

May 2016

A Simulator Application for Distributed Leader Election Algorithms

Sugeeswara Gurudeniya
University of Nevada, Las Vegas

Follow this and additional works at: <https://digitalscholarship.unlv.edu/thesesdissertations>



Part of the [Computer Sciences Commons](#)

Repository Citation

Gurudeniya, Sugeeswara, "A Simulator Application for Distributed Leader Election Algorithms" (2016).
UNLV Theses, Dissertations, Professional Papers, and Capstones. 2677.
<http://dx.doi.org/10.34917/9112075>

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

A SIMULATOR APPLICATION FOR DISTRIBUTED
LEADER ELECTION ALGORITHMS

by

Sugeeswara Gurudeniya
Bachelor of Science in Mathematics
University of Peradeniya, Sri Lanka
2006

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2016

Copyright 2016 by Sugeeswara Gurudeniya
All Rights Reserved



Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 21, 2016

This thesis prepared by

Sugeeswara Gurudeniya

entitled

A Simulator Application for Distributed Leader Election Algorithms

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Ajoy Datta, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Yoohwan Kim, Ph.D.
Examination Committee Member

John Minor, Ph.D.
Examination Committee Member

Venkatesan Muthukumar, Ph.D.
Graduate College Faculty Representative

Abstract

We present an application program, Distributed Algorithm Simulator, to simulate the execution of distributed leader election algorithms in a ring-network. The application was developed using Visual C# on Microsoft .NET Framework 4.5. The Distributed Algorithm Simulator consists of two major components: A Visual Simulator, which visually demonstrates the execution of the algorithms; and a Textual Simulator, which simulates the execution in text format. In both cases the end-result can be saved to a file.

The Visual Simulator displays the network in a ring orientation with circles representing the nodes, and numbers on them showing the node IDs. The user has the ability to choose which variables of the algorithm are displayed at each step and the speed at which each step is performed. Once a simulation has been finished, the user can step through the execution of the algorithm forward and backward.

The Textual Simulator displays – in a multiline Textbox – the status of each variable at each step during the execution. As before, the user can run and pause the simulation as well as control the speed of the execution. Finally, the user can save the results to a text file.

Acknowledgements

I would like to express my sincere gratitude to my thesis advisor Dr. Ajoy K. Datta for giving me the opportunity to work with him on this thesis. Without his guidance, support, and encouragement this thesis would not have come to fruition.

Furthermore, I would like to thank my committee members, Dr. Yoohwan Kim, Dr. John Minor, and Dr. Venkatesan Muthukumar for their support and for being part of my thesis committee.

Finally, I would like to thank my wife, Sandamali Weerasooriya, for her constant support and encouragement through the good times and the bad alike.

SUGEESWARA GURUDENIYA

University of Nevada, Las Vegas

May 2016

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Tables	vii
List of Figures	xi
1 Introduction.....	1
1.1 Contribution.....	1
1.2 Outline	2
2 Background	4
2.1 Introduction to Distributed Computing	4
2.2 History of Distributed Computing	5
2.3 Applications of Distributed Computing	6
3 Distributed Leader Election Algorithms.....	10
3.1 Terminology	10
3.2 Leader Election Problem	13
3.3 Leader Election in Ring Networks.....	15
3.4 Distributed Leader Election Algorithms in Ring Networks.....	15
4 Simulator Application – Design.....	22

4.1	Introduction	22
4.2	System Requirements	26
4.3	Design	26
4.4	UML Diagrams.....	29
5	Simulator Application – Implementation	42
5.1	Source Code Documentation	42
5.2	Explanation of Crucial Components	68
6	User Guide	85
6.1	Download	85
6.2	Installation	85
6.3	Distributed Algorithm Simulator Main Window.....	90
6.4	Visual Simulator.....	103
6.5	Textual Simulator	112
6.6	Error Messages	117
7	Conclusion and Future Work	122
7.1	Conclusion.....	122
7.2	Future Work	122
	Bibliography	124
	Curriculum Vitae	126

