computation, we can declare that termination has been detected. It should be noted that the presented solution is to be used only in a synchronous environment; however, Rana presents possible enhancements to use in order to transform the solution to be applicable in an asynchronous environment. Although the solution is simple, we must note that it is dependent on a number of constraints. First, and most importantly, it is not applicable in an asynchronous environment. Second, it forces the underlying structure of the distributed system to be a simple Hamiltonian cycle. Neither of these characteristics are likely to be found when considering a distributed system in general.

Our approach, the Final Work approach, greatly expands on this notion of detecting the completion of a local predicate. The paper considers the detection of the local predicate as a given and focuses on the actual global termination detection problem over the entire system. Obviously, the "local predicate" shall depend on the computation being executed as well as the node being considered. We expand on this notion throughout the paper. Further, we present a similar solution to the global detection portion of the problem.

Also in 1983, in [23], J. Misra presents a very early strategy in the detection of termination in a distributed system. Here, we have another prime example of a wave-based approach. Essentially, the algorithm's approach revolves around the idea of a single token. This token traverses through the given distributed system, traveling from node to node, in order to detect termination. Like most authors, Misra assumes the termination definition to be the equivalent of every node being in a passive state with no messages on any link in the system. Initially, all nodes are painted black. Essentially, the token marker runs from node to node, incrementing a local count as it traverses past each node that is in a passive state. Also, as it passes through each passive node, it paints the given node white to signify its passive state. If the marker is able to traverse every node without running into an active node, it considers the system to be terminated. This is an important paper in the sense that it was one of the first to introduce the concept of a traveling marker whose sole purpose is to detect termination.

In [20], we are introduced to a major player in distributed termination detection problem: Friedemann Mattern. In this paper, he presents an entire slew of distributed termination detection algorithms. The algorithms all follow the wave-based approach as described above. Further, the algorithms presented are to be used in an environment in which asynchronous message passing can be assumed. It is also assumed that the model shall allow only a finite amount of time for messages to be sent. However, it is not assumed that messages shall be discovered in the same order that they are sent. These are the environment conditions which most closely resemble the conditions considered as the end goal for the approach presented in our paper.

Basically, all of the algorithms presented by Mattern in this paper are based on message-counting. With this approach, Mattern essentially creates solutions with the following idea in mind: In order to detect termination, we can count how many total messages are sent by the computation. We can also count how many total messages pertaining to the computation have been received by the individual processes used by the computation. Setting aside some small, but important, details, the main idea is that termination follows when the total number of messages received equals the total number of messages sent. Obviously, all nodes would also need to be in an idle state. This paper really gives a good notion to all of the trouble we shall run into when importing a synchronous solution to an asynchronous environment. That is, once we must consider various types of system failures(node/link failures or inordinate delays), the synchronous algorithm may need to be modified in order to operate correctly in an asynchronous environment. Although we created our approach to be used within a synchronous environment, we created the approach with these exportation difficulties in mind. That being said, we have avoided the reliance on system clocks and global snapshots.

2.3.2 Invigilator Approach

Another popular approach to solving the DTD problem is known as the Invigilator approach. In this approach, we have a single node that acts as a leader of all other nodes used in the system in question. This leader node, or coordinator as it is sometimes called, is responsible for maintaining the status of all other nodes used by the computation. Throughout the lifespan of the underlying computation, each node shall continually update the coordinator of its status. Obviously, this approach's success shall be based around the topography of the underlying system. That is, both message count and time delay can be kept at a lesser amount if the coordinator node is more closely connected to each node in the system. The closer this coordinator node is connected to each node in the system, the less time it shall take to send update messages back and forth from individual nodes to the coordinator node and vice versa. Relating this to the wave-based approaches discussed earlier, we can think of the invigilator approach as a wave-base approach with a single initiator, with all 'waves' of messages simply coming into a single node. Obviously, the more important piece here is the fact that the

underlying topology is restricted in such a way that the single initiator has direct communication with all other nodes in the system.

Given this heavy reliance on the underlying topology of the distributed system, Invigilator approaches generally assume that the underlying topology is the shape of a star. A star, in terms of distributed systems, indicates a topology such that we have a single node which has direct access to all other nodes in the system. That is, there exists a single node in the system which has a link connecting to each other node in the system. Obviously, in the broader terms of the invigilator approach, this node would be our coordinator node.

Given this assumption, there have been several invigilator approaches created to date; however, they are not as abundant as the wave-based approaches as discussed earlier. Here, we shall discuss one such approach in [26]. The main reason for this lack of abundance is that underlying system topologies generally cannot be assumed to be so advantageous in the sense that we would have a single node that has such instant communication with all other nodes in the system. It should be noted that any invigilator approach can also work in a topology other than a star topology; however, this shall generally induce unreasonable time delays and added message complexity.

In [26], N. Mittal et al. present and discuss a family of different algorithms to be used to determine the termination status of non-diffusing computations. Here, a non-diffusing computation is one that does not end under its own accord. Further, they do not assume a specific underlying system topology. For instance, unlike a few of the previous papers, these algorithms do not assume that the underlying network must be a tree. However, in the creation of their algorithms, they combine the ideas of the invigilator and computation tree approaches. By doing so, they achieve the optimal detection delay time complexity of the invigilator approach with the optimal message complexity of the computation tree approach.

The main idea behind the algorithm is simple. Considering the computation tree approach, a specialized version of a parental responsibility algorithm(discussed next), we have a computation tree being built during the execution of the underlying computation. This tree is built with a single initiator, which is place as the root to the entire computation tree. In the computation tree approach, for a node to report its status, it must become inactive towards its parent. This leads to additional messages needed as children report to their parents, and said parents report to their parents, and so on. Now, consider the Invigilator approach. Here, we know that a node will only report to the *coordinator* when it has become inactive. So, the goal is to combine the minimal detection delays experienced by the computation tree based approach with the minimal message complexity of the Invigilator approach. This is done in an interesting way. Basically, each time a node desires to report its inactive status, it makes a decision. Basically, a node shall simply report to its parent node until the path from the node to the coordinator is longer than D, where D is the diameter of the system of nodes. By doing so, they achieve a detection delay of O(D) and a message complexity of O(M). Here, M refers to the amount of messages exchanged by the underlying computation during its execution.

Although, on its own, the Invigilator approach is undesirable in the sense that it generally requires that the underlying topology be restricted to be in the shape of a star, we can still see that the approach shows promise. In fact, Mittal et al., with their respective algorithm, show us that the Invigilator approach can still be utilized in an auxiliary way to improve the performance metrics of an existing approach.

2.3.3 Parental Responsibility

The second most popular approach to solving the distributed termination detection problem is likely the parental responsibility approach. In this approach, we actually build a computation tree during the execution of the underlying computation. Generally, we build the tree during the execution of the computation, and while the computation unwinds the tree is dismantled. Once the tree is fully dismantled, the computation can be considered terminated.

The parental responsibility approach can often yield decent complexities: In terms of both message passing and time delay. However, the main negative of such an approach is the busy work involved in forming a computation tree. Further, the approach is often reliant on the underlying topography of the distributed system. If the underlying topography is not in the form of a tree, extra care must be taken to ensure the approach will still work under the different conditions. This extra care will usually add extra delays and messages to the solution.

In 1980, the termination detection problem for distributed systems was first brought to the attention of the computer science community in a joint manner by Nissim Francez and the group of Edsger Dijkstra and C.S. Scholten. In [10], Dijkstra and Scholten discuss the problem and present a solution to be used in a scenario where the underlying topology is in the form of a tree. They also present us with the very first parental responsibility solution to the problem. First, we consider the scenario where the underlying system topology is in the shape of a tree. Here, we have a single initiator node: The root of the tree. A node then handles incoming messages in one of two ways. If a node is currently active in the computation, it sends an acknowledgment message back to the node that originally sent the message. If a node is not yet active in the computation, the node is added to the tree by setting it as the child of the node that originally sent the message. The algorithm ends by the following rule: A given node has no children left and has also become passive in the computation, it sends a release message to its parent and removes itself from the tree. Then, the algorithm is considered to be terminated once the root, or the sole initiator, has no children left and has also become passive. This simple, but elegant, algorithm can also be extended to two other underlying topologies: acyclic and cyclic graphs.

In [24], J. Misra and K.M. Chandy present a very early discussion on the distributed termination detection problem and some peculiarities pertaining to the details of the underlying system. Like the Dijkstra-Scholten algorithm, it utilizes a parental responsibility based approach. Actually, the approach presented by Misra and Chandy is an adaptation of the Dijkstra-Scholten algorithm with a few important differences. First, whereas the Dijkstra-Scholten algorithm handles a system model where messages do not have to wait to send messages, Misra and Chandy tackle the problem in the scenario where messages cannot be sent under certain circumstances: When a link is being used for instance. The key difference here is that it was somewhat easy for a node to locally decide that it was passive given the non-waiting assumption of the Dijkstra-Scholten algorithm. However, since messages can be arbitrarily delayed, there is another type of passivity to consider: The passive status induced by the fact that a node is expecting a message but the message has been delayed for some reason. To distinguish between these types of passivity, the authors introduce a sort of signaling scheme. Essentially, the nodes emit two types of signals: Activity signals and Blocking signals. Since these signals can now distinguish between the two types of passivity, the Dijkstra-Scholten algorithm is modified to work in this more restricted communication protocol. This is another landmark paper as it shows us some important differences to consider when developing algorithms under different environmental/model conditions. It helps us to see how an algorithm can be modified to operate under different conditions; however, this shall often introduce additional complexity.

In [3], A.H. Baker et al. present an algorithm based upon the SKR algorithm. The SKR algorithm is unique from other algorithms in the sense that it has the ability to adapt to varying stress levels of the underlying computation. Because of this adaptive efficiency, Baker et al. chose this as the base for their algorithm, which is also to be used in parallel with the underlying computation. It should also be noted that these algorithms are to be used in an environment where the memory system could be described as a distributed-memory system. These algorithms can be grouped with several others in the sense that it utilizes a distributed system with a spanning tree-type structure. Once again, we have access to a fine example of an algorithm that can be modified to perform better under different circumstances.

In [16], N.R. Mahapatra and S. Dutt present an algorithm with the detection delay time as the most important metric to optimize, while still retaining decent complexity over the other metrics. The algorithm, which resembles many previously presented spanning-tree type algorithms, has an additional prerequisite that was non-existent in previous algorithms: The algorithm assumes the spanning tree to be static throughout the detection of termination detection. That is, the spanning tree shall not change during the process of detecting termination. By doing so, the approach minimizes the amount of time required for the parent of the static tree to exchange messages with any other node in the system. This is an important paper as our approach also considers the detection delay to be the most important metric to consider when designing a distributed termination detection algorithm.

2.3.4 Credit Recovery

Another elegant approach to the DTD problem is the Credit/Recovery approach. In this approach, all active nodes hold a portion of an imaginary weight called a 'credit'. As a node transitions from an active phase to a passive phase, it relinquishes its portion of the credit to a nearby node that is still active. Termination is detected once we have a situation where a single node, which has 100% of the original credit, becomes passive. We can see that this correctly detects termination because the only way the last node could possibly have 100% of the original credit is in the scenario where every other node has become passive. So, once this final node becomes passive, we have a situation where every node in the system has become passive. Assuming that all links are empty, this would indicate that termination has occurred and it would have been successfully detected by the approach. This approach is interesting as it was originally derived from garbage collection schemes. This shows us that many problems in distributed systems are related. In some cases, solutions to certain problems are applicable as solutions to other problems.

In [32], Gerard Tel and Friedemann Mattern present a series of different distributed termination detection algorithms which have been derived from various garbage collection schemes. This paper serves as a prime example of how the average garbage collection scheme can be modified to answer the DTD problem in the form of a Credit/Recovery approach. For our purposes, garbage collection simply means the automatic deallocation of certain memory registers at appropriate times. Essentially, they present the argument that garbage collection schemes and the distributed termination detection problem share many similarities. They share enough similarities to the point that the authors are able to present the distributed termination detection problem as an example of a garbage collection problem. Given this, they argue, and show, that solutions to the garbage collection problem can be modified in order to solve the distributed termination detection problem. They go on to show a series of transformations from garbage collection schemes to solutions to the distributed termination detection problem. Interestingly enough, through these transformations, they essentially re-create some already popular termination detection algorithms, as well as new algorithms altogether. So, once again, we can see the feasibility of altering solutions for given problems into solutions to the termination detection problem.

This gives us optimism to the assumption that current solutions can be further altered in order to achieve better performance.

In [33], S. Venkatesan writes about the distributed termination detection problem in the presence of node failures. Here, we have another example of an elegant credit/recovery based solution to the DTD problem. However, the approach presented in the paper assumes that the underlying topology is formed as a spanning tree, with a distinguished leader node at the root of the tree. At first, he presents an algorithm that is able to operate in the presence of at most one node failure. Then, once he describes this algorithm, he extends it to operate in the presence of multiple node failures. This is an important paper in the sense that it was one of the first to deal with the distributed termination detection problem in the presence of some type of failure. Further, it shows the process of extending a solution to successfully operate in an environment with one node failure to an environment with multiple node failures. To conclude, Venkatesan also shows some approaches to maintain optimal message complexity in the presence of failures. Certainly, these points shall serve as important sources of knowledge in the development of our approach. The algorithm itself is interesting as it combines some aspects of the credit/recovery approach with the parental responsibility approach. This once again shows us that optimal results can be achieved when approaches are combined in some scenarios.

2.3.5 Auxiliary Papers

Now, we shall observe some auxiliary papers on the subject of distributed termination detection. These papers are included as they often show us transformations to existing problems, or existing network conditions, that are able to modify a given algorithm for the better. Further, we have also included some important papers on some concepts of distributed systems, without reference to the DTD problem. We feel that these concepts, although not directly related to the problem at hand, certainly can play a role in devising a strategy to solve the problem in some way. Also, we include some algorithms that simply do not fit in the above categorizations, but still prove to be interesting approaches.

In [7], K.M. Chandy and L. Lamport present a very important paper in regards to the detection of global states in a given distributed system. Specifically, they introduce the concept of the distributed snapshot, which is used to give a detailed glimpse of the total state of the entire distributed system at any given time. Other than the fact that this paper is a prime example of a landmark paper, it is particularly important to our discussion regarding the termination of a distributed system. Essentially, the argument presented is that termination is an example of a stable property: A property that remains true once it has become true. Recall the definition of termination: All links are empty and all nodes have become idle/passive. Through the use of global snapshots, we can accurately determine whether nodes are active/passive and whether links contain messages or not, at any given snapshot. In terms of general approaches to the distributed termination detection problem, this idea serves as a very important base in discovering the termination status of a given system without utilizing any specific details about the computation in question. Although this really is quite the opposite of the approach we have taken in our paper, the concept of stable properties and distributed snapshots are important to solving the distributed termination detection problem in general.

In [8], we have an early landmark paper by K.M. Chandy and J. Misra. This paper, although it does not specifically deal with termination detection, it does deal with the similar issue of deadlock detection. As mentioned previously, like termination detection, the detection of deadlock is considered a stable property, which is a property that remains true after the moment it has become true and shall not revert to a false state. The algorithms assume that there is no global controller and that messages are received in a standard FIFO fashion. This paper served as an important source for the creation of our

approach since it helped to describe the difficulties encountered in producing such an algorithm. This concept of a stable property is also important to the main approach presented in our paper as we consider the local termination status of nodes to be a stable property.

In [18], Jeff Matocha presents a step-by-step conversion of a distributed termination detection algorithm from a "normal" environment to a mobile wireless network environment. The algorithm he chose for conversion was the previously mentioned Dijkstra-Scholten algorithm. Essentially, Matocha transforms the Dijkstra-Scholten algorithm directly to a wireless environment and then performs a series of enhancements in order to perform better in a wireless environment. Although it is not an entirely original algorithm, it shows us that the Dijkstra-Scholten algorithm, as well as distributed termination detection algorithms in general, can be altered in order to adapt to different environments and they can also be altered to perform better in different environments. This is important as the main algorithm presented in our paper is essentially a transformation of the Dijkstra-Scholten algorithm.

In [4], Bertsekas et al. present a discussion on the convergence rate and termination of asynchronous, iterative computations. Here, we are specifically concerned with the discussion on termination. Essentially, they come to the conclusion that the termination conditions of iterative computations can be defined somewhat generally. That is, termination happens when there is no message in transit and an update by a given node i presents no change in the value x_i, which is equivalent to the value x at the node i. We notice immediately that this is a very similar definition to the one explained in the previous section. This allows the use of any popular distributed termination detection algorithm to be used in order to detect termination. This is important as our main approach presented in this paper is dependent on being able to define the termination properties of specific computations in a simple, generalized manner.

In [30], S.A. Savari and D.P. Bertsekas present an interesting discussion on the concept of asynchronous iterative algorithms. Specifically, we are concerned with their discussion on the termination properties of these types of algorithms. This paper is essentially an extension of paper [4], which was also written by D.P. Bertsekas. The authors use the original algorithm in the original paper and present several different approaches in order to extend this algorithm in order to better handle the ambiguity of local termination. Once again, this serves as a good example of how distributed termination detection problems are easily modifiable to perform better under certain circumstances. This also gives us a good glimpse of the type of modifications necessary to extend an algorithm to different circumstances.

In [9], D.M. Dhamdhere et al. present a discussion on distributed termination detection in the presence of a dynamic environment. To this point, we have been discussing solutions to the DTD problem in the presence of a static distributed environment, where a static environment can be described as an environment where the processes to be used in the computation are known from the start of the computation and no additional processes may join the computation in progress. Dhamdhere et al. instead focus on a distributed termination detection solution that allows processes to join and leave a process dynamically throughout the lifespan of the computation. Obviously, given a dynamically changing environment, this forces us to think about an entire new set of different factors. Specifically, the main issue is that we shall not be entirely sure as to the total amount of processes being executed upon at any given time. This paper does a fine job in explaining these peculiarities and also explaining the strategies used to properly handle these peculiarities.

In [21], Friedemann Mattern once again presents another distributed termination detection algorithm. This time, the algorithm is to be used in an asynchronous environment in which messages

are not guaranteed to be received/processed in the order they are sent. That is, the message channels are non-FIFO in nature. The algorithm presented, the Vector algorithm, utilizes vectors to retain local information about the underlying system. By doing so, Friedemann is able to create an algorithm that is quickly able to detect termination while also preserving an ideal worst-case bound in terms of message complexity. Further, through his tests, Mattern has found that in an average scenario, the Vector algorithm requires much less messages than this worst-case complexity would indicate. However, these complexities are reliant on the underlying topology of the system in question.

Chapter 3
Specific Computations and their Termination Properties

In this section, we shall observe the termination properties of a few specific distributed computations. These have been chosen in such a manner that all of the algorithms perform somewhat differently throughout the life of the computation. We shall observe a couple distributed query computations, a leader election computation, a few breadth-first search computations, an edge-coloring algorithm and three consensus computations. We shall only discuss the algorithms in the amount of detail necessary to observe their termination properties. Hopefully, we can quantify these properties to the point that they give us a clue as how to efficiently detect termination in general.

3.1 Distributed Database Query

In [11], Yahiko Kambayashi et al. discuss a specialized strategy to process database queries in a distributed environment. Essentially, their goal here is to use the concept of "generalized semi-joins" in order to handle the issue of cyclic queries. First, they add the necessary attributes to the given query in order to give it a tree structure. Second, now that the query has a tree structure, they apply the semi-join procedure. The semi-join procedure consists of nodes combining records with each other across direct links.

In [31], Dan Stefanescu et al. present a discussion on the evaluation of generalized path queries. Specifically, they present a discussion on detecting the termination of their algorithm. It is interesting here to note that this is an example of an algorithm that does not have a specific notion of final tasks. To solve this, they employ a simple timing mechanism combined with the values of specific boolean variables in order to detect termination. This is quite similar to many of the popular general detection algorithms presented over the last thirty years.

This is an example of a computation that is quite difficult to predict. The computation depends on the properties of the original query and this makes it difficult to guess how the algorithm will perform its function. We could assume that each node can be considered terminated once it has finished its work. The difficulty here is in accurately capturing the notion of its final work. Additionally, the termination properties of this type of computation must be considered carefully as these properties differ slightly from node to node.

3.2 Leader Election

The goal of a leader election algorithm is to elect a single node, out of any of the nodes present in the computation, to be a leader node. This leader node would then serve as a centralized source to which all other nodes report. This is good example of a computation that finds much use in a distributed environment. In [12], King et al. present a leader election algorithm with scalability in mind. In the algorithm, each node exchanges a series of messages to a group of neighboring nodes. The final step is for the nodes to count the amount of specialized response messages received in the final round. The node that had the majority of its messages responded to is elected the leader. Once this final step is complete, the algorithm is considered terminated.

This algorithm performs much in terms of computation with each node entering many active phases throughout the lifespan of the algorithm. In terms of termination, we can see that the algorithm

is satisfied by the general definition of termination. That is, the computation shall be considered terminated once all links have become empty and all nodes have become passive. Further, we can note a distinct final step. That is, each node must elect a leader. Once this is done, a node does no further 'important' processing.

3.3 Breadth-First Search

Breadth-first search algorithms are used for many reasons: Testing for bipartiteness, finding the shortest path between a pair of nodes, and finding all nodes of a single component to name a few. In terms of distributed systems, finding the shortest path, or more importantly the smallest delay, between nodes, is of great importance. In [5], Aydin Buluc et al. present a few breadth-first search algorithms, both of a serial and parallel nature, to be executed on distributed memory systems. The algorithms generally proceed as such. First, a node is chosen to be the source node of the computation. The computation begins at this source node, labels it as visited, and inspects all nodes with which it is directly connected. Then, we label each of these connected nodes which we are inspecting and repeat the process for every node we have not yet visited. We proceed in this manner until all nodes have been visited. If a node is labeled as visited, it shall simply ignore any incoming messages related to the algorithm.

Here, the termination properties of individual nodes is quite simple to distinguish. In fact, all nodes used in the computation shall observe similar termination conditions. Whenever a given node is labeled as visited, we know that it has no more *important* tasks to consider. That is, we should not need to visit any node labeled as visited. Furthermore, any messages related to the computation shall simply be ignored on any node that has been labeled as visited. Once again, we see that there is a distinct notion of a final task.