List of Tables

Table 1: Classes of ring networks.	13
Table 2: Action table of LCR algorithm.....	18
Table 3: Action table of UNIQUE_k algorithm.....	21
Table 4: Classes of DistributedAlgorithmSimulator namespace.	43
Table 5: Enumerations of DistributedAlgorithmSimulator namespace.....	43
Table 6: Algorithm enumeration syntax.....	44
Table 7: Members of Algorithm enumeration.....	44
Table 8: Common class syntax.	44
Table 9: Fields of Common class.....	45
Table 10: DistributedAlgorithmSimulator class syntax.....	46
Table 11: DistributedAlgorithmSimulator class constructor.....	46
Table 12: DistributedAlgorithmSimulator class methods.	47
Table 13: ExtensionMethods class syntax.	48
Table 14: ExtensionMethods class methods.....	48
Table 15: FileOperations class syntax.	49
Table 16: FileOperations class methods.....	49
Table 17: History class syntax.....	49
Table 18: History class constructor.	49
Table 19: History class properties.....	50

Table 20: NodeControl class syntax.	50
Table 21: NodeControl class constructor.	50
Table 22: NodeControl class methods.....	51
Table 23: NodeInfoControl class syntax.	51
Table 24: NodeInfoControl class constructor.....	51
Table 25: TextualSimulator class syntax.....	52
Table 26: TextualSimulator class constructor.	52
Table 27: TextualSimulator class methods.	52
Table 28: VisualSimulator class syntax.	53
Table 29: VisualSimulator class constructor.....	53
Table 30: VisualSimulator class methods.	54
Table 31: Classes of WrapperClasses namespace.....	55
Table 32: Structures of WrapperClasses namespace.	55
Table 33: Message class syntax.....	56
Table 34: Node class syntax.	56
Table 35: Node class properties.	56
Table 36: Node class methods.....	57
Table 37: NodeState class syntax.	57
Table 38: NodeState class constructor.....	57
Table 39: NodeState class properties.	58
Table 40: NodeState class methods.	58

Table 41: NoteState.Item structure syntax.....	58
Table 42: NodeState.Item structure properties.....	58
Table 43: Utility class syntax.	59
Table 44: Utility class methods.	59
Table 45: Utility class fields.	59
Table 46: Classes of DistributedAlgorithms namespace.....	60
Table 47: LCRMsg class syntax.	60
Table 48: LCRMsg class constructor.	61
Table 49: LCRMsg class properties.	61
Table 50: LCRNode class syntax.....	61
Table 51: LCRNode class constructor.....	61
Table 52: LCRNode class properties.....	62
Table 53: LCRNode class methods.	62
Table 54: LCRUtility class syntax.....	63
Table 55: LCRUtility class constructor.	63
Table 56: LCRUtility class methods.....	63
Table 57: UniqueKMsg class syntax.	64
Table 58: UniqueKMsg class constructor.....	64
Table 59: UniqueKMsg class properties.....	64
Table 60: UniqueKNode class syntax.	65
Table 61: UniqueKNode class constructor.....	65

Table 62: UniqueKNode class properties.	66
Table 63: UniqueKNode class methods.....	66
Table 64: UniqueKUtility class syntax.	67
Table 65: UniqueKUtility class constructor.....	67
Table 66: UniqueKUtility class methods.	68
Table 67: Organization of folders according to namespaces.....	70
Table 68: Visual Simulator button icons and their functionality.....	104
Table 69: Descriptions of counters used in Visual Simulator.....	105

List of Figures

Figure 1: Execution of UNIQUE_k algorithm in the Visual Simulator.	23
Figure 2: Execution of LCR algorithm in the Textual Simulator.	25
Figure 3: Code Map of Distributed Algorithm Simulator	28
Figure 4: Class diagram of the entire system.	30
Figure 5: Class diagram of DistributedAlgorithmSimulator class.	31
Figure 6: Class diagram of ExtensionMethods class.	32
Figure 7: Class diagram of FileOperations class.	32
Figure 8: Class diagram of VisualSimulator class.	33
Figure 9: Class diagram of TextualSimulator class.	34
Figure 10: Class diagram of Node class and its derived classes.	35
Figure 11: Class diagram of Utility class and its derived classes.	36
Figure 12: Class diagram of Message class and its derived classes.	37
Figure 13: Sequence diagram of simulation launch process.	38
Figure 14: Sequence diagram of the initialize process of Visual Simulator.	39
Figure 15: Sequence diagram of the initialize process of Textual Simulator.	40
Figure 16: Sequence diagram of TimerCallback process.	41
Figure 17: Source code organization in Visual Studio IDE.	69
Figure 18: Organization of nodes in a ring network.	71
Figure 19: Relationships between neighboring nodes in a ring network.	72

Figure 20: CreateNetwork() method.....	74
Figure 21: TimerCallback() method.	76
Figure 22: Representation of list of nodes and their LinkIDs.	77
Figure 23: Relationship between node list and message buffer.	78
Figure 24: Receiving a message from the counter-clockwise neighbor.	79
Figure 25: Sending a message to the clockwise neighbor.	80
Figure 26: An instance of a NodeControl control.	80
Figure 27: An instance of a NodeInfoControl control.	80
Figure 28: Code sample of the NodeControl class.	82
Figure 29: Code Sample of the NodeInfoControl Class.....	84
Figure 30: [Preparing to Install] screen.....	86
Figure 31: Distributed Algorithm Simulator splash screen.	86
Figure 32: [Welcome] screen.	87
Figure 33: [Destination Folder] screen.	88
Figure 34: [Ready to Install the Program] screen.	88
Figure 35: [Installing Distributed Algorithm Simulator] screen.....	89
Figure 36: [InstallShield Wizard Complete] screen.	90
Figure 37: Distributed Algorithm Simulator main window.	91
Figure 38: Selecting [Input IDs] radio button.....	93
Figure 39: Entering node IDs.	94
Figure 40: Selecting [Read from File] radio button.....	95

Figure 41: File open dialog box.	96
Figure 42: File path of the selected file.	97
Figure 43: Reading data from the input source.	98
Figure 44: Selecting an algorithm to simulate.	100
Figure 45: Selecting the Visual Simulation type.	101
Figure 46: Selecting the Textual Simulation type.	102
Figure 47: Prompting user to run a Textual Simulation.	103
Figure 48: Visual Simulator initial state.	103
Figure 49: Displaying only selected variables.	106
Figure 50: Executing the simulation.	107
Figure 51: Enabled navigation buttons.	108
Figure 52: [End of the Simulation] message.	109
Figure 53: [Beginning of the Simulation] message.	109
Figure 54: [Save to File] dialog box.	110
Figure 55: Save as a text file.	111
Figure 56: Save as a log file.	111
Figure 57: Textual Simulator initial state.	112
Figure 58: Running the Textual Simulator.	113
Figure 59: Finished status of the Textual Simulation.	114
Figure 60: Output format of the Textual Simulation.	116
Figure 61: [Could not Find the File] error message.	117

Figure 62: [Incorrect Input String Format] error message.	117
Figure 63: [Node IDs not Entered] error message.	118
Figure 64: [Insufficient number of Node IDs] error message for UNIQUE_k.	119
Figure 65: [Insufficient number of Nodes IDs] error message for LCR.	119
Figure 66: [Ring Contains Non-Unique IDs] error message.	120
Figure 67: [Ring does not Contain Repeating IDs] error message.	120
Figure 68: [No Unique ID] error message.....	121

Chapter 1

Introduction

In this chapter we describe the contribution we hope to make with the Distributed Algorithm Simulator presented in this thesis. Then we outline each of the rest of the chapters in this document.