3.4 Edge Coloring

Simply put, the edge coloring problem is to color every edge/link of a given graph so that no two adjacent edges have the same color. In [17], Madhav Marathe et al. present an experimental study of an intentionally simple edge-coloring distributed graph algorithm. The algorithm proceeds, round by round, by tentatively issuing a color to every link in the system in parallel. For every pair of adjacent links that share the same color, the colors issued are revoked and these links continue to operate into the next round. However, if a given link is found to not share its color with any neighboring nodes, this link ends its computation and does not proceed to the next round.

Here, we yet again see an obvious notion of a final task. Once a link chooses an acceptable color, it terminates, not to be operated upon any further. However, unlike the other computation categories in this section, we are dealing with the final task of a link, rather than the final task of a node. This shall slightly muddle the process of definining the final task of the nodes for this type of algorithm; however, it shall not muddle the execution of the detection approach in any way.

3.5 Consensus

The general goal of consensus algorithms is for nodes to examine values and decide on a universal value across all nodes. In [27], Phillipe Parvedy et al. present a consensus algorithm to be used under a synchronous but not fault-free environment. In this algorithm, each node continually sends messages until a value is agreed upon. The termination requirements here are that all fault-free

nodes eventually terminate. This is reflected through the algorithm as each process that does not fail eventually decides on a value and terminates.

In [6], Bogdan Chlebus and Dariusz Kowalski present a randomized consensus algorithm. Once again, this algorithm is to be used in a synchronous but not fault-free environment. In this algorithm, we proceed by rounds with each node sending to different nodes at each round. At the final step, each node that has not faulted decides on a value and terminates.

In [1], James Aspnes presents a few randomized protocols for consensus. However, he presents protocols to be used in an asynchronous environment rather than a synchronous environment. Like the previous two consensus algorithms, these protocols consist of much message passing between nodes followed by a decision process, after which the protocols terminate. However, unlike the other two algorithms, we are dealing with asynchronous behavior. When dealing with this asynchronous behavior, it is more difficult to determine exactly how the algorithm performs in terms of how many times each node enters an active state.

Although these consensus algorithms have somewhat different methods of computation from the other presented algorithms, the termination behavior is very much the same. Essentially, we are seeing that we can consolidate an entire algorithm's work into its final task. That is, it does not matter what the computation's goal is, we just need to know *how* it terminates. Further, we note that although the computations vary widely in terms of node usage, the termination properties are somewhat similar. That is, each node only terminates after all of its work has been completed. We shall use this concept in developing our our new definition below.

3.6 New Termination Definition

All of the approaches discussed in the related work section assume the following definition of termination: The system can be considered terminated once all nodes have become passive and all links between all nodes are empty. Can we redefine this definition using the information from these individual computations?

From the above discussions, we have seen that an individual node can be considered terminated only after all of its 'important work' has been completed. How can this be translated to the general definition of termination? Consider the following scenario. We have a system of nodes executing some kind of distributed computation. If a single node has completed all of its work, we know that it cannot be expecting any more important messages. Therefore, if all nodes in the computation have completed their work, then it must be the case that there are no messages related to the computation active within the system. Also, since each node has completed all of its work, it must also be the case that the node has become passive. Therefore, we can redefine the notion of termination as follows: A system can be considered terminated, as it relates to a given computation, if all nodes used in the computation has completed its work as it relates to the computation in question. Given that we accurately define the notion of a node's final work for a given computation, this new definition is equivalent to the generally accepted version of termination detection.

This new, but equivalent, definition of termination is the key to the entire paper. Our goal is to approach the problem with a different strategy that has been employed by algorithms in the past. By doing so, we hope to find a unique solution to the distributed termination detection problem.

Chapter 4
Initial Sequence of Termination Detection Algorithms

Before discussing our sequence of algorithms, let us observe the notation to be used in the definitions.

4.1 Notation

Our algorithms shall operate on a system S. This system shall consist of a set of n nodes. The set U consists of all nodes which actually enter an active state throughout the lifespan of the computation. Note that U is a subset of S.

Before delving into the definitions, we must make a distinction here about relays. Certainly, a node which is used as a relay may receive a message, do a very small amount of work in finding the next link on which to send the message, and then send the message. Here, we can see that the node becomes *active*; however, for the sake of our definition, we shall term the functionality of a relay to be *relay active*. That is, a node that performs some amount of substantial work is to be considered *active*. However, a node that simply acts as a relay is only to be considered *relay active*.

Note also that labeling a node as terminated does not mean that the node is incapable of performing additional work. It is simply a change of description in terms of the node's status. Therefore, even though a node is labeled as terminated, it may still act as a relay or perform other local functions not related to the underlying computation in question.

4.2 Naive Algorithm

Since we are attempting to arrive at a simpler solution for the problem of general termination, let's start with a somewhat naive but simple algorithm.

For each node in the set U: Terminate directly after reaching a passive state from an active state. Once terminated, inform a leader node of your termination status.(1)

It is easy to see that termination holds here. If a node is used whatsoever for a given computation, it shall terminate after it performs some amount of work. Note here that we only check the nodes in the set U, the set of nodes that are used during the computation. That is, we are not concerned with nodes that never enter an *active* state throughout the lifespan of the computation. Obviously, this algorithm has some flaws. If a node has more than one period where it becomes active, the algorithm simply does not work. The node will be labeled as terminated and the behavior is undefined when a terminated node receives additional input from the computation in question. However, consider some imaginary set of distributed computations where each node used by the algorithm is used exactly once. Let’s also assume that we are dealing with a perfect environment. That is, we are dealing with a synchronous system with no link or node failures. Our naive algorithm is unmatched in terms of early termination for this situation. Obviously, if each node terminates directly after its only period in an active state, this equates to the least amount of up-time for each node, leading to the earliest termination time of the overall system. This leads us to believe that while this algorithm is quite flawed, there is hope that the idea is worth investigating.

What can we do to improve this naive algorithm? Obviously, in terms of a single node, we cannot just terminate after one period of activity. Many algorithms, e.g., consensus algorithms, require that each node repeatedly send to its neighboring nodes throughout the computation. However, what if we knew exactly how many active states each node will encounter? Surely this is computable for some set of algorithms in fault-less, synchronous conditions. If we had this information, a better algorithm could be defined as follows.

4.3 Smarter Algorithm

For each node in the set U: Terminate directly after all active periods of computing have completed. Once terminated, inform a leader node of your termination status.(2)

Given a synchronous system with no failures, it is easy to see that termination holds for algorithm (2). Each node simply does all of its required work and then terminates. Since there are no failures and all rounds proceed in a lock-step manner, there shall be no issues with nodes entering fewer or more active periods throughout the computation.

Similar to our naive algorithm, algorithm two has a fast time in terms of complete termination across all nodes used in the computation. The argument is similar in that each node shall terminate directly after it has completed all of its work. We shall have an additional comparison each time a node becomes active, however. Essentially, we have taken the naive algorithm and extended it to work on a larger subset of distributed computations.

However, we still have issues to resolve. First, certainly there are many algorithms for which it would be impossible to ascertain how many times a node becomes active throughout the lifespan of the computation. Secondly, when faults are considered, the amount of times a node becomes active throughout a computation could change depending on the particular algorithm's handling of faults. So, even if we did have this information, the transition from a perfect synchronous system to a faulty asynchronous system is quite complex.

4.4 Statistics-Based Algorithm

This introduces a slight change to our approach. Instead of relying on explicit knowledge of how many times a node becomes active, we shall rely on statistical input. That is, we should simulate a given algorithm on a given set of nodes with a simulated asynchronous environment. Obviously, we still cannot just terminate depending on the values returned by these simulations; however, they can give us a decent approximation as to how many times each node in the system becomes active. These simulations could also give us approximations as to how long each node waits between active states, including the maximum and minimum amount of delay each node experiences. Define $count_i$ to be the amount of times a node i becomes active, inferred from the statistical data. Define max_i to be the maximum amount of time node i has to wait between any two periods of activity.

Given this statistical data, and a system that implements timers local to each node, we can develop a more sophisticated algorithm.

Increment a local variable count, initially 0, each time a node becomes active from a passive state. Start a timer from 0 each time a node enters a passive state. For each node in the set U: Terminate after this local count variable becomes equal to $count_i$ * m or terminate when the timer reaches max_i * n. Once terminated, inform a leader node of your termination status.(3)

Although the reliance on counters and timing constraints should be considered a negative, it is worth mentioning that Friedemann Mattern included counters in his famous 4-counter method [20].

Obviously, every node shall be guaranteed to terminate as the timer will eventually reach the desired limit if the local count has not already met the desired value. The issue here is trying to find the balance between guaranteed algorithm correctness and efficiency. Obviously, if we choose multipliers (m and n) that are too small, the nodes might terminate before the algorithm finishes. If we choose multipliers that are too large, we begin to lose efficiency. However, as technology continues to grow and faults become less frequent, this method might be worthy of consideration.

4.5 Final Work Algorithm

Consider algorithm (2) for a moment. This algorithm is guaranteed to work 100% of the time if we know exactly how many times a node shall become active throughout a distributed computation when working in a perfect environment. In addition, this algorithm would also produce excellent complexities in terms of time and message sending. We have mentioned time efficiency previously, and we can also see that (2) adds no additional messages to the algorithm's message complexity. The implausible part here is the fact that we need to know too much about the computation in question. This leads us to our final, non-general approach for this paper.

Essentially, what if we could impress the given computation's "final work" onto each node locally? Rather than knowing how many times each node enters an active state, we instead just observe the final action of each node. From our research on simple computations from the previous chapter, we have seen that the termination properties of this set of algorithms are very similar. We should use this to our advantage and this is done in the final algorithm below.

Given a distributed computation, impress upon every node in the set U the "final work" of the computation. For each node in the set U: Terminate directly after this "final work" has been completed. Once terminated, inform a leader node of your termination status.(4)

As long as a given node correctly identifies its own "final work" and this node does not crash, the node is guaranteed to terminate by algorithm (4).

The strengths of the approach here are obvious. Given a fault-free environment, each node used would be guaranteed to terminate, leading to the overall termination of the algorithm. In addition, we still have excellent message and time complexity for the same reasons given previously.

There are still two major weaknesses with this algorithm. First, it is not always possible to determine the "final work" of each node for a given type of computation. To be certain, there are distributed computations where it shall be impossible to define the final work of the algorithm, let alone the final work of each individual node. Second, the ability to impress the final work of a computation node-by-node is difficult to translate to the general sense. Essentially, the computations themselves would have to contain this information in order for (4) to remain a "general" termination algorithm.

In addition to these points, we also lose the sense of generality with such an approach. Essentially, we would just be creating a sort of wrapper function that detects termination based on which computation is being performed. The end result shall be a solution that encompasses all

computations. Therefore, the solution is general; however, the approach taken requires information from each computation covered by the solution, and is therefore a non-general approach. However, this "wrapper function" approach also gives us an opportunity to retain the most efficient general termination algorithms to date. That is, in the case of a computation that cannot be defined under the Final Work approach, we can simply utilize the most popular of current general termination algorithms in our wrapper function. Therefore, we would see improved efficiency for computations for which we can properly define the Final Work approach, and still experience previously enjoyed efficiency for computations for which we cannot properly define under the Final Work paradigm. Due to this fact, we believe that the approach is worthy of further investigation.

Next, we shall select a few families of distributed computations and attempt to ascertain the final work properties of these algorithms.

Chapter 5
The Final Work Approach

How do we ascertain the final work of an algorithm? Certainly this is a simple task for many types of algorithms. However, there are sets of algorithms that would be more difficult to reason about. Specifically, sorting algorithms serve as an example of computations that have different types of ending behavior when considering the behavior of each node used in the computation. Throughout this section, we shall attempt to categorize different computations in such a way that they are easily covered by our Final Work approach. Essentially, we would like to make as few categories as possible to keep the generality of the approach intact without jeopardizing the effectiveness of the approach. However, we might find computations that do not have any specific final work, especially when we consider that each node used in the computation may terminate differently.

At this point, it is important that we realize that in terms of the final work of a given algorithm, it might be the case that the nodes used in the computation do not all necessarily end their work with the same exact task as every other node used in the computation. For instance, as we shall see when we discuss leader election computations, there shall be two types of nodes used in the computation: The eventual leader node as well as the remaining non-leader nodes. As we shall see, we can use some intuition to easily handle this issue in terms of leader election computations. However, this example shows that in situations such as these, we must be careful to correctly define the final work for both types of nodes so that the approach may correctly identify the final work of any given node used in the original computation.

Before looking at the specific final work conditions of the previously described computations, let us consider the real strength of the Final Work approach. The real strength of the approach lies in the manner in which it has been developed. As long as we are able to accurately define the notion of final work for a given computation, we are guaranteed that the termination conditions shall be accurately identified. That is, the difficulty is abstracted to the task of identifying the final work of each different type of computation. Given these final work definitions, the Final Work approach shall always operate in the exact same manner, regardless of what type of underlying computation is being executed.

5.1 Leader Election

Let us first consider leader-election algorithms. We already know that this set of algorithms can be grouped together because of the nature of the final work that each node performs. Let us first consider the nodes which are not elected as the leader. These nodes shall essentially label themselves as non-leader once they realize they cannot possibly be the leader. So, most leader election algorithms simply assign a variable to *non-leader* or something similar. Even if this is not done precisely as described, the algorithms could be altered slightly to make it easier to categorize them in a similar grouping. That is, say we have a leader election algorithm that does not go to the trouble of labeling a node as non-leader (perhaps this is not as important as simply finding the leader). This particular algorithm could simply be changed to do this variable assignment described previously in order for each non-leader node to follow a similar definition of final work. This shall add to the space complexity only slightly by adding an extra variable at each node. The time complexity shall not change. Alternatively, a different version of leader-election could be chosen such that it better fits our general perception of leader-election computations, as discussed in chapter four. We shall also associate a *type* variable with each algorithm capable of running on the given system of nodes. Since we are

dealing with only leader election computations here, they shall all have this field labeled as *leader election*.

Now let us consider the node that is elected leader. This node shall have a slightly different but ultimately similar notion of final work. Essentially, it shall assign to the same variable; however, it shall obviously assign something similar to *leader* rather than *non-leader* as we had before. Still, this behavior is similar enough that for the sake of detection we really do not have to distinguish between what is assigned to the variable. We simply need to see that the variable was assigned a value.
Note that in an asynchronous environment, some leader-election algorithms give the leader node the responsibility of informing other nodes that it is the leader. In this case, the definition of final work shall be slightly different. First, in terms of the leader node, its final work shall be defined as the sending of these informing messages to other nodes in the system. In terms of non-leader nodes, their final work shall instead be the reception of these messages individually. That is, once a node receives one of these leader declaration messages, it shall be labeled as terminated and act as an active relay from that point forward. The key here is to fully understand the given computation and define its termination properties accurately.

Now, assume that our detection algorithm is running on top of a given leader election algorithm. The detection algorithm shall note that the algorithm being executed falls into the leader election category by observing the *type* variable. Each non-leader node that exits the computation shall assign the status variable to *non-leader*. The single leader node shall assign the status variable to *leader*. Our detection algorithm shall see, node by node, that these nodes have performed their final work. Noting this, each node shall label itself as terminated as this final work is performed. As each node is labeled as terminated, one of the informing processes of chapter seven shall begin locally at each node.

Let us discuss the notion of *active relays* once again. As we have seen, these pose some issues given our definition of active nodes as compared to passive nodes. Obviously, if a node must act as a relay, it still might be entering active states until all of the algorithm's work has ended, even if the node is labeled as terminated. Here, we shall do some hand-waving to avoid this issue. Consider the definition of final work. If the final work has been observed on a particular node, we know that we should not be receiving any *important* computing tasks for this node throughout the remainder of the computation (we could still receive relay messages on nodes which have terminated). If all nodes have achieved their final work, we know for a fact that all *important* messages must have been received. That is, there are no more messages being sent within the system. So, while an individual node might be labeled as terminated, it does not necessarily mean that the node itself has finished entering active states. It could still act as a relay. As we shall see in chapter seven, the key is not to terminate individual nodes, but rather to label them as terminated for the informing portion of the algorithm.

5.2 Consensus

Next, let us consider another somewhat easy set of algorithms. Consensus algorithms, as described in chapter three, all follow a somewhat similar route from the algorithm's beginning to the algorithm's end. Here, the final work shall be chosen as such. Each node's end goal throughout the course of the algorithm is to simply choose a value. Hopefully, if the consensus algorithm is correct, these values all match or fall within some chosen bound. So, similar to the leader election set's final work approach, the consensus set's approach shall also look for the assigning of the given decision variable, *decision*. Once *decision* has been assigned by a given node, we know that the important work for that node is now complete. Also, assign the *type* variable to *consensus* for any algorithms that fall under this set.

Note here that we might also have relay nodes active after being labeled as terminated; however, due to the same arguments given in the previous section, we can see that these shall not present us with any issues. Also note here that unlike the leader election algorithms we do not have to distinguish between nodes. Most algorithms shall treat all of the nodes in a similar manner. That is, the goal of each node in the computation is to find a decision value.

So, assume that our detection algorithm is now running on top of a given consensus algorithm that is covered by the algorithm itself. Once again, the detection algorithm shall note that the *type* variable is set to *consensus*. Seeing this, it shall decide that the following computation to be run is a consensus computation. Given that we are dealing with a perfect system, eventually, each working node in the system shall set its decision variable. As each node decides its value, it shall label itself as terminated. Once all nodes have been labeled as terminated, the informing process of chapter seven shall take control of the detection algorithm.

5.3 Breadth-First Search

Breadth-first search computations shall also be easy to define under the Final Work approach. Initially, all nodes in the system are labeled as unvisited. As these nodes are visited, a certain variable, let us assume it is called *visited*, has its value changed from false to true. As explained in section three, these visited nodes shall no longer be responsible for any important tasks. So, when the *visited* variable experiences a value change, we know the given node can be labeled as terminated. For any computation that follows this general procedure, we shall issue the value of *breadth-first search* to its *type* variable.

So, very similar to both the leader election and consensus groups of algorithms, we can simplify the final work of every node in the system to be the simple change of a given variable. Once again, the detection algorithm shall take note of the type of algorithm being run. It shall do so by reading the *type* variable and seeing that we are dealing with a *breadth-first search* type algorithm. So, like the other two algorithm groups, our Final Work algorithm, running locally at each node, shall observe the alteration of the target variable, in this case *visited*, and in turn shall label its host node as terminated. As each individual node is labeled as terminated, they shall each begin the informing process of chapter seven.

5.4 Edge Coloring

Although the edge-coloring problem is more of a thoretical problem than a realistic distributed problem, it is worth mentioning as it commands a slightly more involved procedure in terms of defininig the final work of the individual nodes.

As ascertained in chapter four, we have a situation where the links, rather than the nodes, exhibit behavior with respect to the termination of the algorithm as a whole. That is, the status of the links, rather than the nodes, offer us the concept of final work.

However, we must recall that every link in a given system has nodes attached. As far as we are concerned, there exists no link that is not part of an overall graph, connected by a node at each end. Since this is the case, and since it is actually the nodes that are responsible for any logic concerning the computation, we can still correctly define the final work in terms of the nodes. Let us assume again that any algorithm that is defined to be an edge-coloring algorithm shall have the *type* variable set to *edge-*

coloring.

Once a color is chosen, and it is found to not experience any sort of color collision with another adjacent edge, the color of the edge is finalized. Note that since every edge connects exactly two nodes, both connected nodes must be informed of the final color selection for the given edge. So, in order for a node to experience its final task, it is obvious that it must properly color all of its edges with a final color selection. Given the definition of the edge-coloring algorithm, we know that all important tasks, such as color inspecting/manipulation, shall cease upon the final edge receiving a final color. So, we would assume the final work for this subset of computations could be defined to be the point when all edges on a node's edge list has received a final color. However, for some edge-coloring algorithms, this would prove to be a premature definition of termination given our loose understanding of edge-coloring algorithms in general. So, for now, keep this definition in mind and we shall come back to redefine it once we have further inspected the algorithm below. However, in the case of a computation that incorporates no such overall termination procedure, this definition shall be adequate.

Recall, that in some edge-coloring algorithms, we have another algorithm course to consider. That is, if the edge-coloring computation finds that there are not an adequate amount of colors in the color pallet for a given node, the algorithm as a whole shall experience a failed conclusion. Here, we must consider a secondary definition of final work. For a given node, the computation shall repeatedly try different colors until the pallet is exhausted. This shall be a legitimate definition of final work. That is, once this pallet has been exhausted, we know that the node in question, after perhaps sending abort messages to the other nodes in the computation, shall no longer examine further messages as it has ultimately decided the fate of the entire computation.

Now, let us return to our first, premature definition of final work. We can see that a node must realize that all other nodes have experienced a successful coloring of all connected edges before it can officially label itself as terminated. Otherwise, a given node shall have no knowledge of the overall termination status of a given computation. So, the initial final work definition shall instead be when a node has received a success message from every other node in the system.

This change in definition actually helps us to refine the two final work possibilities into a similar category. That is, the termination of a specific node can be wholly defined as follows: Label a node for termination if it receives an abort message from any node in the computation OR if it receives a success message from every node in the system and it itself has successfully colored all edges.
So, assume that our detection algorithm is running on top of a given edge-coloring computation. First, it shall inspect the *type* variable and note that the computation is of the type *edge-coloring*. The nodes, one by one, shall label themselves as terminated through the two possible final work possibilities. Upon being labeled as terminated, each node shall locally begin the informing process of chapter seven.
In the case of edge-coloring algorithms, we can see how vital it is to correctly define the final work of each node in the computation. If we had only considered our premature definition, we would have a situation where nodes might be labeled as terminated when in actuality they still have one or more important tasks to consider. However, as noted previously, the complexity of the approach has been intentionally abstracted to defining the final work properties of the given computation. This shall lead to an easier, less-involved implementation of the actual detection algorithm.

5.5 Distributed Query Processing

As discussed in chapter four, it is quite difficult to express the notion of final work in terms of distributed query processing. In [31], the authors discuss the difficulty of detecting the termination of

their algorithm and simply describe a popular distributed termination detection algorithm, or slight variation thereof, to be used instead of a concrete definition of termination. Similarly, in [11], the authors do not give a concrete notion of the final work of the individual nodes.