1.1 CONTRIBUTION

In this thesis we present an application to simulate the execution of distributed leader election algorithms in ring networks. The application, henceforth referred to as the Distributed Algorithm Simulator, has two major components; The Visual Simulator and the Textual Simulator. The Visual Simulator can visually simulate the execution of distributed leader election algorithms in networks containing up to 18 nodes. The Textual Simulator does not have an upper limit on the number of nodes.

We hope this application will be a useful tool for education purposes. Because of their inherent distributed nature, it could be somewhat difficult to visualize and understand distributed algorithms. Our application's visual nature, its ability to demonstrate the contents of each variable

at each step, the ability to step forward and backward through the execution, and the ability to save the results so they can be analyzed later should help students overcome those challenges.

1.2 OUTLINE

In Chapter 2 we give an introduction to distributed computing and take a look at its history and applications. First, we briefly introduce the concept of distributed computing and the central idea behind it. Then, we take a look at how and why it came to be widely used by going over the history of distributed computing. Finally, we list and explain a few practical applications of distributed applications that are in use today, focusing on the diversity of disciplines that make use of distributed computing.

In Chapter 3 we introduce the distributed leader election algorithms. We start by defining the terminology used throughout this document. Then, we introduce the leader election problem by formally defining it, and move on to the leader election problem in networks of ring topology, which is the focus of the Distributed Algorithm Simulator. Finally, we introduce the two distributed leader election algorithms we have chosen to implement. First we informally describe them, and then we give a formal definition including all the steps that must be performed during the execution of the algorithms.

Chapter 4 elaborates the design of the application. First, we introduce and explain the main components of the application and their inner workings. Later, using class diagrams we describe the structure of the application, and we use sequence diagrams to explain core processes of the application.

Chapter 5 takes a similar approach and describes in detail the implementation of the Distributed Algorithm Simulator. First, we present documentation of all the namespaces, classes, methods, and properties used in implementing the system. We give a description of each of those including their types, parameters, return values, and usage. Then, we focus on the core components of the system such as creating the network topology, the message communication methodology, and how the execution of algorithms is emulated, and give a detailed description using illustrations and code samples.

Chapter 6 is organized as a user guide to the end-user of the Distributed Algorithm Simulator. Using screenshots, we first explain the installation, and then give a description of all 3 windows of the application; namely, the main window, the Visual Simulator window, and the Textual Simulator window. We explain step-by-step how to use the application, and wrap up by giving a list of error messages the application generates.

Finally, we conclude the thesis with Chapter 7, which contains both conclusion and our recommendations for future work.

Chapter 2

Background

In this chapter we give an introduction to distributed computing and take a look at its history and applications. Then, we list and explain a few practical applications of distributed applications that are in use today, focusing on the diversity of disciplines that make use of distributed computing.

2.1 INTRODUCTION TO DISTRIBUTED COMPUTING

The natural world is full of distributed computing. Flocks of birds fly in perfect V-formation. A colony of termites comprising millions of individuals cooperate to build a mound 9 meters tall [1], which is about 1,000 times their body length. To put it into perspective, that would be the equivalent of humans building a skyscraper 1.7 kilometers tall. A school of fish swim in coordination to avoid predators. During the development of an embryo, billions of cells cooperate to make different body parts to ‘put together’ an animal. What all these have in common is distributed processing: a flock of bird doesn’t have a single bird which controls the behavior of the others, no single termite instructs the others on what to do to make a termite mound, neither does a school of fish has a leader choreographing the movements of the entire school, and, finally, embryonic cells are a collection of entities that are not even conscious, let alone having a central

controlling unit. What all of them do have are a set of local rules built into them by the process of evolution, and simply by each individual blindly following those same local rules they achieve these marvelous feats.

The central idea behind distributed computing using computers is much the same; it combines a number of small, relatively less powerful, and often physically distributed computing units (henceforth called nodes) to perform a useful task and achieve a result which otherwise requires a considerably powerful machine. The most crucial aspect of distributed computing is that each node possesses the same algorithm and executes it on its own hardware, exactly the same as, say, termites in a termite colony. All the nodes have their private tasks to complete, but they must still share certain common resources and information, and a certain degree of coordination is necessary in order to successfully complete their individual tasks [2]. Once all the nodes complete their tasks the algorithm terminates, and by then the system as a whole must have achieved some useful result.

2.2 HISTORY OF DISTRIBUTED COMPUTING

In the early days of computing, any task that required large computations and massive processing power invariably called for supercomputers. However, with the price of personal computers rapidly declining while supercomputers remain expensive, an alternative was needed [3].

One early solution to this problem was clustering. There are many forms of clustering, but Beowulf Clustering introduced by Donald Becker and Thomas Sterling in 1993 particularly made an effort to take off-the-shelf computers and put together a cluster that can rival supercomputers

[4]. However, due to a range of problems, such as needing a dedicated network, the lack of security, and the difficulty of writing specialized software, clustering never managed to solve the problem entirely.

Distributed computing is much the same in many ways; it takes a large problem, breaks it into smaller units, and allows many nodes to work together in parallel. The key difference is that distributed computing allows the nodes to be multifunctional and multipurpose computers that can exist anywhere in the world as long as they are connected to the internet which lends in a great deal of flexibility. Whereas in clustering and supercomputing data is generally processed only once, distributed computing allows the distribution of work units to multiple nodes, multiple times. This serves two functions: to drastically decrease the possibilities of processing errors, and to account for processing which is done on slower CPUs. Furthermore, distributed computing focuses on making work units as small as possible so that they can be handled by any computer in the network. All of the above enables us to take advantage of millions of computers connected to the internet all over the world and to work as one system, which is an immensely powerful idea.

2.3 APPLICATIONS OF DISTRIBUTED COMPUTING

Because of its immense collective power, relatively cheap cost, and the ability to utilize millions of computers all around the world, distributed computing has become a major tool in computing in a wide range of fields. There are thousands of ongoing projects representing an array

of disciplines. Following are a few chosen examples to demonstrate the diversity. Each project's discipline is listed inside brackets next to the title.

2.3.1 Einstein@Home (Astrophysics)

Einstein@Home [5] is a volunteer distributed computing project that searches through data from the LIGO (Laser Interferometer Gravitational-Wave Observatory) detectors for evidence of continuous gravitational-wave sources, which are expected from objects such as rapidly spinning non-axisymmetric neutron stars. Running on the Berkeley Open Infrastructure for Network Computing (BOINC) software platform, Einstein@Home is hosted by the University of Wisconsin–Milwaukee and the Max Planck Institute for Gravitational Physics (Albert Einstein Institute, Hannover, Germany). The project had discovered 49 pulsars as of December 2014. As of January 2016, the project is reported to be using 773 active processing units [6].

2.3.2 Big and Ugly Rendering Project (Art)

Big and Ugly Rendering Project (BURP) [7] is a non-commercial distributed computing project using the BOINC framework. It is under development to work as a publicly distributed system for the rendering of 3D graphics. BURP is a free software distributed under the GNU General Public License V3 license.

2.3.3 Climateprediction.net (Climate Study)

Climateprediction.net (CPDN) [8] is a distributed computing project to investigate and reduce uncertainties in climate modelling. It aims to do this by running hundreds of thousands of different

models (a large climate ensemble) using the donated idle time of ordinary personal computers, thereby leading to a better understanding of how models are affected by small changes in the many parameters known to influence the global climate. The project relies on the volunteer computing model using the BOINC framework where voluntary participants agree to run some processes of the project at the client-side on their personal computers after receiving tasks from the server-side for treatment.