This is an example of a type of algorithm that is too difficult, perhaps impossible, to capture the notion of final work from node to node. So, as mentioned previously, we would simply use a popular termination detection algorithm in place of the final work approach. By doing so, we still have the means to detect termination with the same efficiencies we would have experienced otherwise.

5.6 General Concerns and Ideas

We have thus far been using intentionally simple examples of distributed computations. Many of these computations exhibit very similar notions of final work. That is, for a given node, there seems to be a change in value of one or more variables and then the node can be considered to have finished its important tasks. However, we must realize that often algorithms do more than this simple value change followed by an abrupt termination. Often, a given node shall also be responsible for smaller tasks, such as sending messages to other nodes in order to update neighboring nodes of the given node's status change. We saw that this is a possibility for both leader-election and edge-coloring types of algorithms. We declare, for the sake of feasibility, that situations such as these can still be handled by our Final Work approach with the following solution.

Given this situation, we have two options. The first solution would be to alter the actual computation in order to better fit the Final Work approach. This is undesirable as altering the actual computation should always be a last resort due to the complex issues it shall introduce. The second solution would be to simply alter the definition of final work for the given computation. Considering the leader-election example, we could simply change the final work to be the sending/reception of these final messages rather than the value change of a target variable. This is the strength of the Final Work approach as we are guaranteed excellent efficiencies as well as dependable functionality as long as the final work of the nodes of a given computation can be accurately defined.

The sometimes muddled and multiple definitions of final work also present a concern. For some computations, we can see that different versions of final work are necessary to match the slight differences in the algorithms themselves. For example, we have two broad notions of final work for leader-election algorithms depending on whether or not the given leader-election algorithm has the responsibility of informing all other nodes in the computation of the leader's UID. Although, as mentioned previously, this complexity in terms of defining the final work was expected. This is simply an unfortunate side effect due to the abstraction of detection algorithm complexity to the individual final work definitions rather than having the complexity present in the detection algorithm itself. That is, if we can successfully and accurately define these final work concepts for a given computation, we can guarantee excellent efficiencies as well as algorithm correctness.

However, the previously mentioned process of informing all nodes of a given node's status invokes other thoughts in terms of our Final Work approach. In our small list of algorithm types, we saw that both leader-election and edge-coloring algorithms can have this sort of informing process. This presents us with another potential category that could possibly fit many wildly different types of computation under one broad definition of final work. That is, the final work for these types of algorithms could be defined as the sending of these final informing messages, or as the reception of these final informing messages, or the combination of both. This way, we could create an extremely broad group of computations that all operate under the same notion of final work. To be certain, this

group could consist of wildly different types of computation (i.e. edge-coloring as compared to leader-election) that are all joined by their similar notions of final work. To be sure, this would greatly reduce the complexity of the overall grouping process.

Chapter 6
Final Work Approach: Specific Mechanics

Now that we have developed the final work concept for a moderate amount of different distributed computations, let us actually see how this information can be utilized to solve the distributed termination detection problem.

6.1 Individual Node Termination

Now, we have thrown around the notion of terminating a node many times to this point. To be certain, we are not talking about ending a node. That is, we are not insinuating that once a node is labeled as terminated, that it can no longer perform any task. As discussed previously, the node still may act as a relay. Additionally, the node may still perform tasks unrelated to the computation in question (i.e. Tasks for a different computation that happens to be executing at the same time).

So, consider an individual node. Each node has some amount of knowledge of the computation(s) currently being run locally on the node itself. Due to our final work descriptions above, the node can inspect the *type* variable of a computation and decide from what category the final work information belongs. Given this knowledge, the node inspects the final work information for the given computation and creates proper listener objects. For instance, in the case of a consensus type algorithm, the node shall note that it needs to create a listener object for the assignment of some *decision* variable. This shall be done at each node. Then, individually at each node, as this *decision* variable receives a value, the listener shall be alerted and label the node for termination. This would be done by setting a variable, call it *terminationStatus,* local to each node, from false to true. For all intents and purposes, this would then instantaneously trigger the informing process of the next section.

We can imagine similar situations for the other groups of computations. For example, if the type variable of the computation was set to edge-coloring, the node would then create a listener to check for the sending of either an *abort* message or the reception of all possible *success* messages. Once one of these cases has been satisfied, the node shall label itself as terminated and begin the informing process of section seven. The key here is to accurately define the final work of a given computation. Given that the individual computation's final work is defined properly, we can guarantee the success of the Final Work approach.

To be sure, labeling a node for termination is just a formality. To save local node space, the informing process of the upcoming section could begin immediately after the listener has been alerted, thereby bypassing the labeling process. At this point, there is no reason for the node to retain this information as the informing process has been initiated. This might be desirable if local node space is at a premium. However, if operating in an imperfect environment, it may be desirable to perform this variable assignment as a means to keep track of the overall progress in the presence of possible message dropping or node failures.

6.2 Informing Process

Assume that we have categorized all possible different distributed computations in regards to their final work as described above. Assume that this information is stored within each node used by the computation(s) in question. We know that when the given node in the given computation observes

this final work, the node shall label itself as terminated. Recalling our redefined version of termination from above, we know that if each node is successfully labeled as terminated, the entire computation can be considered terminated.

However, is this useful to us by itself? Surely each node knows that it has terminated; however, with no knowledge of any other node's status, no node in the computation can decide whether or not the entire computation has terminated. We shall present solutions to this portion of the problem now. Recall, from the introduction chapter, there were a few distinct types of distributed termination detection algorithms. We shall discuss a few of these and decide in which scenario each would be applicable.

The easiest approach in terms of communication between nodes as well as time efficiency would be to have a single root node which has direct communication with all other nodes used in the computation. Recalling the discussion for the introduction chapter, this approach is called the Invigilator approach. The underlying topology assumed by this approach is a star topology: Where there is a single node, the coordinator, which has direct communication with all other nodes in the system. However, unlike the Invigilator approach as discussed in the introductory chapter, our coordinator would be a dedicated leader. That is, this node would not be used at all during the actual underlying computation. This node would simply serve as an active listener towards all nodes used in the computation. Once a given node observes its final work it shall label itself as terminated as well as sending a "terminated #x" message to the coordinator/listener node, where "#x" is the UID of the node in question. Once the listener node has received these messages from every node used in the computation, it can be decided that the computation has terminated. This essentially follows the same approach as does the famous 2-phase commit protocol. If we are concerned about the non-locking properties of a computation, we could upgrade to the 3-phase commit protocol.

Note, however, there is another subtle, yet important difference in our Invigilator approach as compared to the general Invigilator approach. We are not at all concerned with sending update statuses continually during the lifespan of the underlying computation. Given our Final Work approach, it is not necessary to do such updates. We are only concerned with the moment that a node decides it itself has terminated, and sends a message declaring as much. Until that point, it is simply assumed that the node still has important work to accomplish and has not labeled itself as terminated. This is great in terms of message complexity because local status updates are handled locally by each node in the computation. A node shall require no messages to update itself locally. Therefore, in this scenario, our message complexity is bounded above by O(n), where n is the total number of nodes to be used in the computation. This is calculated by assuming that each of the n nodes creates exactly one message to inform the coordinator node that it has locally terminated.

Similarly, for time delay between termination and termination detection, we shall experience very small delays. In fact, the termination delay shall be bounded above by O(c), where c is a constant equal to the largest time delay from the coordinator node to any other node in the system. Here it is assumed that the coordinator's processing of the incoming termination messages is also constant, and negligible, when compared to the delay experienced by sending across the links. Performance shall be discussed further in the Performance chapter to come.

The major weakness of this approach is simple. It might not be feasible that we have some sort of coordinator node with direct communication with every other node in the system. That is, we very likely will have a topology that is not star-based. So, for our next detection approach, instead of creating a single coordinator node, we should simply elect a single existing node to be the leader node.

Obviously, we would just use one of the many leader election algorithms proven to be effective(Or, to reduce detection delay, we could choose the node such that it is 'near the middle' of a distributed system). Note that this leader election algorithm would then only consider the nodes used in the computation. That is, the leader node would be selected from one of the nodes that actually enter an active state throughout the lifespan of the computation in question.

This second approach, which we shall call the Elected approach, is more akin to the wave-based approach as discussed in the introductory chapter. In this Elected approach, recalling terminology of the wave-based approach, we shall have one 'initiator', called the leader node. Once a leader has been elected, it would be this leader node that all nodes report to during the detection process. Note here, because we have not assumed an underlying topology, it might be the case that we do not have direct communication between the leader node and every other node in the system. If this is the case, local routing algorithms would need to be placed in each node so that the termination messages are able to reach the leader node. Obviously, since these nodes are already being used to run computations, we can safely assume that local routing information already exists within these nodes. So, no additional programming needs to be done in order to construct routes from a terminated node to the leader node.

Similar to our modified Invigilator approach, in the Elected approach, once the leader node has received termination messages from every node used throughout the computation, it shall decide that the computation has terminated. This shall carry out exactly as the Invigilator approach except for the fact that the leader node is not assumed to have direct communication with all other nodes in the system. This shall not affect the mechanics of the approach in any way, but rather affect the complexities instead. This shall be discussed and compared in the Performance section.

Now, in considering the two previously mentioned detection approaches, is it enough that the leader/coordinator node know of the overall termination status of the computation? Or, does every node in the computation need to be informed of the termination status of the computation? If only the leader node needs this information, the previous two approaches shall be satisfactory. However, if it is necessary that every node used in the computation be aware of the termination status, then we have further work to consider.

This additional work can be handled in two ways. First, let us consider the previous two approaches and how to expand upon them in order to accomplish this additional work. Assume all nodes have been labeled as terminated and have sent termination messages to the coordinator/leader node. This node then decides that the algorithm has terminated. The coordinator/leader node shall then inform all necessary nodes that the computation has terminated so that subsequent computations might be executed. Given the first approach, the coordinator node would simply directly send the termination status back to each node in the system. Similarly, given the second approach, and using the local routing information, it is not an issue for the leader node to simply send the termination status back to every node in the system.

However, if our goal is to inform every node in the system of the original computation's termination status, we can also use a flooding algorithm to inform all nodes. That is, once an individual node labels itself as terminated, it shall use a flooding algorithm to distribute its termination status to every other node used in the computation. A flooding algorithm is a somewhat naive algorithm that's goal is for each node in the system to send some bit of information to every other node in the system. This approach, we have dubbed the Flooding approach.

So, each node in the computation, upon termination, sends this termination status to every other

node in the system through a flooding procedure. Upon receiving each termination message(each node would keep a running list of terminated nodes), each node shall check whether or not all nodes have reported this termination status. Once an individual node sees that all nodes used in the computation have been reported as terminated, that node can decide that the original computation has terminated. Given a perfect environment, all nodes shall be guaranteed to correctly discover the termination status.

In comparing the two approaches not dependent on the underlying topology, the Elected and Flooding approaches, we can see that the Flooding approach would be the most efficient in terms of time; however, the Elected approach would be more efficient in terms of message complexity. This is due to the fact that the flooding approach more quickly is able to begin the global informing of the termination status, but must send many more messages in order to do so. Further, if it is enough for a single node to detect the overall termination of the original computation, and if we use the Flooding approach, there is a possibility of a theoretically instantaneous detection of termination. This shall be expanded upon in the performance chapter of this paper.

It is important to note a certain distinction between the Elected and Flooding approaches. The Elected approach is centralized, while the Flooding approach is decentralized. That is, the Elected approach has all information returning back to a central point of information, the leader node. The same can be said of the Invigilator approach. In contrast, the Flooding approach does not have a central point of command. All nodes are weighted equally and informed of the termination status in the exact same manner. Since we are already assuming that we know some information about the underlying computation, the type of underlying computation, centralized or decentralized, can help us to choose the better suited of these detection strategies.

Using one of these approaches, any system of nodes can satisfy the requirements of our general termination algorithm, regardless of connections between given nodes. As long as the nodes are composed in such a way that the original computation is able to execute properly, this composition also guarantees that our final work algorithm shall be able to operate properly over the same set of nodes.

6.3 Overall Detection

We have seen that, for a given computation, we are able to detect when the computation has finished. However, how do we translate this into the goal of the distributed termination detection problem? Recall that we consider two versions of the distributed termination detection problem. The first version is simply defined as detecting when an individual computation has terminated. We have solved this problem. The second version is defined as detecting when an entire system of nodes has completely terminated. That is, all nodes have become passive and all links have become empty. What we have accomplished to this point can be defined as signaling to all nodes the fact that an individual computation has terminated.

So, in order to properly detect the second version of termination, which we'll call system termination, we must do additional work in terms of keeping track of the computations being executed on the given system of nodes. Simply put, each node in the computation should keep track of the computations being executed. We can presume, with a decent amount of certainty, that the nodes already contain some mechanism to support this tracking scheme. Then, as the overall termination of a given computation is detected and each node is informed of this status, the node can remove the computation from its list of executing computations. Once this list has been reduced to an empty state, we can conclude that the system has terminated all computations. Therefore, this would solve the more general problem of detecting system termination.

Note that while this performs adequately in a perfect environment, it should be noted that transition to an imperfect environment must be done with extreme care. Essentially, as is the case with all distributed computations, message dropping and node failures posit a real problem.

6.4 Mutual Exclusion

Consider the reasons for having a general termination algorithm executing on top of a given distributed computation. Essentially, we would like to know when a given computation ends so we might execute subsequent algorithms. However, what about computations that can be executed concurrently? Surely these computations could be executed during the execution of the currently executing computation without worry. Because of this, we shall consider a system of flags for each computation.

For each computation covered by our general termination algorithm, we shall associate a list. This list shall contain all computations that cannot be executed concurrently with the associated computation. As you probably realize, this list resembles that of a black list used in many areas of computer science. Also note the mirror affect of these lists. If algorithm one appears on algorithm two's blacklist, then algorithm two shall also be on algorithm one's blacklist. This makes sense as we do not want the two algorithms running concurrently so each one should have the other blacklisted.

The list algorithm shall work in the following manner. If there are no computations currently executing on the given set of nodes, we shall allow any single computation to begin executing. This computation shall be chosen to effectively increase efficiency: e.g. choosing a computation that can be executed along with many other computations concurrently. If there are computations being executed on the given set of nodes and we would like to begin executing a new computation we shall use the following approach: We check the blacklist of the algorithm we wish to begin executing. If any of the currently executing computations are found on this list, we must stall the execution of the algorithm in question. Obviously, only if there are no executing computations on the list can we begin executing the given algorithm.

Chapter 7
Performance

Now, we shall investigate how well the Final Work approach performs from a performance standpoint. Let us first assume that we have categorized all possible types of algorithms under the final work approach. In the case of algorithms that are not definable under final work, we simply employ a popular/efficient general detection algorithm.

In terms of performance, we have two main metrics with which we should concern ourselves: *Detection Delay* and added *Message Complexity*. These have been discussed in the background chapter. First, and usually considered more important by most authors, we must consider the time delay between achieving termination and the detection of termination globally. Specifically, we are concerned with a single node's detection of global termination as once a single node has made this discovery it can then spread this information to the necessary nodes. Second, we must also consider the message overhead introduced by our DTD algorithm. Hopefully, our DTD algorithm would not add a great deal of extra messages to the execution of the underlying computation.

For defined algorithms, the Final Work approach has no waiting period in terms of waiting for nodes to become inactive. Once each node has detected its local termination, and a single node makes the discovery that all nodes have achieved local termination, we have reached the situation where a single node is aware of the underlying computation's termination. Specifically, the detection delay can be defined as the time between the point where the final node decides it has locally terminated and the point where a single node has discovered the global termination.

7.1 Invigilator Approach

As mentioned previously, we have three different types of informing processes to consider. We shall first discuss the detection delay produced by the Invigilator approach. Recall that the Invigilator approach assumes a star topology. Therefore, we have a situation where the coordinator node has direct contact with all other nodes in the system. What is the worst-case scenario here in terms of detection delay? This scenario would be the case where the node connected by the link with the largest detection delay is the last node to complete its final work. In this case, the detection delay will be equal to the delay on this link. Therefore, even in the worst-case scenario, we experience a constant detection delay, $\Omega(c)$. In fact, in all scenarios, we experience a constant detection delay, $O(c)$.

Now, still considering the Invigilator approach, we shall also have to consider message complexity. Recall that we detect termination once all nodes have reported in to the coordinator node. Since each node is directly connected to the coordinator, we shall only ever create n messages, where n is the amount of nodes used by the computation. Therefore, the message complexity shall be linear in terms of the amount of nodes in the system in all scenarios, $O(n)$.

7.2 Elected Approach

Now, we shall analyze the performance metrics of the Elected approach. Recall that this approach is similar to the Invigilator approach except that it does not assume a star topology. Instead, the approach assumes a given node to be the leader. This leader node shall then be equivalent to the Coordinator node of the Invigilator approach. Therefore, the key difference here is that we do not have

a single link between each node and the leader node. Instead, we could have any number of nodes between a given node and the leader node. How does this affect the detection delay? In general, we can assume that the delay will now be bounded by the diameter of the distributed system, which we shall call D. Simply put, the diameter can be defined as the longest-shortest path between any pair of nodes in the system. Since each link has a constant delay, the overall detection delay is equal to the diameter of the distributed system, O(D).

Recall that with our Final Work approach, we only create a control message once a node locally terminates. So, the added path lengths from each node the the leader node will not add to 'created' control messages. Therefore, the message complexity is still O(n). However, these messages will need to be passed around more than they are passed in the Invigilator approach. Therefore, if we consider a 'new' message to be created each time an existing message is sent by relay, the worst-case would be the diameter times the amount of nodes in the system, $\Omega(D * n)$.

7.3 Flooding Approach

Finally, we shall now analyze the metrics of the Flooding approach. Recall that the Flooding approach 'floods' the termination status of each node to every other node in the system. So, actually, the best case scenario here would be the case when all except one node discovers its local termination, then enough time passes to allow all other nodes to flood their termination information to all other nodes, and then the final node detects its local termination. In this case, the final node shall already know that all other nodes used in the computation have achieved there termination status. Therefore, once it detects its local termination, it can instantaneously realize the global termination detection of the underlying computation! So, in this best case scenario, we have an instantaneous detection delay of O(1). This is a fantastic complexity.

Still considering the Flooding approach, let us consider the worst case scenario. In the worst case, we shall have a situation where all nodes detect their local termination at the same time. Here, with the Flooding approach, our time delay shall be equivalent to the time delay of the longest-shortest path found within the system, the diameter. So, our time delay in the worst case scenario is O(D).

Now, let us consider the message complexity of the Flooding approach. Here, we can see that this shall yield very poor message complexities. In fact, this approach is similar to the wave-based approaches discussed in the Background chapter in the sense that it produces many control messages. Here, we can assign a soft worst-case bound of $\Omega(n^2)$, where n is the amount of nodes in the system. There is potential for this to be improved upon; however, we shall do no worse than this worst case.

7.4 General Considerations

The chief reason for the improvement in both the detection delay and message complexity is due to the fact that nodes can begin reporting their individual termination statuses as soon as they experience their individual final work. That is, they begin the process of informing the leader node, or all other nodes in the system, even as the underlying computation continues to execute. Therefore, no time is wasted by waiting for nodes to cease activity; whereas this waiting period is existent in many general approaches to the problem.

Our approach also experiences reduced complexity in terms of distributed concepts required by algorithms that employ a general approach to the problem. Examples of these concepts are distributed global snapshots or distributed clock synching. Since our approach is local in nature, we need not

employ such involved techniques. Admittedly, while defining the final work can sometimes be difficult, the actual algorithm execution is quite simple in that it consists only of active listeners and message sending.

The time performance of our approach is therefore desirable as the efficiency is entirely dependent on the system of nodes and the connections between these nodes. Since this is the case, systems that form the shape of a tree or star shall experience excellent detection times under the Final Work approach. Further, even if the underlying distributed system has a poor topography, we shall still experience excellent metrics under the Final Work approach.

Chapter 8
JAVA Simulation Environment

To help display the efficiency of the approach, we have developed a simulation environment based on the JAVA language. We shall now discuss the details of the program at length. First, let us discuss some of the tough design decisions made during the development of our JAVA simulation environment, tentatively labeled as FinalSim.

8.1 Design Decisions

In creating this FinalSim environment, we faced an early choice of whether the simulations should take place in real-time or be performed on a discrete basis. The benefits of an approach based on discrete events are obvious. First, simulations can be completed nearly instantaneously since there is no concept of "real time" in consideration. This allows for a considerable advantage in terms of time efficiency when compared to an approach which utilizes a real-time clock. Second, a discrete-event approach would allow us to easily track exactly when certain events have occurred. Further, the discrete-event approach allows us to easily model events as real-world phenomena. This would allow us to create a design with less coupling but a higher level of cohesion.

On the other hand, a real-time approach also has some advantages to consider. Consider the possible displays of the two approaches. Considering the discrete-event approach, we shall really only be left with a simple printout of events. This output, at most, would contain a short event description, the time at which the event occurred, and possibly a cause-effect relationship display in order to see how events have affected each other. Further, we cannot see this process occur in real-time(if we are using an approach strictly dependent on a discrete-event paradigm). Now, consider a real time approach. Here, there is more potential in displaying output to the user since the program will be executing in real time. Most importantly, this gives an easy way to design a graphical output, based in real time, as the events take place throughout the simulation. When considering the nature of the work(Distributed systems with computations being executed upon them), this graphical approach based on a real time paradigm would be considered a great advantage. Further, it would allow more power for the user(Which is considered an advantage since we shall be the only ones to use the project!). However, it must be noted that dealing with graphics adds an additional layer of complexity to both the design process as well as overall final product. This is because we must have an additional class or set of classes in order to process data and display graphical output.