CPDN, which is run primarily by Oxford University in England, has harnessed more computing power and generated more data than any other climate modelling project. It has produced over 100 million model years of data so far. As of December 2010, there are more than 32,000 active participants from 147 countries with a total BOINC credit of more than 14 billion, reporting about 90 teraflops (90 trillion operations per second) of processing power. [9]

2.3.4 Folding@home (Molecular Biology)

Folding@home [10] is a distributed computing project for disease research that simulates protein folding, computational drug design, and other types of molecular dynamics. The project uses the idle processing resources of thousands of personal computers owned by volunteers who have installed the software on their systems. Its main purpose is to determine the mechanisms of protein folding, which is the process by which proteins reach their final three-dimensional structure, and to examine the causes of protein misfolding. This is of significant academic interest with major implications for medical research into Alzheimer's disease, Huntington's disease, and many forms of cancer, among other diseases. Folding@home is developed and operated by the Pande

Laboratory at Stanford University, under the direction of Prof. Vijay Pande, and is shared by various scientific institutions and research laboratories across the world.

This project has pioneered the use of GPUs, PlayStation 3s, Message Passing Interface (used for computing on multi-core processors), and some Sony Xperia smartphones for distributed computing and scientific research. The project uses a statistical simulation methodology that is a paradigm shift from traditional computational approaches.

Folding@home is one of the world's fastest computing systems, with a speed of approximately 40 petaFLOPS [11]. This performance from its large-scale computing network has allowed researchers to run computationally expensive atomic-level simulations of protein folding thousands of times longer than formerly achieved. Since its launch on October 1, 2000, the Pande Lab has produced 129 scientific research papers as a direct result of Folding@home.

2.3.5 SETI@home (Astrobiology)

SETI@home [12] is an Internet-based public volunteer computing project employing the BOINC software platform, hosted by the Space Sciences Laboratory, at the University of California, Berkeley, in the United States. Its purpose is to analyze radio signals, searching for signs of extraterrestrial intelligence, and as such, is one of many activities undertaken as part of the worldwide SETI (Search for Extra-Terrestrial Intelligence) effort. SETI@home was released to the public on May 17, 1999, making it the third large-scale use of distributed computing over the Internet for research purposes.

Chapter 3

Distributed Leader Election Algorithms

In this chapter we first introduce the distributed leader election problem, followed by an introduction to the leader election in ring networks which is the focus of this thesis. Then, we introduce the two distributed leader election algorithms we have chosen to implement to be simulated.

3.1 TERMINOLOGY

Below, we formally define and describe a number of terms and symbols that are consistently used throughout this document.

NODE

The units a ring is comprised of. In reality they may be actual computers connected via a LAN or over the Internet, or they may be processes in a multi-core computer.

RING

A network comprised of *nodes*, connected by some medium laid out in a ring orientation. We consider networks of rings of nodes, P_1, P_2, \dots, P_n , for $n \geq 2$. The ring is bidirectional, meaning that information can flow in either direction. For unidirectional algorithms implemented, we choose the direction to be *clockwise*. As such, each node P_i can receive messages only from its counterclockwise (*left*) neighbor, P_{i-1} , and can only send messages to its clockwise (*right*) neighbor, P_{i+1} . We interpret all subscripts modulo n , e.g., $P_{n+1} = P_1$ and $P_0 = P_n$.

We assume the asynchronous message passing model of computation. Each message takes at most one unit of time to reach its destination. However, we will also assume that no message is lost, and if P_i receives several consecutive messages from P_{i-1} while it is idle, P_i will act on them in the order they are received.

For the algorithm given in this document, we assume that no node knows the size of the ring. We also assume that each node P has an ID, $P.id$, which need not be distinct. Comparison is the only operation permitted on IDs. Henceforth, when we say *ring network*, or simply the *ring*, we mean a network which satisfies the above conditions. Let R be the class of all such networks.

ALGORITHM

When we say *algorithm*, we mean a *uniform* distributed algorithm, meaning, a distributed algorithm such that every node has the same code. We will also assume that every computation of an algorithm begins at a configuration where every node is at a designated *initial state*.

ID

An ID used to identify nodes. An ID may or may not be unique within the ring. In case of a unique ID, we refer to it as a UID.

n

Size of the ring in terms of number of nodes.

k

Maximum number of times an ID occurs within the ring. For instance, consider a ring comprised of nodes carrying IDs 1, 2, 2, 5, 5, 5. Then k would be 3 as 5 repeats 3-times. In the case $k = 1$ all nodes carry unique IDs.

STEP

Since messages cannot pass each other in the ring, every computation in our model can be emulated by a synchronous computation. We define the steps accordingly. If a node P executes an action which takes place at time t in the synchronous emulation, we say that P executes that action at *Step* t . Since our model is asynchronous, nodes in different parts of the ring may execute the same step at different times.

ROUND

A round consists of n consecutive steps. Again, since our model is asynchronous, nodes at different parts of the ring may complete a given round at different times.

CLASSES OF RINGS

A *class of rings* is a collection of rings with different combinations of IDs which share a set of common properties. The following table lists and explains the classes of rings which we refer to throughout this document.

Table 1: Classes of ring networks.

Class of Rings	Description
R	All unidirectional ring networks.
$(A \subset R)$	Asymmetric rings.
$(U \subset A)$	Rings with unique IDs.
$U^* (U^* \subset U)$	Rings where at least one ID in the ring is unique.
$K_k (K_k \subset A)$	No ID occurs more than k times where $k \geq 1$ is a given integer. Note: $K_1 = U$.
U_k^*	$U_k^* = U^* \cap K_k$

3.2 LEADER ELECTION PROBLEM

In distributed computing, usually all the nodes in the network are identical except for the IDs they may possess. Leader election is the process of electing a single node as the ‘leader’ in the

network so that it can be distinguished from all the other nodes in the ring [13]. Usually, the node's ID is used for identifying the leader. It is not necessary for *all* the nodes in the network to have UIDs to solve leader election, but there must be at least one UID, and always a node with a UID will be elected as the leader.

Before the leader election algorithm has begun, all nodes including the eventual leader are unaware of the node which will serve as the leader. Once the algorithm has finished execution, the leader must know it is the leader, all the other nodes must know they are not the leader, and they must also know the UID of the leader. For instance, one common practice is to compare UIDs of nodes and elect one among them that fits some criteria such as the largest or the smallest UID in the network.

3.2.1 Formal Definition of Leader Election Problem

An algorithm solves the leader election problem if: [14]

- States of nodes are divided into elected and not elected states. Once elected, it remains as elected.
- In every execution, exactly one node becomes elected and the rest determine that they are not elected.

A valid leader election algorithm must meet the following conditions: [15]

- **Termination:** the algorithm should finish eventually within a finite time once the leader is selected.
- **Uniqueness:** there is exactly one node that considers itself as leader.

- **Agreement:** all other nodes know who the leader is.

3.3 LEADER ELECTION IN RING NETWORKS

The leader election in ring networks refers to the process of electing a leader in a network of ring topology. Any given node is connected to exactly two nodes, referred to as its clockwise neighbor and counter-clockwise neighbor. A ring can be one of two types: unidirectional and bidirectional. In unidirectional rings, messages can be transmitted in only one direction: either clockwise or counter-clockwise. In bidirectional rings, messages may be transmitted in either direction. The Distributed Algorithm Simulator can simulate both types of algorithms, but we have implemented only unidirectional algorithms.