In the end, it was decided that a real-time approach would allow easier analysis of the simulations as they proceed. Being able to watch the simulations unfold in real time allow us to confirm more easily that a simulation is working properly. Additionally, it allows us to watch the Final Work approach take place first hand. Although the advantages of an approach based on the analysis of discrete events, such as the increase in time efficiency and the potential for a highly cohesive, loosely coupled design, were important to consider, we felt that the increased power for the user experienced in a real time environment would be more beneficial

The program is wholly written in the JAVA programming language. Essentially, it serves as a scaffolding of sorts in order to run simple distributed events such as message sending and message processing. At first, there were no specific algorithms, such as leader election or consensus algorithms, coded into the program. Instead, we implemented general algorithms which send a given amount of

messages randomly between all nodes in a given system. The idea follows closely to the overall idea of the Final Work approach. That is, we are not necessarily concerned with the specific type of algorithm to be executed. Instead, we assume that we have deduced the required final work properties of the given algorithm. This isolation of thought allows us to run a wide array of different random algorithms and focus on how the Final Work approach performs. Since this initial creation, a generic leader election algorithm and a generic consensus algorithm have been hard-coded into the program in order to test some implementation details of the Final Work approach.

The main metrics taken into consideration are as follows. The time required to detect termination after the algorithm has ended and the ratio of this detection time compared to the overall execution time. These are displayed during the actual simulation and is also displayed upon the simulation's end.

8.2 Functionality

Users are able to add nodes and links through two different mechanisms. First, the user may add a node or link through a simple spreadsheet format. For adding a node, data such as the node name and node position shall be entered. For adding a link, data such as the link name as well as the names of the connecting nodes shall be entered. Note that these operations shall fail if existing names are given. Additionally, the add link command shall not be performed if non-existent node names are given. This is done to preserve system consistency as a link should only exist if it connects exactly two nodes. Alternatively, the user may add nodes/links through a graphical interface. Note here that names cannot be specified and the nodes/links shall be given generic names upon creation. Nodes and links may also be deleted through the graphical interface. If a node is deleted in this way, all attached links shall be deleted as well. Note that upon creation of a link, it shall be assigned a time delay based on the distance between the nodes which it attaches. Individual link delays can be changed through the graphical interface. The graphical interface displays a graphical representation of the current system. However, the user may also view a text based display of node/link data.

Simulations may be run through the simulation mechanism. Given the user's current distributed system, a set of tasks are instantiated upon the execution of a simulation. The user may choose either to execute a consensus algorithm or a leader-election algorithm. Note that it is the user's responsibility to insure there are no partitions between communicating nodes before running a simulation. Also note that in contrast to link delays, node delays are instantiated upon the creation of a simulation event. The tasks, or messages, shall travel from node to node across the system of links until they reach their destination. Noting that the algorithm type is leader-election or consensus, the nodes shall label their final work as complete only when they decide on a leader node or a consensus value. Once this has happened, the node will create a new task to send back to the master node (NODE-0 by default). Upon receiving this message, the master node shall take note of the original creator of the task. Once this master node has received final work completion messages from each node used in the computation, it shall decide that the original computation has terminated. All events are timed appropriately and a graphical display is provided to display the performance in terms of the earlier mentioned metrics. Note here, that in terms of the current implementation, we have used the Elected detection approach as described in chapter on informing processes. That is, we only are interested in informing the master node of the termination status of the computation. The nodes shall rely upon local routing algorithms to inform the master node of the status. In the future, we hope to implement the Flooding version of detection as well since this is the version that allows for a theoretically instantaneous detection time.

8.3 Execution Example

For an easy example, let us consider a simple and small distributed system of only four nodes. By clicking on the view graph button, we are given a blank template with which to create our system(Figure 9a).

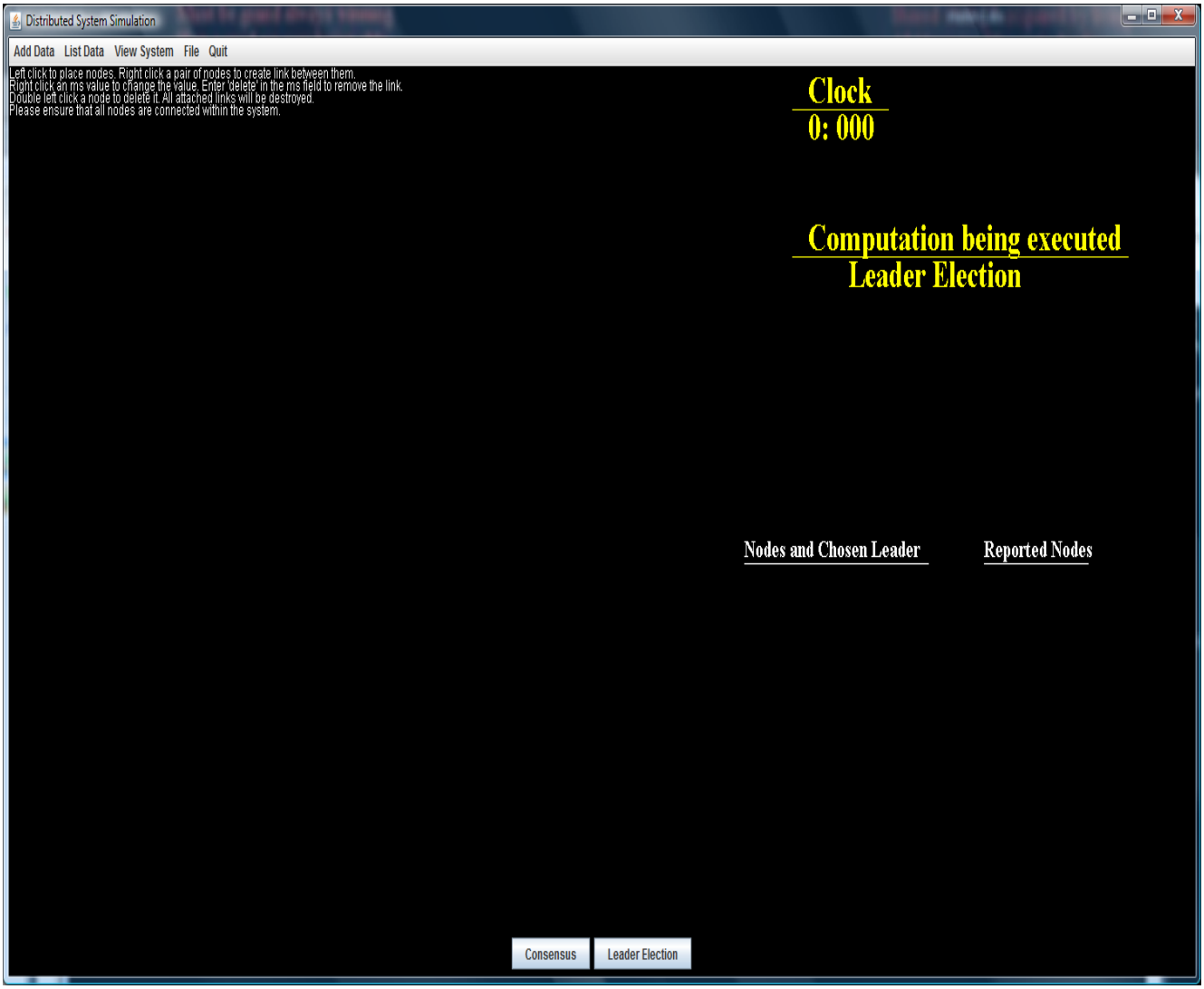


Figure 9a

Now, we proceed to create our small system by creating four nodes. Note here that NODE-0 is the 'leader node', as described previously. Because this node shall only serve as a master, it shall not be given a UID(Figure 9b).

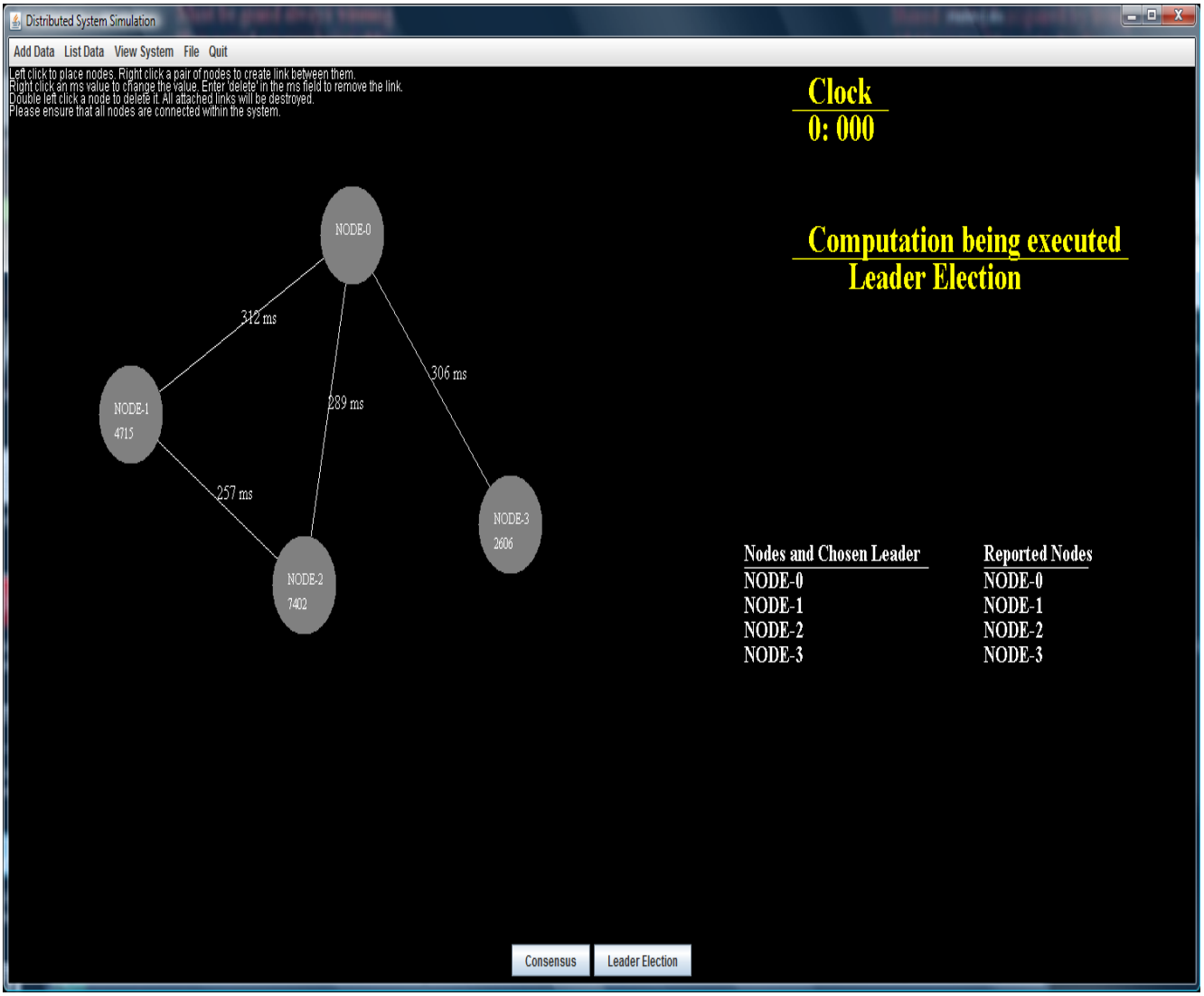


Figure 9b

As you can see, during the creation process, the program has instantiated latency delays on the links we had created. Next, once we have made certain that there are no partitions existent in the system, we may proceed in running the simulation. To do so, we press the 'run simulation' button under the 'view system button'.

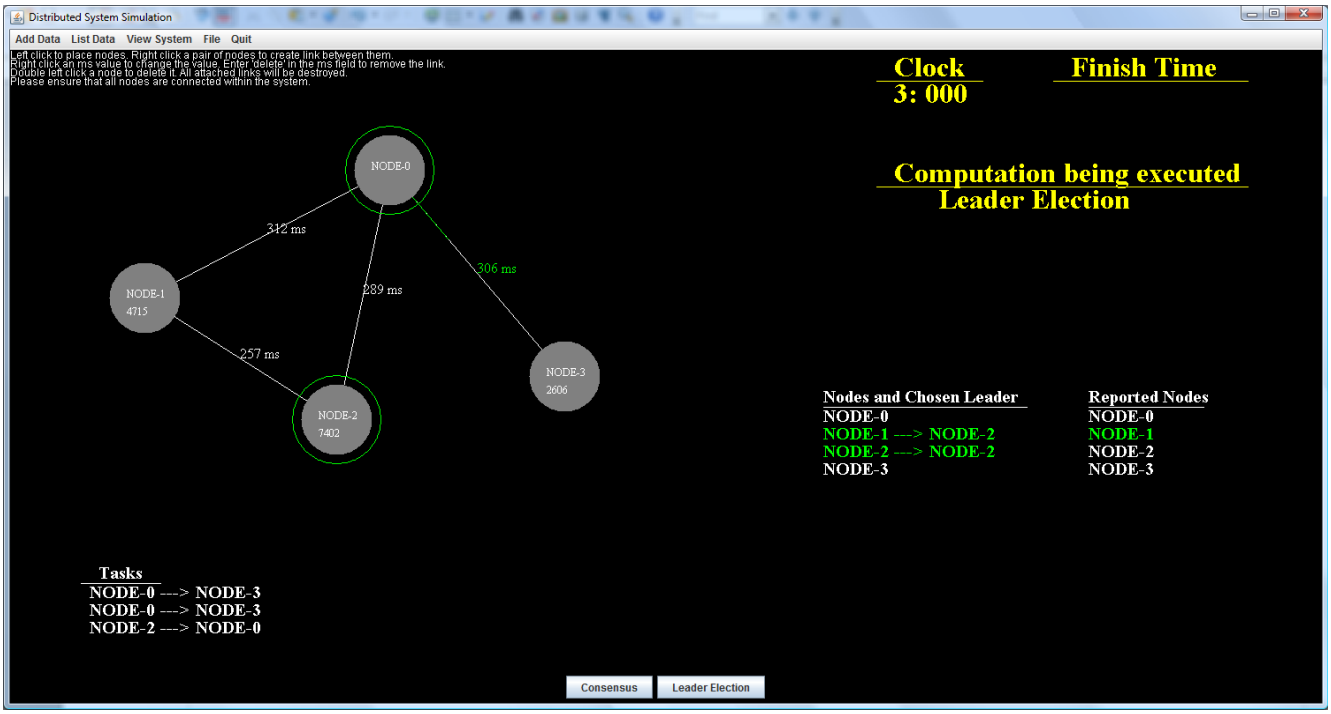


Figure 9c

In figure 9c, we can see the simulation during its execution. We can see that nodes 1 and 2 have chosen a leader; however, only NODE-1 has reported back to the master node at this point. We can also see that nodes Node-0 and Node-2 are currently processing recently received messages while the link between Node-0 and Node-3 is active.

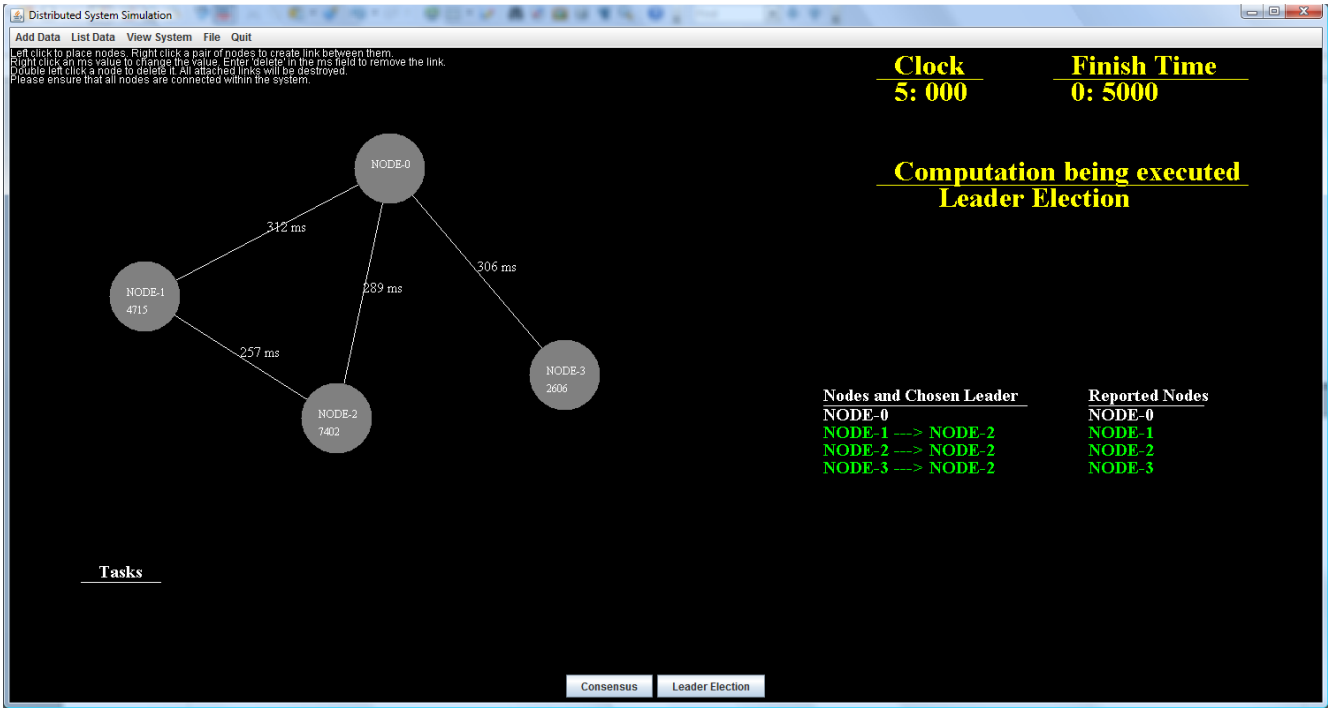


Figure 9d

In figure 9d, we can see the simulation after it has been completed. As we can see, all nodes have correctly chosen NODE-2 as the leader as it has the largest UID. Upon making this selection, each node noted its individual termination status and began the process of informing the master node, NODE-0. From the figure, we can see that all nodes have successfully gotten word back to NODE-0. Once NODE-0 discovers that all nodes have reported, the simulation ends and displays the metrics in the upper right corner. Here, we can see that the entire simulation took five seconds to execute and the delay between actual termination and the detection of said termination only took .5 seconds. This means that the actual computation executed for 4.5 seconds and the detection delay took .5 seconds. This is an efficient delay; however, it could be improved upon if we implement the flooding version of the detection portion of the algorithm, which is one of our goals in the future work section.

When executing simulations, the user must remember to ensure that there are no partitions in the system. Other than this stipulation, any distributed system you can construct with the constructs of the program should be executable without error. For more information regarding the project, please refer to the design document found in the appendix.

Chapter 9
Importing to an Imperfect Environment

All discussion to this point in regards to our Final Work approach has assumed that the algorithm is operating in a perfect synchronous environment. That is, the algorithm executes in perfect lock-step, or round by round, fashion. Further we have assumed that the system shall never experience any type of fault. Note that this assumption has been true for not only for our distributed termination detection algorithm, but also for the underlying computation as well. Obviously, depending on which type of environment the underlying computation is executing shall also be the same type of environment on which our DTD algorithm shall operate.

We have made the assumption about the perfect environment for two main reasons. First, some of the underlying computation examples we had observed in chapter four were intended to be executed upon fault-less synchronous environments. Second, and more importantly, by assuming a perfect synchronous environment, we were able to make the creation process easier in the sense that we did not necessarily have to plan how to handle faults and message ordering issues during this creation process. However, it should be noted that we did have these concerns in the back of our minds during this creation process. As we shall see, due to these early considerations, this importing process shall be somewhat of an easy transition in some cases.

Now that we have developed our Final Work approach to the point it would operate efficiently in a synchronous and fault-less environment, we shall now discuss the difficulties in importing the approach to an asynchronous and faulty environment. First, we shall discuss the various difficulties and possible alterations that would need to be made to the approach in order for it to adapt to an asynchronous communication model.

9.1 Asynchronous Environment

Consider the properties of an asynchronous communication model(Recall the Background chapter or [15] for a more thorough refresher on the topic). In this type of model, we know there are essentially no rules when considering the time delays of communication between nodes in the system. That is, we cannot assume the notion of a 'round'. Further, we cannot make any assumptions regarding the order of sent messages. That is, messages shall not necessarily be read in the order they had been sent. The only real assumption we can make about time delay between sent messages is that this delay shall at least be finite. That is, a message cannot be held in limbo indefinitely. However, when we discuss link failures, we must also consider the possibility of messages being upheld indefinitely.

Now, consider the Final Work approach. The approach basically consists of two broad steps. First, every node used by the underlying computation decides it has locally terminated at some point during the lifespan of the underlying computation. This local termination detection is completely contained in each individual node. Therefore, we can see that the asynchronous nature of the communication model has absolutely no effect on this step.

The second broad step of the Final Work approach is the process of the nodes sharing their local termination status with each other. If you recall, we had discussed a few different approaches to complete this step. Although it does not matter which we choose, let us consider the Flooding

approach. If you recall, this approach worked like a basic flooding algorithm, where every node, upon termination, would send its updated status to every other node used in the computation. Does the asynchronous nature of the underlying communication model affect this step? In short, no! As you recall, we have the global detection of termination once a single node discovers that every node used by the computation has terminated. Regardless of which node makes this discovery, there is absolutely no reliance on the time delays between communicating nodes. Further, there is absolutely no reliance upon the ordering of these messages in terms of the time they are received. Similar arguments can be made for the other aforementioned detection processes since none of them are reliant upon the ordering of messages.

So, given these arguments, we can see that we have designed our approach in such a way that it operates in the same manner whether it is executing in a synchronous or asynchronous environment. This is desirable.

9.2 Node Faults

Node faults, or process failures, can be described as a node that suddenly dies. Here, for a node to die is equivalent to a node ceasing any and all activity. It shall perform no internal computing. Further, it shall not send nor receive any messages. If a message is sent to a dead node, it shall simply be ignored by the dead node. Once again, refer to [15] for a more thorough discussion on the topic.

How does this affect our Final Work approach? Once again, let us consider the two broad steps of the approach. What happens if a node dies before the DTD algorithm has a chance to decide the node's final work has been completed? This, in turn, would mean that the underlying computation also has not been able to observe its final work. That is, the node shall still be used in the computation(Assuming our definition of final work was correct for the underlying computation). Since the termination status as interpreted by the DTD algorithm and the actual final work status of the node itself are both false, we can see that we maintain consistency in this scenario. In terms of handling the fault, our Final Work approach would simply adopt the fault-tolerance protocol of the underlying computation. That is, if the dead node is simply dropped from the underlying computation, then it shall also be dropped from consideration by our DTD algorithm. On the other hand, if the underlying computation is blocking, then it is inclined to wait for the dead node to be rebooted and rejoin the underlying computation. Once again, our DTD algorithm would simply do the same. Regardless of which approach is taken, we can see that our Final Work algorithm shall handle the matter just as gracefully as the underlying computation shall.

Now, fault-tolerance is not always as simple as the two cases presented in the previous paragraph. Certainly, some protocols must call for a series of acknowledgment messages and responses to these message. Obviously, different computations shall employ a broad range of approaches to handle fault-tolerance. The point here is that the Final Work approach maintains a very simple strategy of communication which is executed on top of the underlying computation. Therefore, regardless of what type of fault-tolerance protocol exists for the underlying computation, we are confident that these protocols can simply be reproduced in most cases by our approach. The end result of this is desirable: The Final Work approach is fully contained within the fault-tolerance protocol of the underlying computation. That is, as long as this protocol can be reproduced, we can guarantee that a node failure shall not impact our metrics any more than it would effect the complexities of the underlying computation.

9.3 Link Faults

A link failure is simply a failure where a message is dropped or lost upon being sent on a link. Like node faults, link faults shall affect our DTD algorithms in a similar fashion. How do link failures affect our DTD algorithm? Let us consider the Flooding approach for the detection portion of our algorithm. Here, we have many messages being sent by each node. The goal with the Flooding approach is to inform at least one node of the local termination detection of every node in the system. Obviously, if we only experience a few dropped messages here and there, we shall still easily have a situation where at least one of the nodes in the system receive all of the necessary messages. The only scenario with which we would have to worry is the case where we have a small choke point between nodes as in figure 10a. In this situation, if the message from NODE-4 to NODE-5 is dropped, then we have a situation where no node in the system, except for NODE-4, shall know of NODE-4's termination status. To remedy this situation, we would likely use some sort of message acknowledgment system to ensure messages are correctly sent. However, in this case, we would unfortunately have to add additional messages.

Figure 10a

In the future, we would like to expand upon the protocols to handle faulty environments as we feel that the Final Work approach has potential to handle faults in an elegant manner.

Chapter 10
Conclusion

In this project report, we have done a great deal of work in attempting to find a new approach from which to solve the distributed termination detection problem. Given the general definition of what it means for a distributed computation to be in a terminated state, we have constructed a new, but still equivalent, definition of termination within a distributed system. We arrived at this new definition by first observing the termination properties of a handful of different distributed computations. By isolating these properties, we were able to come up with a different scope from which to define termination. This would then allow us to attack the problem with a different approach than most other algorithms. This, in turn, would lead to a unique solution to the DTD problem.

With this new definition of termination in mind, we then created a sequence of algorithms to solve the DTD problem. Although the first two algorithms in the sequence are somewhat naive, they serve as the base solutions from which the third and fourth algorithms borrow.

Algorithm three, which has to do with using statistical information such as mean process times, mean message delays, as well as average number of faults, was only briefly touched upon during this project report. We came to the conclusion that although utilizing such statistical information could help in the development of a DTD solution, it is not enough on its own on which to base an entire algorithm. Instead, it is our opinion that such statistical information should simply be used in an auxiliary way in order to help improve the efficiency of other popular approaches.

The remainder of the report was used to discuss the fourth, and final, algorithm of the sequence: The Final Work approach. This approach, which utilizes the termination properties of specific distributed computations, is then discussed in terms of feasibility. After this discussion, the Final Work is expanded upon by describing exactly how specific computations would have their termination status detected by the Final Work approach.

We then described some of the more detailed aspects of the algorithm, such as how overall detection shall proceed and how the approach can be used to solve the termination of a single computation as well as how it can be used to detect the overall termination status of the entire distributed system.

We also created and discussed our JAVA simulation environment: FinalSim. Through the use of FinalSim, we are able to accurately display the Final Work approach in action. We present an example of these scenarios in action in the report.

Finally, we also presented a discussion of the feasibility of using the approach in a faulty, asynchronous environment. As can be read in the discussion, the approach makes no assumption about rounds or a lock-step communication model, and therefore is not affected by an asynchronous environment. Faults, on the other hand, can be handled by an injection of fault-tolerance. However, this adds to the overall message complexity of the approach.

In terms of performance, our approach does well when compared to popular DTD algorithms of today. It boasts a worst-case complexity to popular algorithms of today, which also can be reduced significantly based on the underlying network topology. In some instances, the detection of global

termination by a single node can even be done in constant time(Instantaneously from a theoretical point of view). In terms of message complexity, depending on which type of detection is used, the worst-case scenario is bounded by $O(n^2)$, where n represents the number of nodes used by the underlying computation. However, when assuming that the underlying distributed system has a more favorable shape, we achieve message complexities ranging from O(n) to O(D*n) to $O(n^2)$. Detection delay, depending on the type of detection process used, ranges from O(c) to O(D).

The greatest point of concern as it pertains to the Final Work approach is the fact that it cannot be truly general on its own accord. This is because some distributed computations simply do not display termination properties detailed enough such that they are definable under the Final Work paradigm. Therefore, in the case for these undefinable computations, we would simply substitute some other popular DTD algorithm for the Final Work approach, and therefore still experience equivalent complexities for those computations. Further, under certain circumstances, the Final Work approach boasts an instantaneous detection of termination. This leads us to believe that the approach has merit in its potential and it is warranted further consideration.

Chapter 11
Future Work

Although we have done a good deal of work on this project, we have left open some avenues when it comes to future work on the project. That being said, we shall now discuss a few potential possibilities worthy of consideration if one wishes to pursue.

To this point, we have produced a great deal of discussion in terms of our approach. We have mainly discussed it from a theoretical standpoint. In terms of implementation, we have executed and tested the approach using our JAVA simulation environment. Although this has given us great control in testing our approach, the approach must be implemented and tested in an actual distributed environment. Further, the approach should be tested using actual distributed computations. This way, the approach can be tested in an actual environment. Obviously, this shall require more resources than did our simple simulation environment. For instance, one would need access to a distributed system with the authority to modify the computers on the system. Basically, this shall require administrative privileges on the system in question. It would be wise to first locate a somewhat small distributed system, something like 10 to 20 nodes. For the purposes of the problem, this system should be strongly connected as this is to be expected by most distributed systems in general. Once the Final Work approach has been implemented on each individual node in the system, we should select a few actual distributed computations to execute on the system. As was done previously in the project, these specific computations should first be selected for their simplicity. Pending the success of the approach with these simple computations, more complex computations can be executed and therefore tested, which leads to our next opportunity for future work on the project.

Recall that within this project report we had only described a few simple computations and how they are defined under the Final Work approach. Obviously, must attempt to categorize all further potential distributed computations under the Final Work paradigm. As discussed, true generality is not possible solely with the Final Work approach since there exist some computations which cannot be accurately defined under the paradigm. However, in these scenarios, we could simply substitute the Final Work approach with some other popular DTD algorithm to maintain good efficiencies in these scenarios. Back to the point, however, we would still like to categorize as many distributed computations as possible under the Final Work paradigm. Once a given computation has been categorized and added to the approach's implementation, it can be tested on the environment discussed in the previous paragraph. To some degree, this categorization would be an ongoing process as different computations are constantly being created.

In preparation for these aforementioned projects, it would be wise to further expand upon our FinalSim environment. To accomplish this, we would simply implement more complex computations in the FinalSim environment. Given these added computations, we shall then implement their Final Work definitions into the FinalSim environment. As you can imagine, our FinalSim environment would serve as an adequate scaffolding on which we can test our Final Work definitions. This shall help to save resources as we shall be able to perfect our definitions in a cheap simulation environment before implementing them on an actual distributed system, which would be more costly.

Although we presented a brief discussion on the potential pitfalls of importing to an imperfect distributed environment, we would like to expand upon this discussion in the future. Specifically, we would focus on developing an elegant approach to handle potential link failures. Further, we would also

continue to investigate the possibility of adapting the underlying computation's fault tolerance protocols for use with our Final Work approach.

Finally, although we only touched on the subject briefly, we believe the concept of using statistical information about a given system could be further explored. Specifically, the feasibility of using such an approach combined with popular DTD algorithms should be explored as we believe there is a potential to increase the efficiencies of these algorithms if they are augmented with this statistical information.

APPENDIX

User Requirements

A user must be able to build a distributed system. She can do this through the use of a text-based spreadsheet mechanism or a graphical mechanism. Tasks included in building a system are creating nodes graphically, creating links graphically, creating nodes by spreadsheet, creating links by spreadsheet, deleting nodes, deleting links, and modifying delays between links. Users shall also be able to view the system in a textual representation as well as a graphical representation. The task to accomplish this are listing nodes, listing links, and viewing the graphical representation of the system. Finally, the user shall also be able to execute simulations. This functionality encompasses the most important functionality of the program. We shall now present use-case diagrams to display this functionality in greater detail.

Create a Node Graphically.

From the view graph environment, the user right-clicks within the graphical representation of the system. If the node to be created is a safe distance away from other nodes, the system creates the node and gives it a generic name. If the node is graphically too close to a node(s), nothing happens and control is returned to the user.

	Action Performed by Actor		Response from System
1	The user right-clicks within the graphical environment of the system.		
		2	If the node to be created is too close to another node, do nothing and return control to the user. If the node to be created is in an acceptable position, create the node and assign it a generic name. Return control to the user.

Create a Link Graphically.

From the view graph environment, the user left-clicks within the graphical representation of the system. If the click is within the circumference of a node, the system returns control to the user so he may connect the link to another node. The user left-clicks once more, and if the click is within the circumference of a different node, create the link and assign it a generic name. If, at any stage, the left-click is not within the circumference of a node, return control to the user.

	Action Performed by Actor		Response from System
1	The user left-clicks within the graphical environment of the system.		
		2	If the click is within the circumference of an existent node, the system highlights the node, save its name in memory, and returns control to the user. If the click was not within the circumference of a given node, return control to the user.
3	The user again left-clicks within the graphical environment of the system.		
		4	If the click is within the circumference of an existent node and is not the node clicked on previously in step 2, the system creates a link, assigns the link a generic name, and assigns the link information to the node objects which it connects. If the click is within the circumference of the original node found in step 2, this deletes the node(below).

Create a Node by Spreadsheet.

The user clicks the 'add node' button from the 'add data' button. The system then generates a spreadsheet asking for a node name, x coordinate, and y coordinate. The user enters this information and clicks 'add node' or 'cancel operation'. If 'cancel operation' is pressed, no new data is created and control returns to the user. If 'add node' is pressed, and the parameters pass a series of tests, the system creates the node and adds it to system data. If these parameters are not passed, an appropriate error message is given and control is returned to the user.

	Action Performed by Actor		Response from System
1	The user clicks the 'add node' button from the 'add data' button.		
		2	The system displays a spreadsheet asking for a node name, x coordinate, and y coordinate and returns control to the user.
3	The user fills in the spreadsheet and presses either 'add node' button or 'cancel operation'		
		4	If 'cancel operation' was pressed, the system disregards any entered information and returns control to the user. If 'add button' was pressed, the node name given is not already in use, and the coordinates are such that the node to be created is not too close to any other node in the system, then the system creates the node, assigns it a generic name, and assigns it the coordinates that were given by the user. Return is then controlled to the user.

Create a Link by Spreadsheet.

The user clicks the 'add link' button from the 'add data' button. The system then generates a spreadsheet asking for a link name, and two node names. The user enters this information and clicks 'add link' or 'cancel operation'. If 'cancel operation' is pressed, no new data is created and control returns to the user. If 'add link' is pressed, and the parameters pass a series of tests, the system creates the link and adds it to system data. If these parameters are not passed, an appropriate error message is given and control is returned to the user.

	Action Performed by Actor		Response from System
1	The user clicks the 'add link' button from the 'add data' button.		
		2	The system displays a spreadsheet asking for a link name, and two node names and returns control to the user.
3	The user fills in the spreadsheet and presses either 'add link' button or 'cancel operation'		
		4	If 'cancel operation' was pressed, the system disregards any entered information and returns control to the user. If 'add link' button was pressed, the link name given is not already in use, and the node names given both exist within the data in the system, then the system creates the link connecting the nodes that were given, assigns it a generic name, and assigns the given nodes the link object that was just created. Return is then controlled to the user.

Delete a Node Graphically.

From the view graph environment, the user left-clicks within the graphical representation of the system. If the click is within the circumference of a node, the system returns control to the user. The user left-clicks once more, and if the click is within the circumference of the same node, delete the node and remove the node from all system data. If there were any links connected to the node, the system deletes all of these links as well. If, at any stage, the left-click is not within the circumference of a node, return control to the user.

	Action Performed by Actor		Response from System
1	The user left-clicks within the graphical environment of the system.		
		2	If the click is within the circumference of an existent node, the system highlights the node, save its name in memory, and returns control to the user. If the click was not within the circumference of a given node, return control to the user.
3	The user again left-clicks within the graphical environment of the system.		
		4	If the click is within the circumference of an existent node and is equivalent to the node clicked on previously in step 2, the system deletes the node, and removes the node's information, and all links connected to the recently deleted node, from all system data. If the click is within the circumference of a different node than the one found in step 2, this creates a link(above).

Delete a Link Graphically.

From the view graph environment, the user left-clicks within the graphical representation of the system. If the click is within the graphical representation of a link, the system displays a prompt to change the delay ms and returns control to the user. The user enters 'delete' in the prompt. The system then deletes the link and removes its from all system data.

	Action Performed by Actor		Response from System
1	The user left-clicks within the graphical environment of the system.		
		2	If the click is within the graphical representation of a link, the system displays a prompt to change the delay ms and returns control to the user. If the click was not within the graphical representation of a given link, return control to the user.
3	The user enters 'delete' into the prompt and clicks 'change ms'.		
		4	The system deletes the link and removes it from all system data.

Change the MS Graphically.

From the view graph environment, the user left-clicks within the graphical representation of the system. If the click is within the graphical representation of a link, the system displays a prompt to change the delay ms and returns control to the user. The user enters a value in the prompt. The system then assigns the link this new ms and removes its from all system data.

	Action Performed by Actor		Response from System
1	The user left-clicks within the graphical environment of the system.		
		2	If the click is within the graphical representation of a link, the system displays a prompt to change the delay ms and returns control to the user. If the click was not within the graphical representation of a given link, return control to the user.
3	The user enters an integer value into the prompt and clicks 'change ms'.		
		4	The system assigns the link the new MS and alters it in all system data.

List Nodes.

The user presses the 'list nodes' button from the 'list data' button. The system displays all nodes, and all connected links, currently in memory. The system then returns control to the user.

	Action Performed by Actor		Response from System
1	The user presses the 'list nodes' button from the 'list data' button.		
		2	The system displays all nodes, and all connected links, currently in memory. The system then returns control to the user.

List Links.

The user presses the 'list links' button from the 'list data' button. The system displays all links, the nodes which it connects, and the delays assigned to each link currently in memory. The system then returns control to the user.

	Action Performed by Actor		Response from System
1	The user presses the 'list links' button from the 'list data' button.		
		2	The system displays all links, the nodes which it connects, and the delays assigned to each link currently in memory. The system then returns control to the user.

View Graph

The user presses the 'view graph' button from the 'view system' button. The system loads all system data in memory and displays a graphical representation of a system.

	Action Performed by Actor		Response from System
1	The user presses the 'view graph' button from the 'view system' button.		
		2	The system loads all system data in memory and displays a graphical representation of a system. The system then returns control to the user.

Run Simulation

The user presses the 'run simulation' button from the 'view system' button. The system loads all system data into memory and instantiates the simulation with random nodal delays. It then begins executing a simulation.

	Action Performed by Actor		Response from System
1	The user presses the 'run simulation' button from the 'view system' button.		
		2	The system loads all system data into memory and instantiates the simulation with random nodal delays. It then begins executing a simulation. The system then returns control to the user upon completion of the simulation.

Save Data

The user presses the 'save data' button from the 'file' button. The system prompts the user for a file name. The user enters a file name and presses the save button.

	Action Performed by Actor		Response from System
1	The user presses the 'save data' button from the 'file' button.		
		2	The system prompts the user for a file name.
3	The user enters a file name and presses either the 'save' button or 'cancel operation' button.		
		4	If the user pressed the 'save' button, the system saves all system data into the specified file and control is returned to the user. If the file already exists, it is overwritten. If the user pressed the 'cancel operation' button, nothing is saved and control is returned to the user.

Load Data

The user presses the 'load data' button from the 'file' button. The system prompts the user for a file name. The user enters a file name and presses the load button.

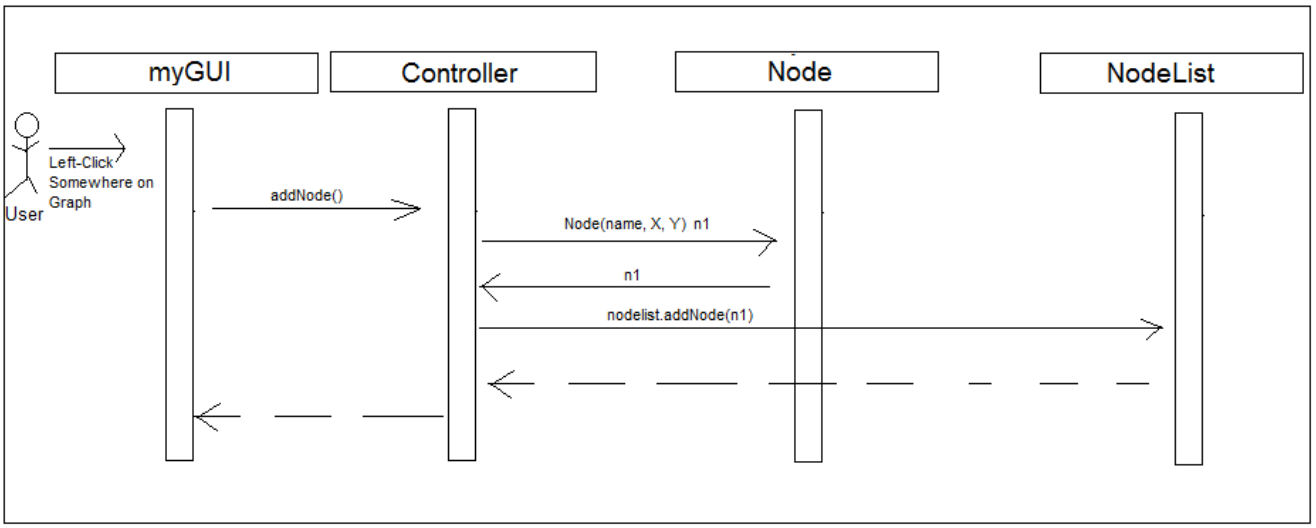
	Action Performed by Actor		Response from System
1	The user presses the 'load data' button from the 'file' button.		
		2	The system prompts the user for a file name.
3	The user enters a file name and presses either the 'load' button or 'cancel operation' button.		
		4	If the user pressed the 'load' button and the file exists, the system loads all system data into memory and control is returned to the user. If the file does not exist, an error is returned and control is returned to the user. If the user pressed the 'cancel operation' button, nothing is loaded and control is returned to the user.

Sequence Diagrams

Next, we shall present some sequence diagrams for the most important pieces of functionality. These sequence diagrams mirror the use-case diagrams from the previous section. In these diagrams, we refer to actual class names. Although the class names are self-explanatory, you may refer to the following section in which classes are fully explained.

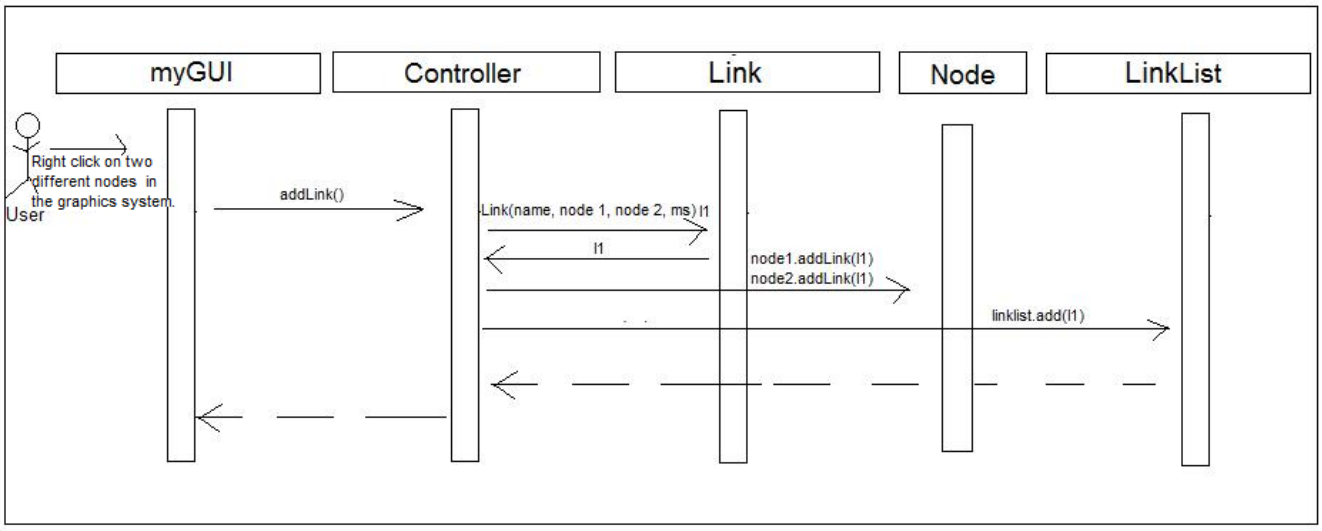
Create a Node Graphically.

Creates a node within the graphical environment of the program. The classes invoked are myGUI, Controller, NodeList, and Node. The myGUI class handles all of the non-business logic involved in the transaction. The Controller class accepts the input from the myGUI class and uses it, along with the Node and NodeList classes, to create the node.



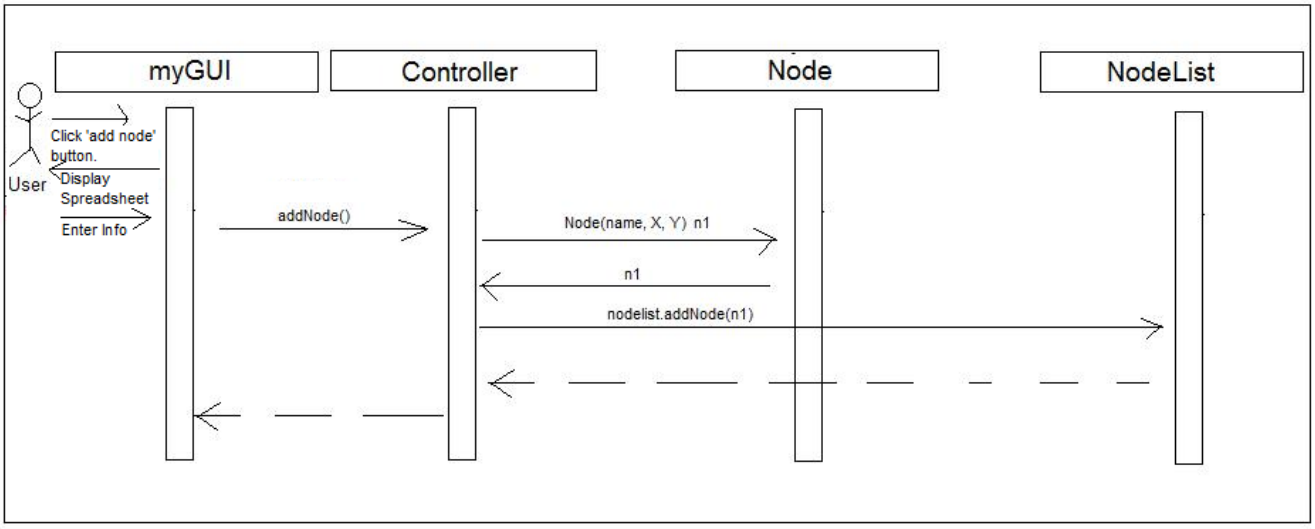
Create a Link Graphically.

Creates a link within the graphical environment of the program. The classes invoked are myGUI, Controller, Node, LinkList, and Link. The myGUI class handles all of the non-business logic involved in the transaction. The Controller class accepts the input from the myGUI class and uses it, along with the remaining classes, to create the link.



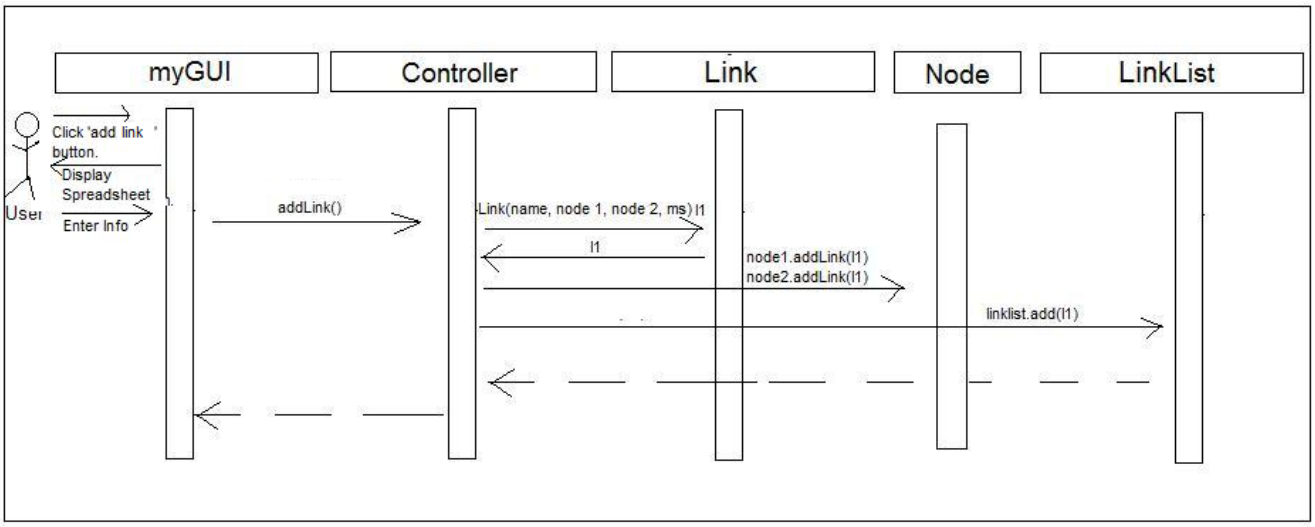
Create a Node by Spreadsheet.

Adds a node through the use of a spreadsheet mechanism. The classes invoked are myGUI, Controller, NodeList, and Node. The myGUI class handles all of the non-business logic involved in the transaction. The Controller class accepts the input from the myGUI class and uses it, along with the Node and NodeList classes, to create the node.



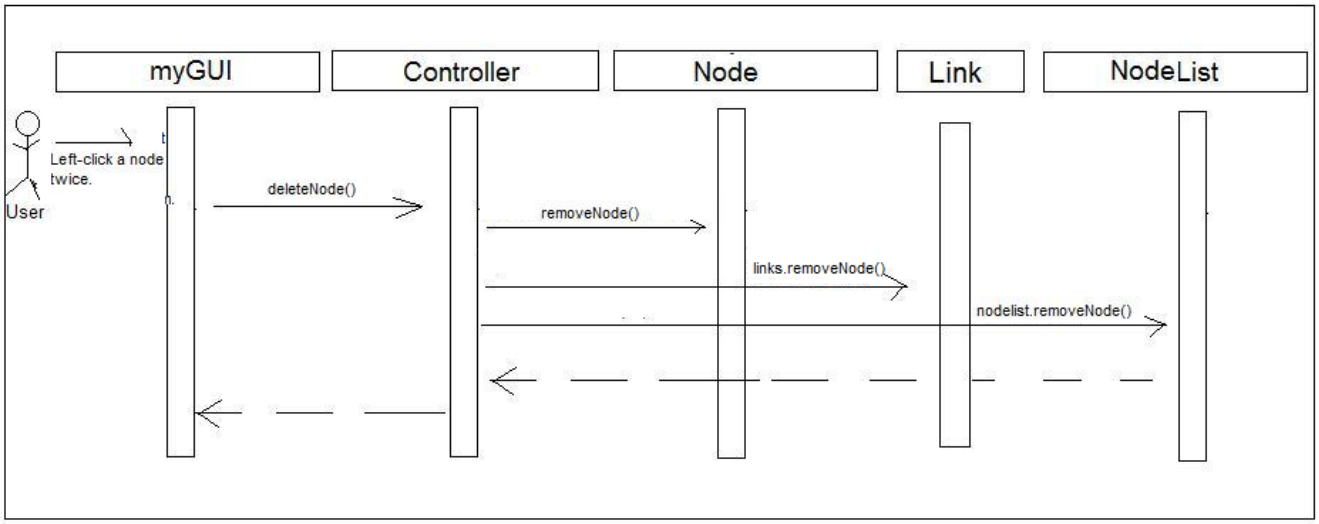
Create a Link by Spreadsheet.

Adds a link through the use of a spreadsheet mechanism. The classes invoked are myGUI, Controller, Link, LinkList, and Node. The myGUI class handles all of the non-business logic involved in the transaction. The Controller class accepts the input from the myGUI class and uses it, along with the remaining classes, to create the link.



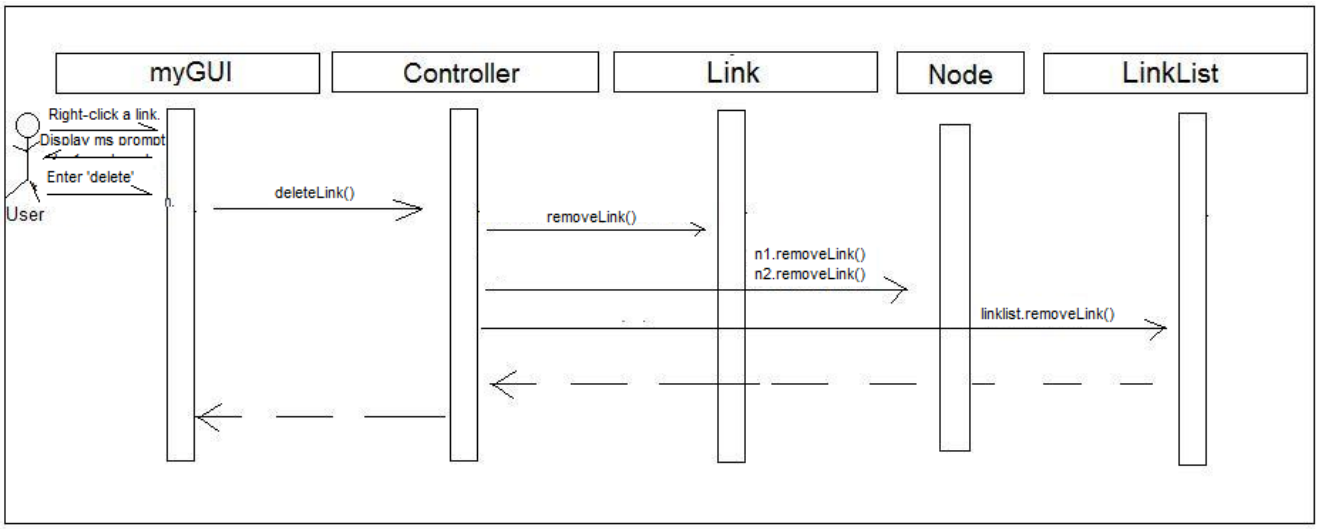
Delete a Node Graphically.

Deletes a node through the graphics mechanism. The classes invoked are myGUI, Controller, NodeList, and Node. The myGUI class handles all of the non-business logic involved in the transaction. The Controller class accepts click locations from the myGUI class and uses it, along with the Node and NodeList classes, to delete the node.



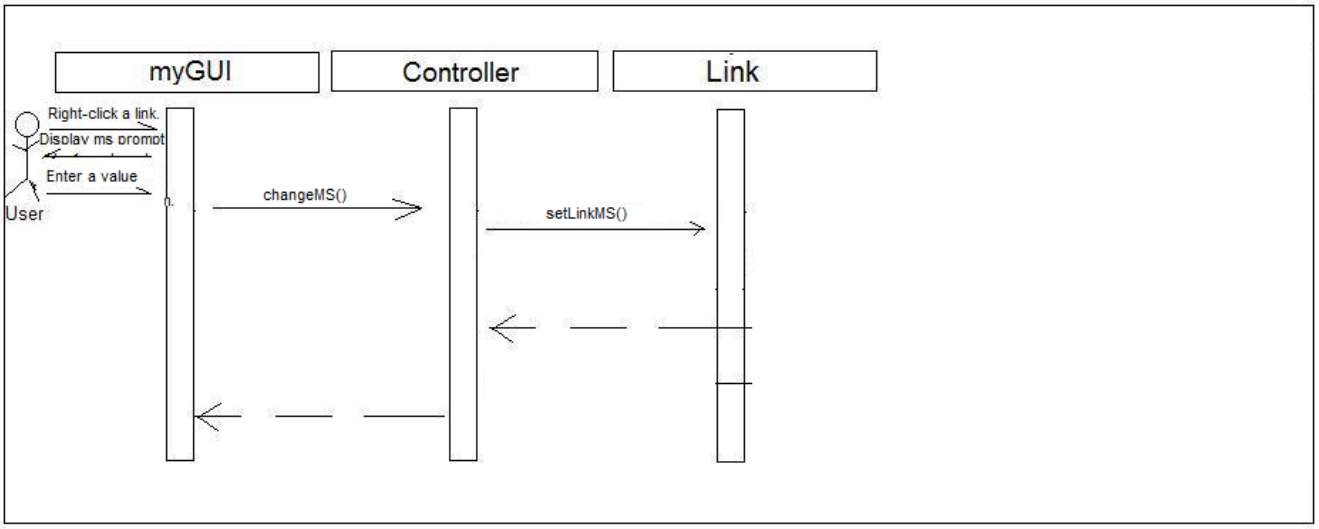
Delete a Link Graphically.

Deletes a link through the graphics mechanism. The classes invoked are myGUI, Controller, LinkList, and Link. The myGUI class handles all of the non-business logic involved in the transaction. The Controller class accepts click locations from the myGUI class and uses it, along with the Link and LinkList classes, to delete the link.



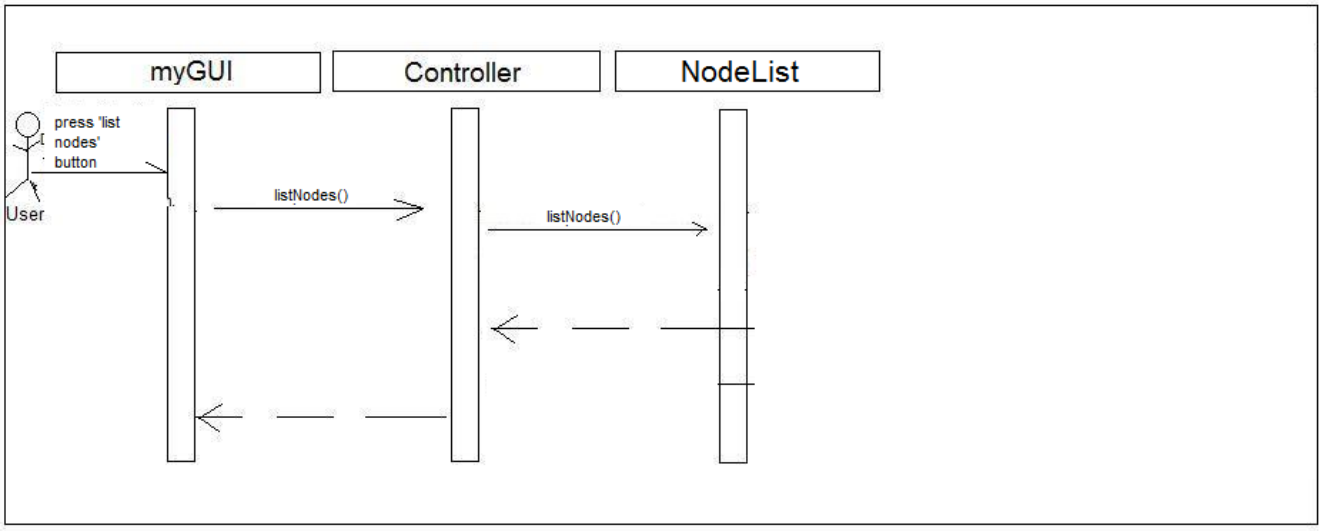
Change the MS Graphically.

Changes the ms delay for a given link. The classes invoked are myGUI, Controller, and Link. The myGUI class handles all of the non-business logic involved in the transaction. The Controller class accepts click locations from the myGUI class and uses it, along with the Link class, to alter the link's ms.



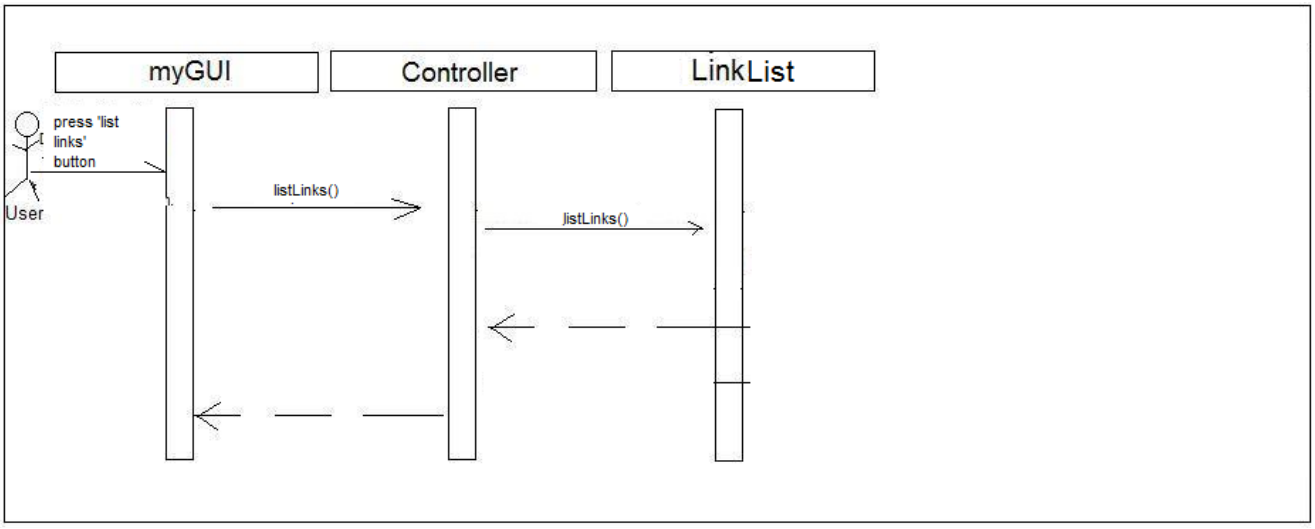
List Nodes.

Lists the nodes in a textual format. The classes invoked are myGUI, Controller, and NodeList. The myGUI class handles all of the non-business logic involved in the transaction.



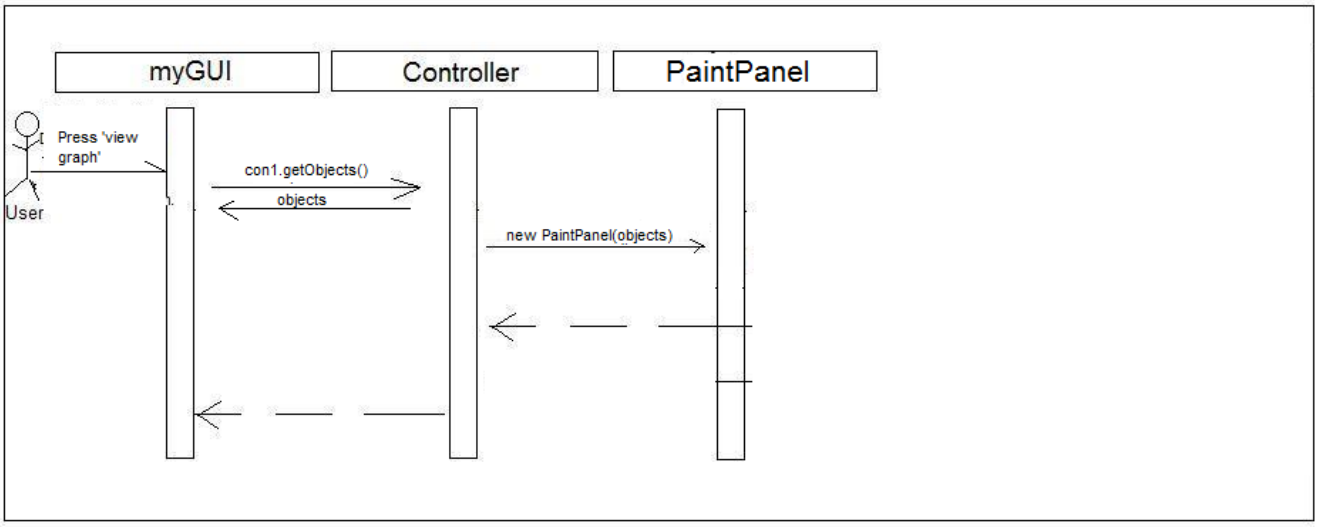
List Links.

Lists the links in a textual format. The classes invoked are myGUI, Controller, and LinkList. The myGUI class handles all of the non-business logic involved in the transaction.



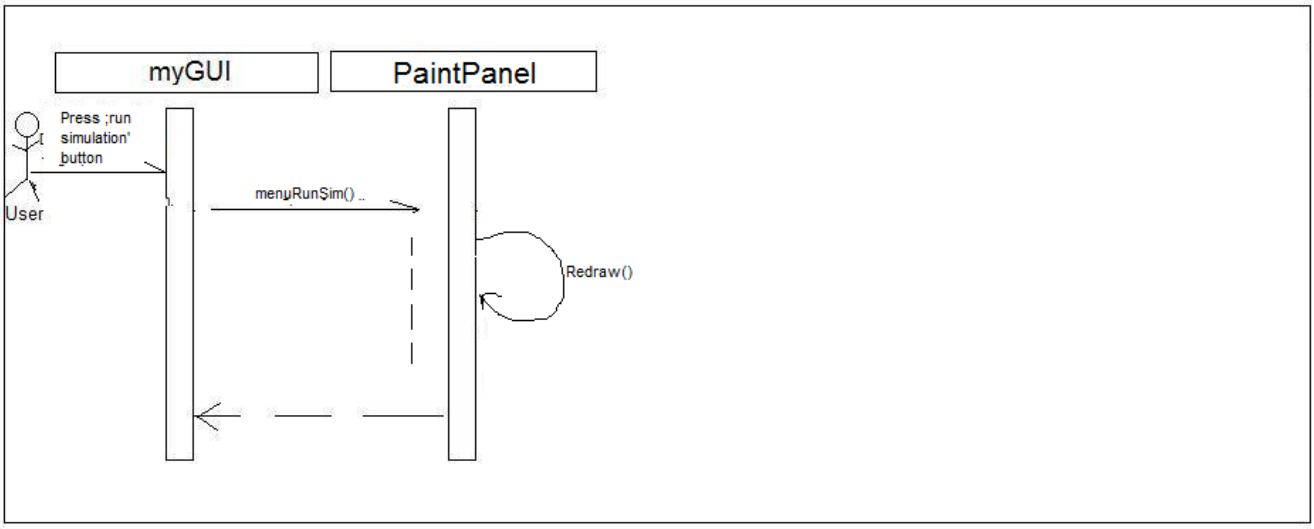
View Graph.

Loads all system information and displays graphics. The classes invoked are myGUI, Controller, and PaintPanel. The myGUI class handles all of the non-business logic involved in the transaction.



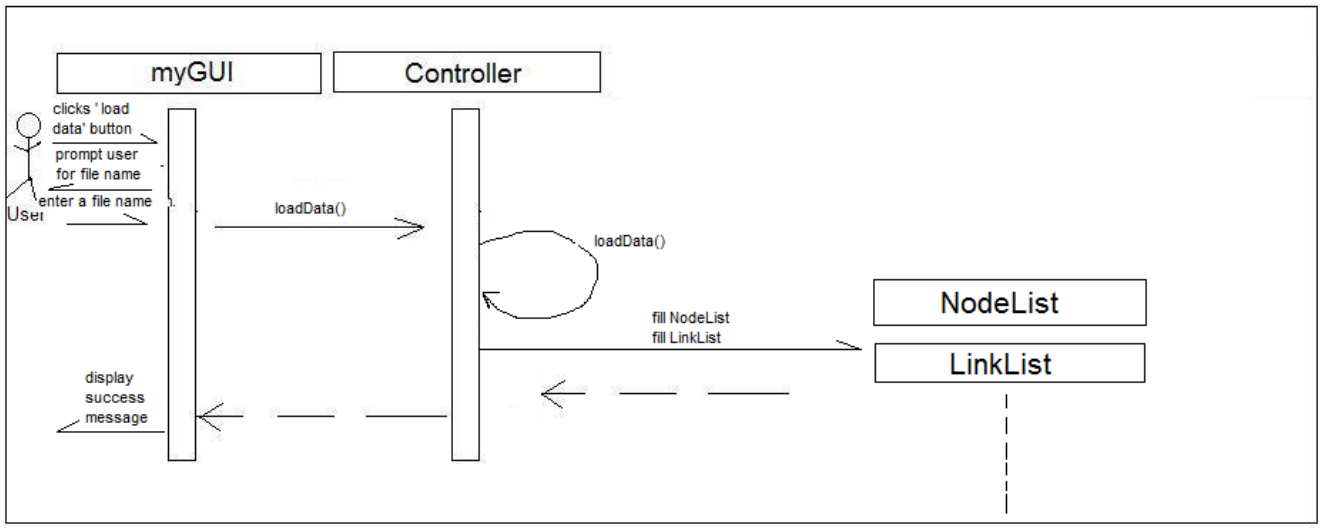
Run Simulation.

Executes a simulation. The classes invoked are myGUI and PaintPanel. The myGUI class handles all of the non-business logic involved in the transaction. PaintPanel is responsible for continually redrawing throughout the simulation's lifespan.



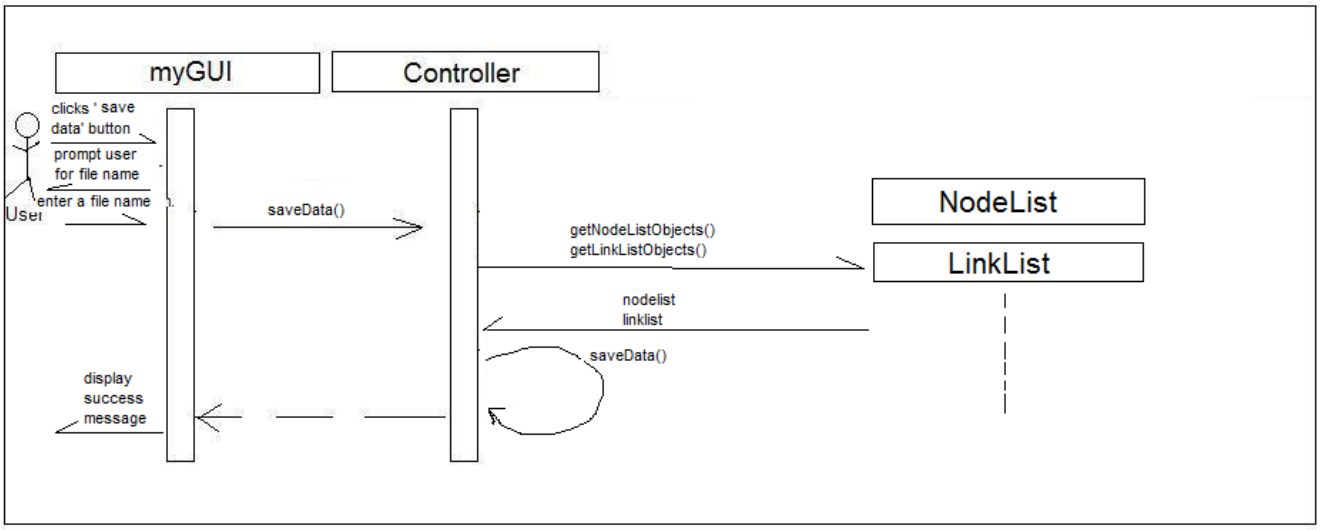
Load Data

Loads data from a file. The myGUI class handles all of the non-business logic involved in the transaction. Controller calls upon NodeList and LinkList to load the data.



Save Data

Saves data to a file. The myGUI class handles all of the non-business logic involved in the transaction. Controller calls upon NodeList and LinkList to save the data.

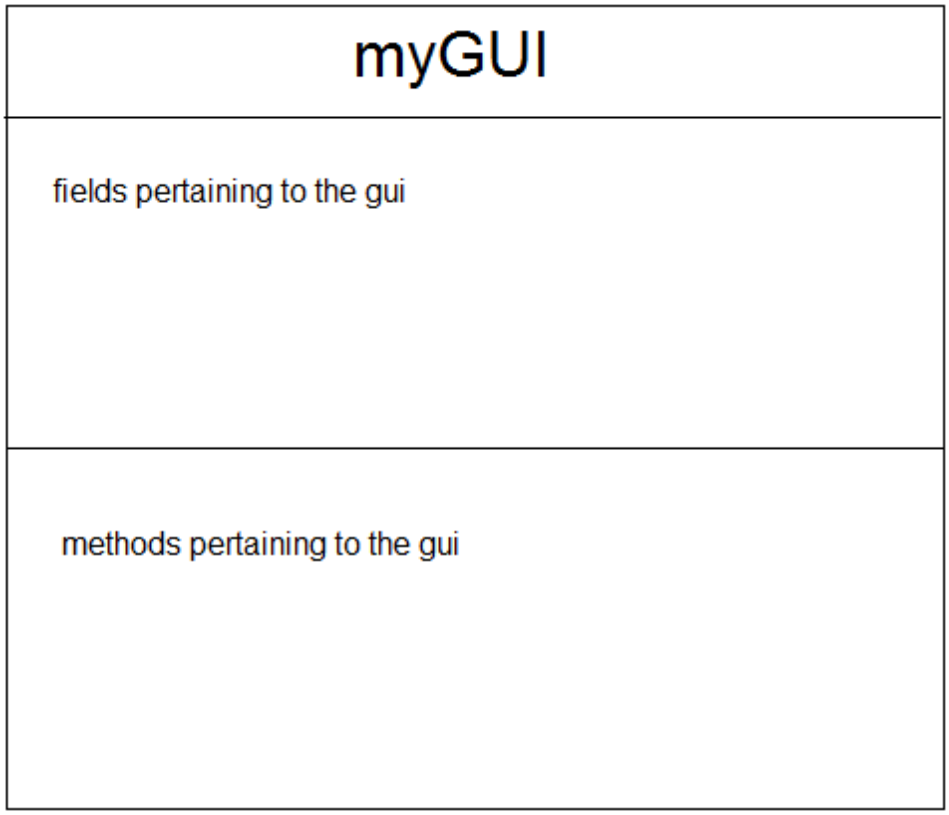


Class Diagrams

Next, the class diagrams shall be presented. They shall be presented in the order they are called upon during an average test case.

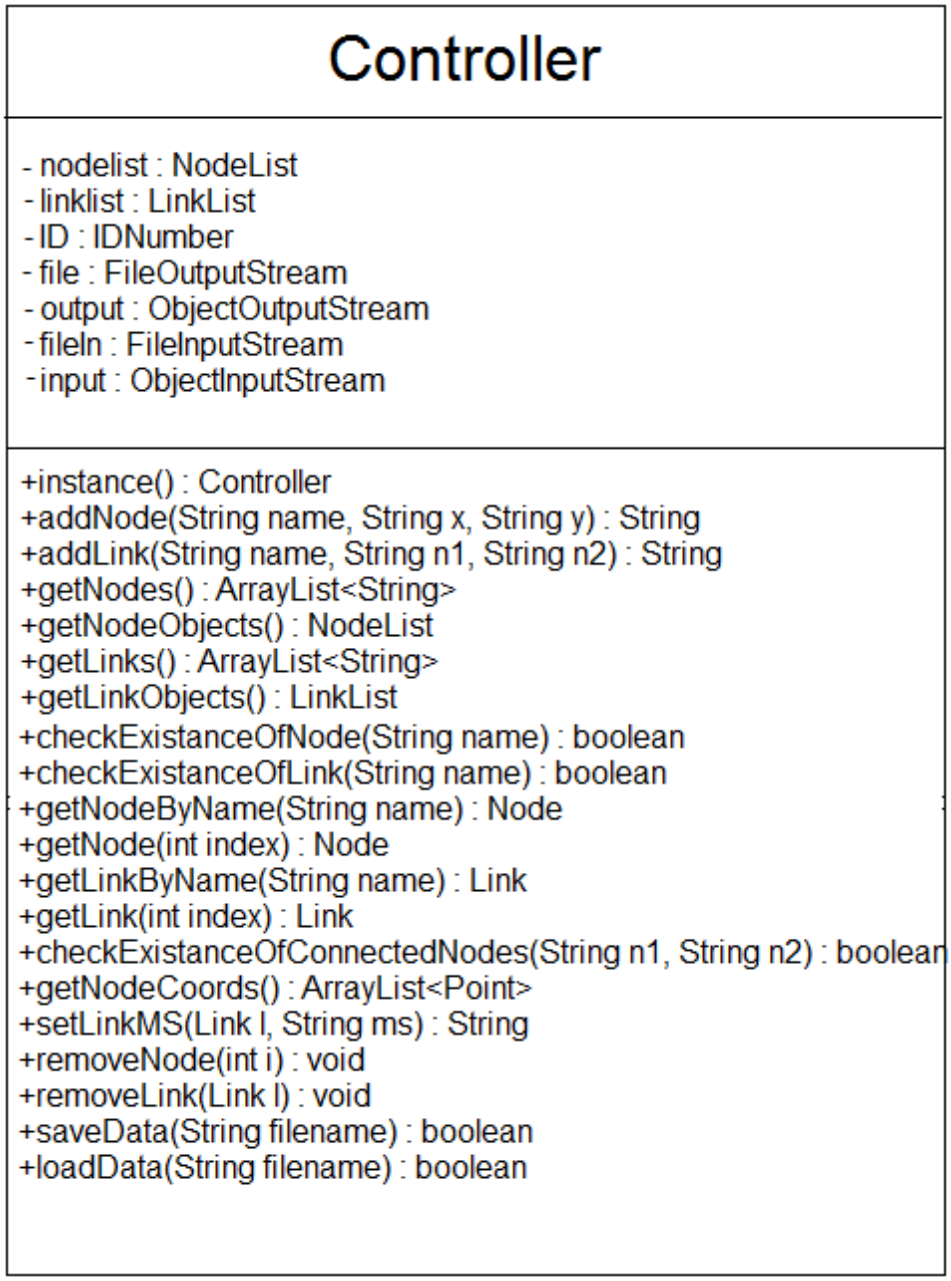
myGUI Class

The myGUI class serves as the GUI for the user to communicate with the system. This class serves as the springboard to execute all of the previously mentioned functionality. That is, when the user interacts with the system in any way, the myGUI class is responsible for handling these interactions. In order to execute the business logic of these transactions, the myGUI class passes the information to the Controller and PaintPanel classes. Here, we have not specifically listed the actual fields and methods as they are not particularly interesting.



Controller Class

The Controller class initiates and handles most of the business logic within the entire program. That is, it **controls** how the program is executed after being passed information from the myGUI class. Listed below are the most important fields and methods it utilizes to perform its tasks.

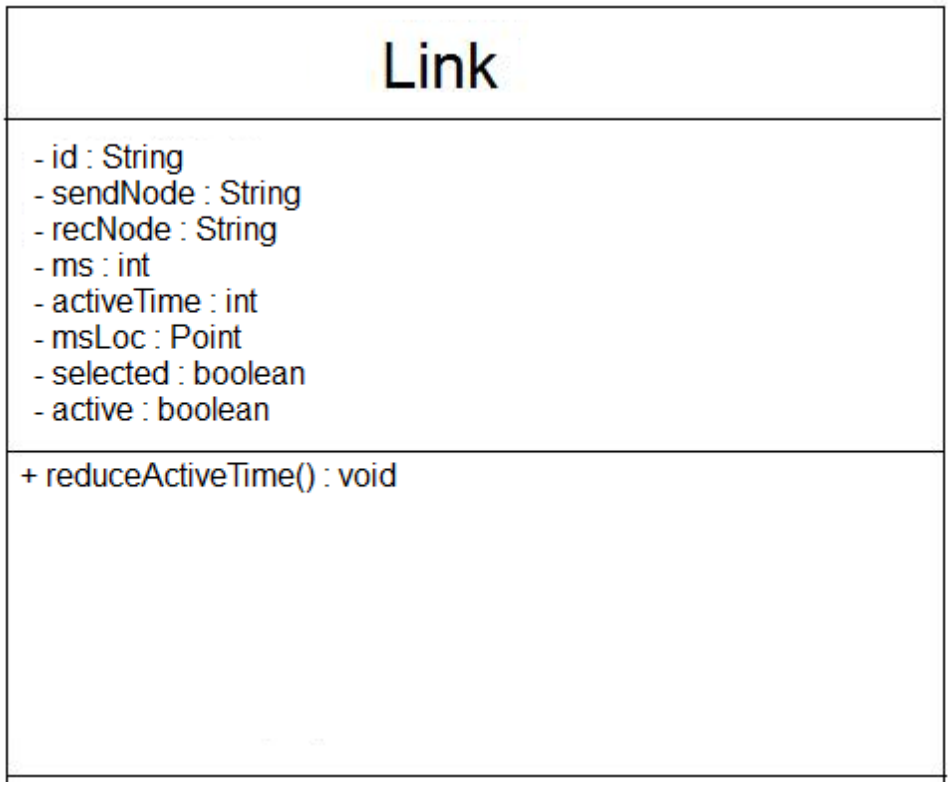


The fields nodelist and linklist are used to retain a running storage of the nodes and links within the distributed system. The ID field is used to automatically generate ID numbers for the various nodes and links to be created. The next four fields, which all pertain to file storage, are used to load and save different distributed system configurations.

Although most method names are self-explanatory, we shall describe some of the more involved methods. As can be seen by the existence of the instance method, the Controller class is singleton, meaning there can only ever be one active object of the given class. This is done as we know we shall only ever need one Controller object. The addNode and addLink methods are used to add nodes and links to the current distributed system in memory. They parameters for these methods are directly taken from the communicating myGUI class. The next four methods are used as simple methods to return the lists of objects or a String based representation of these lists. The checkExistence methods are used to decide whether a given node or link is present in the distributed system. The next four methods are individual versions of the four list functions described above. The checkExistenceOfConnectedNodes function is responsible for seeing whether two nodes are actually connected by a link or not. Finally, getNodeCoords is used to return the graphical coordinates of a given node.

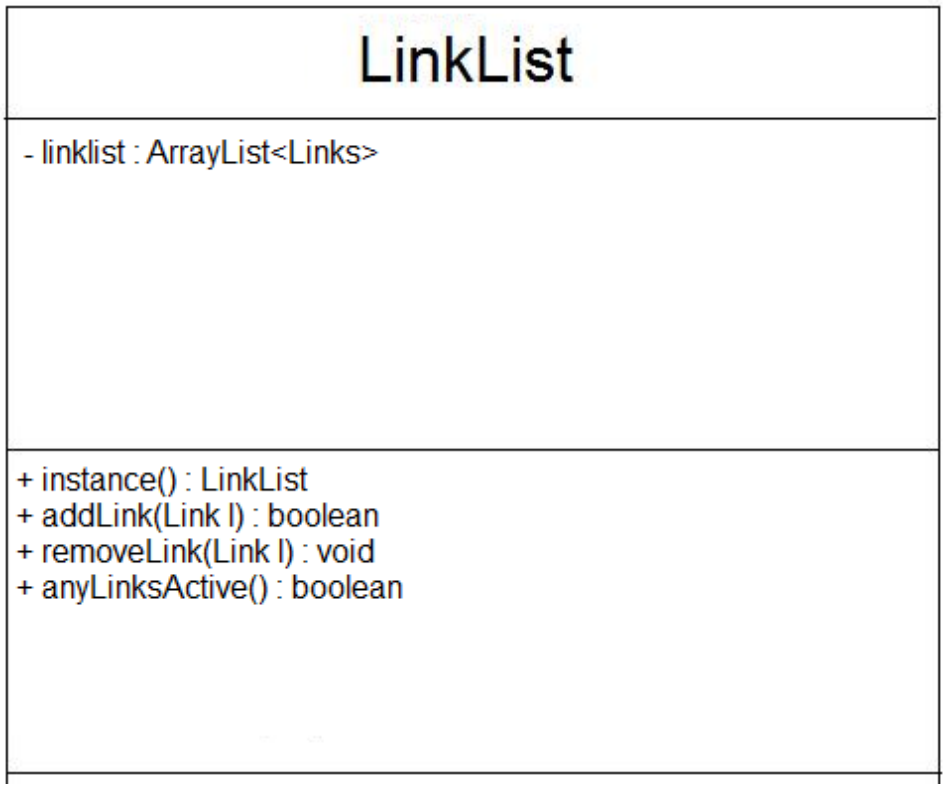
Link Class

The Link class is responsible for representing a link object. Important fields include link id, ms delay, active time and connected nodes. Important methods include reduceActiveTime, which reduces the amount of time remaining for the link to be active. Also included are a plethora of getters and setters.



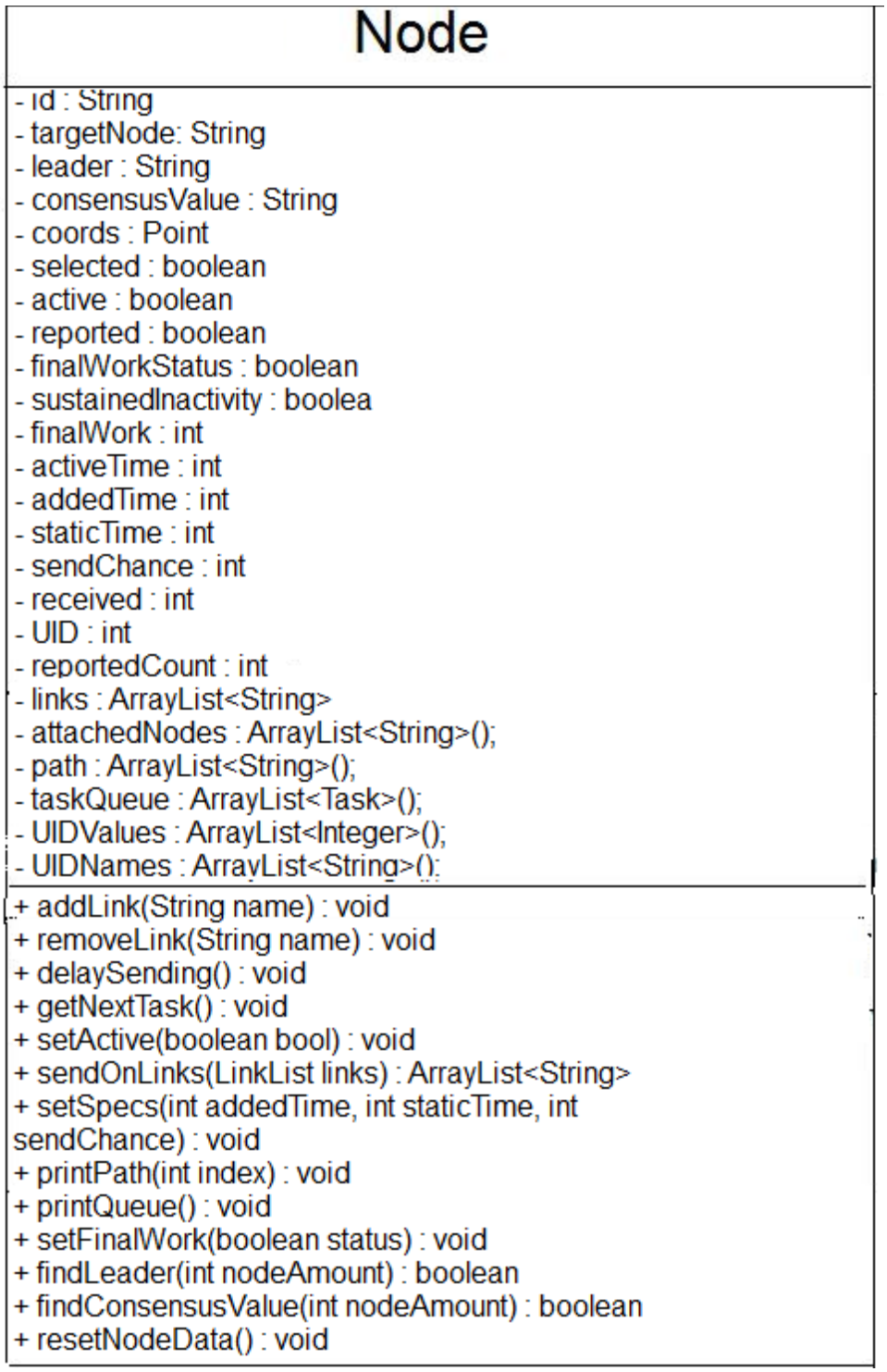
LinkList Class

The LinkList class is responsible for representing a list of Link objects. Like many classes of the project, the LinkList class is singleton as we only ever want one object at any time. Important methods include those that help to manipulate the list of Links.



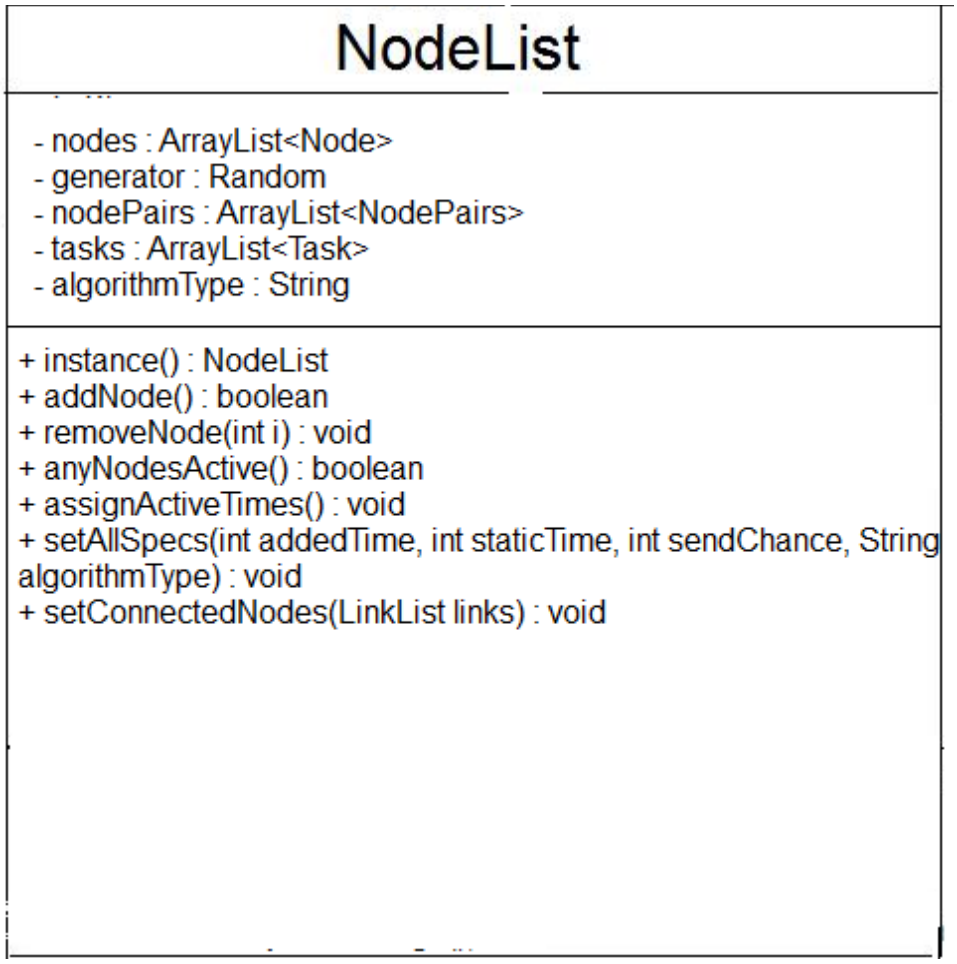
Node Class

The Node class is quite large when compared to the Link class. We can instinctively see this because a node shall have more responsibilities than a link. At this point, all of these responsibilities are contained in the Node class; however, in the future, we wish to break these responsibilities down to a set of classes.



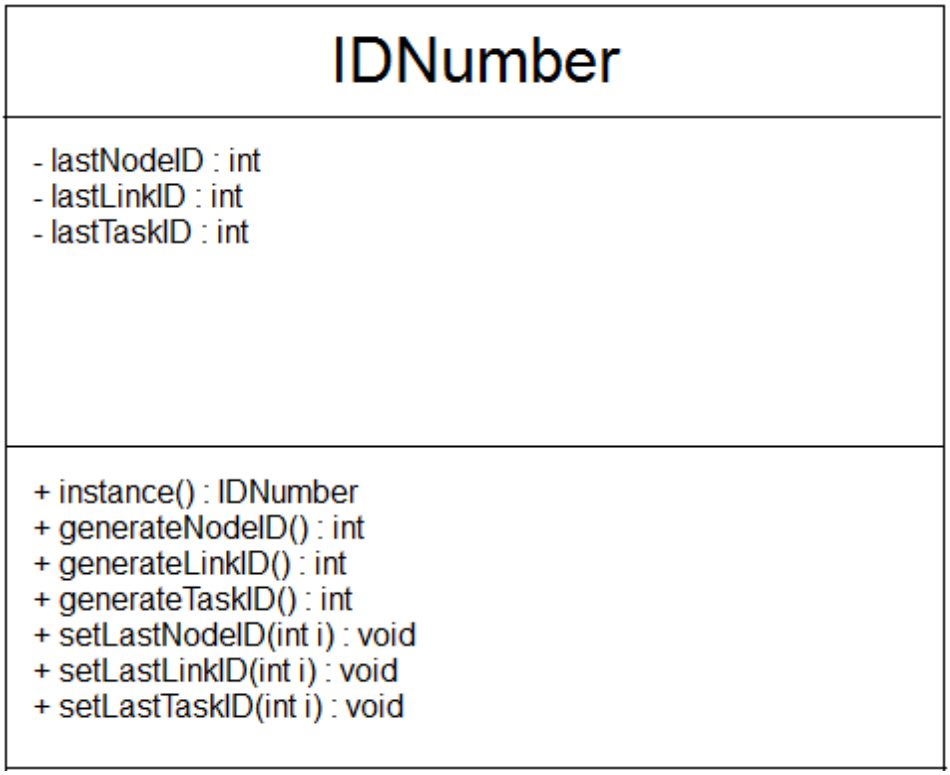
NodeList Class

The NodeList class is responsible for representing a list of Node objects. Like many classes of the project, the NodeList class is singleton as we only ever want one object at any time. Important methods include those that help to manipulate the list of Node objects.



IDNumber Class

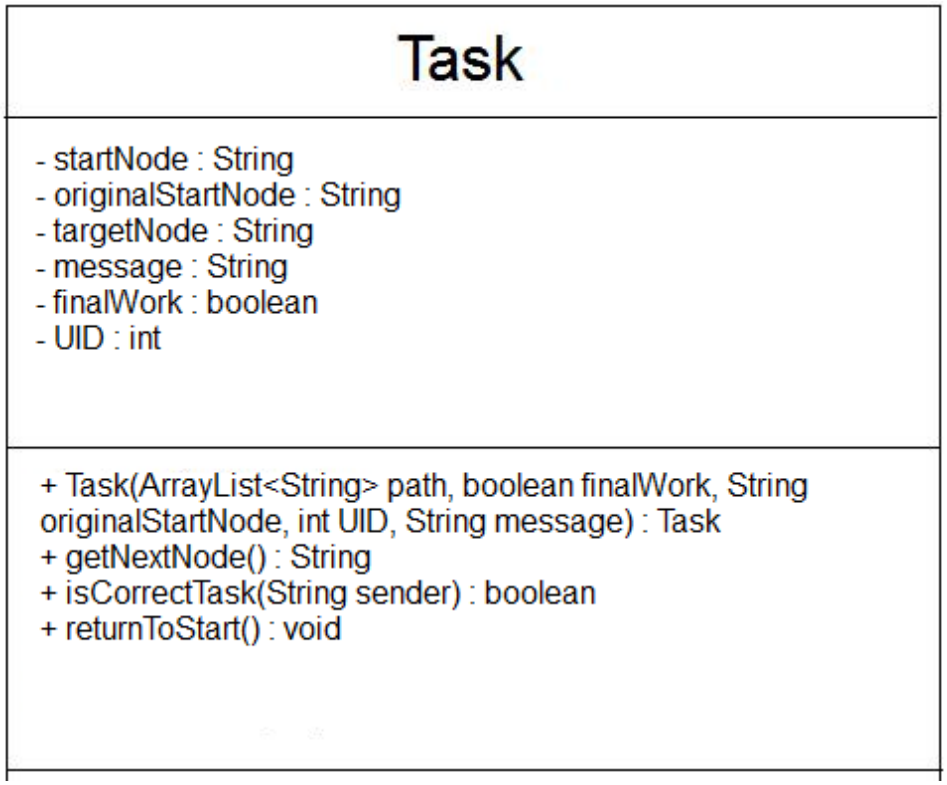
The IDNumber class is a small, simple class that is responsible for generating IDs for the various nodes and links to be created throughout the lifespan of the program.



The three fields retain the last ID of that type that was assigned to an object of the same type. The instance method tells us that the class is singleton. The three generate methods generate a new ID for the given object type. The three setLastID methods do just that and are used for saving/loading purposes.

Task Class

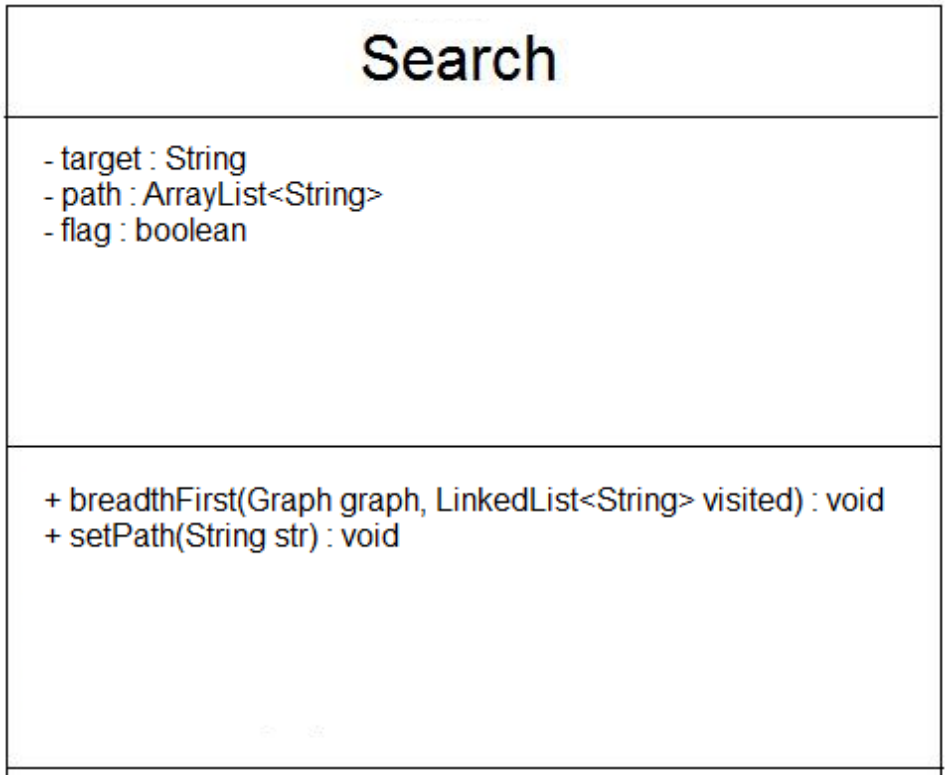
The Task class is a basic object class responsible for representing the tasks to be performed by the system.



Important fields here represent the starting node and target nodes of a particular task. A task also includes a message, UID, and a status of final work. Methods in the Task class are essentially used to update the task's position in the system. Not included here are a good amount of getters and setters.

Search Class

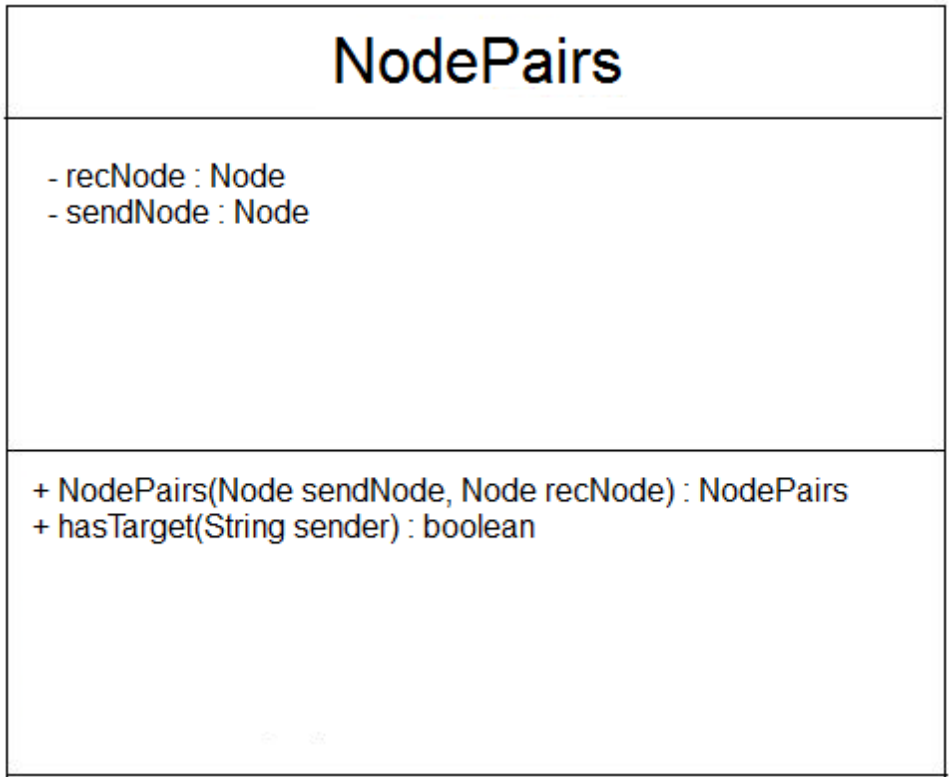
The Search class is responsible for finding the shortest path from node to node in a given system. It mainly utilizes a Breadth-First search to accomplish this task. Note that in an actual implementation of the Final Work approach, this would be done by local routing algorithms at each node.



Although this class is likely small enough to be a simple function within another class, it has been placed in its own class in case the method for finding the shortest path is expanded upon in the future.

NodePairs Class

The NodePairs class is responsible for representing a pair of nodes. At this point, this is chiefly used to represent sending and target node pairs used by the messaging system.



The class really only exists to keep node information paired together in an easy way. It also has getters and setters to allow the easy manipulation of the pairs of nodes.

BIBLIOGRAPHY

[1] Aspnes, J. 2002. *Randomized Protocols For Asynchronous Consensus*. In *Journal of Distributed Computing.* 16, 2-3(Sep. 2003), 165-175. DOI= http://dx.doi.org/10.1007/s00446-002-0081-5.

[2] Babaoglu, O., and Marzullo, K. 1993. *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*. In *Technical Report UBLCS-93-1*. (Jan. 1993), 1-40. DOI= http://dl.acm.org/citation.cfm?id=302434.

[3] Baker, A. H., Crivelli, S., and Jessup E. R. 2008. *An Efficient Parallel Termination Detection Algorithm*. In *International Journal of Parallel, Emergent and Distributed Systems.* 21, 4(2006), 293-301. DOI= www.cs.colorado.edu/department/publications/reports/.../CU-CS-915-01.pdf.

[4] Bertsekas, D.P. and Tsitsiklis, J.N. 1989. *Convergence Rate and Termination of Asynchronous Iterative Algorithms*. In *Proceedings of the 3rd international conference on Supercomputing*. ICS '89(1989), 461-470. DOI= http://dl.acm.org/citation.cfm?id=318894.

[5] Buluc, A. and Madduri, K. 2011. *Parallel Breadth-First Search on Distributed Memory Systems.* In *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.*(2011). DOI= http://dl.acm.org/citation.cfm?id=2063471.

[6] Chlebus, B. S. and Kowalski, D. R. 2009. *Locally Scalable Randomized Consensus for Synchronous Crash Failures.* In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures.* 21 SPAA '09(2009), 290-299. DOI= http://portal.acm.org/citation.cfm?doid=1583991.1584063.

[7] Chandy, K.M. and Lamport, L. 1985. *Distributed Snapshots: Determining Global States of Distributed Systems*. In *ACM Transactions on Computer Systems*. 3, 1(Feb. 1985). 63-75. DOI= http://dl.acm.org/citation.cfm?id=214456.

[8] Chandy, K.M. and Misra, J. 1983. *Distributed Deadlock Detection*. In *ACM Transactions on Computer Systems*. 1, 2(May 1983). 144-156. DOI= http://dl.acm.org/citation.cfm?id=357365.

[9] Dhamdhere, D.M., Iyer, S., and Reddy, E.K.K. 1997. *Distributed Termination Detection for Dynamic Systems*. In *Journal of Parallel Computing*. 22, 14(1997), 2025-2045. DOI= http://dl.acm.org/citation.cfm?id=252010.

[10] Dijkstra, E.W. and Scholten, C.S. 1980. *Termination Detection for Diffusing Computations*. In *Information Processing Letters*. 1980. DOI= http://www.cs.mcgill.ca/~carl/termdtecdifusing.pdf.

[11] Kambayashi, Y., Yoshikawa, M., and Yajima, S. 1982. *Query Processing for Distributed Databases Using Generalized Semi-Joins.* 1 SIGMOD '82(1982), 151-160. DOI= http://portal.acm.org/citation.cfm?id=582381.

[12] King, V., Saia, J., Sanwalani, V., and Vee, E. 2006. *Scalable Leader Election.* In *Proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithms.* 17 SODA '06(2006), 990-999. DOI= http://portal.acm.org/citation.cfm?doid=1109557.1109667.

[13] Kumar, D. 1985. *A Class of Termination Detection Algorithms for Distributed Computations*. In *Technical Report*. (May 1985). 1-39. DOI= http://dl.acm.org/citation.cfm?id=900410&preflayout=tabs.

[14] Lai, T.H. and Yang, T.H. 1986. *On Distributed Snapshots*. In *Journal of Information Processing*

Letters. 25, 3(May 1987). 153-158. DOI= http://dl.acm.org/citation.cfm?id=31090.

[15] Lynch, N. A., 1996. *Distributed Algorithms.* (B. M. Spatz and D. D. Cerra, Eds.). Morgan Kaufmann Publishers.

[16] Mahapatra, N.R. and Dutt, S. 2007. *An Efficient Delay-Optimal Distributed Termination Detection Algorithm*. In *Journal of Parallel and Distributed Computing*. 67, 10(2007), 1047-1066. DOI= http://dl.acm.org/citation.cfm?id=1288057.

[17] Marathe, M., Panconesi, A. and Risinger L. D. 2004. *An Experimental Study of a Simple, Distributed Edge-Coloring Algorithm.* In ACM journal of Experimental Algorithmics(2004), 1-22. DOI= http://dl.acm.org/citation.cfm?id=1005813.1041515.

[18] Matocha, J. 1998. *Distributed Termination Detection in a Mobile Wireless Network*. In *Proceedings of the 36th annual Southeast regional conference*. ACM-SE 36(1998), 207-213. DOI= http://dl.acm.org/citation.cfm?id=275360.

[19] Matocha, J. and Camp, T. 1998. A Taxonomy of Distributed Termination Detection Algorithms. In *The Journal of Systems and Software*. 43(1998), 207-221. DOI= http://ranger.uta.edu/~weems/NOTES4351/TDtaxonomy.pdf.

[20] Mattern, F. 1987. *Algorithms for Distributed Termination Detection.* In *Journal of Distributed Computing.* 2, 3(1987), 161-175. DOI= www.vs.inf.ethz.ch/publ/papers/mattern-dc-1987.pdf.

[21] Mattern, F. 1988. *Experience with a New Distributed Termination Detection Algorithm*. In *Proceedings of the 2nd International Workshop on Distributed Algorithms.* 2(1988), 127-143. DOI= http://dl.acm.org/citation.cfm?id=674863.

[22] Mattern, F. 1988. *Virtual Time and Global States of Distributed Systems*. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. (Oct. 1988), 120-134. DOI= citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.63.4399.

[23] Misra, J. 1983. *Detecting Termination of Distributed Computations Using Markers*. In *Proceedings of the second annual ACM symposium on Principles of distributed computing.* PODC '83(1983), 290-294. DOI= http://dl.acm.org/citation.cfm?id=806729.

[24] Misra, J. and Chandy, K. M. 1982. *Termination Detection of Diffusing Computations in Communicating Sequential Processes.* In *Journal of ACM Transactions on Programming Languages and Systems.* 4, 1(Jan. 1982), 37-43. DOI= http://portal.acm.org/citation.cfm?doid=357153.357156.

[25] Mittal, N., Freiling, F.C., Venkatesan, S., and Penso, L.D. 2005. *Efficient Reduction for Wait-Free Termination Detection in a Crash-Prone Distributed System*. In Proceedings of the 19th International Conference on Distributed Computing. 19(2005), 93-107. DOI= http://dl.acm.org/citation.cfm?id=2162329

[26] Mittal, N., Venkatesan, S., and Peri, S. 2007. *A Family of Optimal Termination Detection Algorithms*. In *Journal of Distributed Computing.* 20, 2(2007), 141-162. DOI= http://dx.doi.org/10.1007/s00446-007-0031-3.

[27] Parvedy, P. R. and Raynal, M. 2004. *Optimal Early Stopping Uniform Consensus in Synchronous Systems with Process Omission Failures*. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. 16 SPAA '04(2004), 302-310. DOI= http://portal.acm.org/citation.cfm?doid=1007912.1007963.

[28] Peri, S. and Mittal, N. 2008. *Improving the Efficacy of a Termination Detection Algorithm*. In *Journal of Information Science and Engineering*. 24, 1(Jan. 2008), 159-174. DOI=

http://www.utdallas.edu/~neerajm/publications/index.html.

[29] Rana, S.P. 1983. *A Distributed Solution to the Termination Detection Problem*. In *Information Processing Letters*. 17(1983), 43-46. DOI= http://www.cs.mcgill.ca/~lli22/575/termination2.pdf.

[30] Savari, S.A. and Bertsekas, D.P. 1996. *Finite Termination of Asynchronous Iterative*. In *Journal Parallel Computing Algorithms*. 22, 1(Jan. 1996). 39-56. DOI= http://dl.acm.org/citation.cfm?id=231676.

[31] Stefanescu, D. C., Thomo, A. and Thomo, L.2005. *Distributed Evaluation of Generalized Path Queries.* In *Proceedings of the 2005 ACM symposium on Applied computing.* SAC '05(2005), 610-616. DOI= http://dl.acm.org/citation.cfm?id=1066819.

[32] Tel, G. and Mattern, F. 1993. *The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes*. In *Journal of ACM Transactions on Programming Languages and Systems.* 15, 1(Jan. 1993), 1-35. DOI= http://dl.acm.org/citation.cfm?id=151646.151647.

[33] Venkatesan, S. 1989. *Reliable Protocols for Distributed Termination Detection.* In *IEEE Transactions on Reliability*. 38, 1(Apr. 1989). 103-110. DOI= http://utdallas.edu/~venky/pubs/RelTerDet.pdf.