3.4 DISTRIBUTED LEADER ELECTION ALGORITHMS IN RING NETWORKS

3.4.1 LCR Algorithm

The LCR algorithm, proposed by Le Lann, Chang, and Roberts, is a leader election algorithm in a ring network [16]. It uses only unidirectional communication and does not require the knowledge of the size of the ring. The LCR algorithm requires all nodes in the ring have UIDs, and at the end of the execution elects the node with the highest UID.

3.4.1.1 Informal Description

In the first step, each node sends its UID to the clockwise neighbor. When a node receives a message, it compares the UID in the incoming message to its own. If the incoming UID is greater

than its own, it keeps passing the UID; if it is less than its own, it discards the incoming UID; if it is equal to its own, the node declares itself the leader. Afterwards, the leader sends its own UID in a special message to inform other nodes of the elected leader. Once the leader receives this special message back, the algorithm terminates.

3.4.1.2 Formal Description

Each node P has the following variables.

- $P.uid$, integer type, non-negative. It is the UID of the node and does not change.
- $P.init$, Boolean, initially TRUE. Becomes FALSE at the first step.
- $P.active$, Boolean, which indicates that P is *active*. If $\neg P.active$, we say P is *passive*.
Initially all nodes are active, and when the algorithm is finished all but the leader becomes passive. Once a node becomes passive it never becomes active.
- $P.leader$, integer type, initially \perp (undefined). When the algorithm is finished, $P.leader = L.uid$ for each P , where L is the leader.
- $P.is_leader$, Boolean, initially FALSE. For L , $P.is_leader$ becomes TRUE during the execution and remains so for the remainder of the execution. For all $P \neq L$, $P.is_leader$ remains FALSE for the entirety of the execution.
- $P.leader_elected$, Boolean, initially FALSE. Eventually $P.leader_elected$ becomes TRUE for all nodes.

The LCR algorithm uses a message of the form $\langle u, sp \rangle$ where,

- u – Integer, UIDs sent by nodes.
- sp – Boolean, indicates whether it's the special message. If $sp = TRUE$, it is the special message and u is the elected leader; otherwise, a regular message.

Following is an action table which formally describes the LCR algorithm. In the Distributed Algorithm Simulator, this is used as the base for the coding of the algorithm. Letters T and F are used to represent the Boolean values TRUE and FALSE respectively.

Table 2: Action table of LCR algorithm.

Action Number	Action Name	Condition	Action
A1	Start	P.init	Send $\langle P.uid, F \rangle$ P.init $\leftarrow F$
A2	Deactivate	P.active Read $\langle u, F \rangle$ $u > P.uid$	Send $\langle u, F \rangle$ P.active $\leftarrow F$
A3	Terminate Message	P.active Read $\langle u, F \rangle$ $u < P.uid$	(nothing)
A4	Elect Leader	P.active Read $\langle u, F \rangle$ $u = P.uid$	Send $\langle P.uid, T \rangle$ P.is_leader $\leftarrow T$ P.leader_elected $\leftarrow T$ P.leader $\leftarrow P.uid$
A5	Passive Forward	$\neg P.active$ Read $\langle u, F \rangle$	Send $\langle u, F \rangle$
A6	Acknowledge Leader	$\neg P.active$ Read $\langle u, T \rangle$	Send $\langle u, T \rangle$ P.leader_elected $\leftarrow T$ P.leader $\leftarrow u$
A7	Finish	P.active Read $\langle u, T \rangle$ $u = P.uid$	(nothing)

3.4.2 UNIQUE_ k Algorithm

The UNIQUE_ k algorithm [17] is a distributed algorithm designed to solve the leader election problem in unidirectional ring networks. The ring must contain at least one node with a UID, and it may or may not contain nodes with repeating IDs. The algorithm elects the node which has the maximum UID as the leader.

3.4.2.1 Informal Description

The fundamental idea of UNIQUE_k is that a node becomes passive if it reads a message which proves that its own ID is not unique. Eventually, all nodes with non-unique IDs become passive.

This paradigm is implemented by tokens, each of which carries the ID of the node which initialized it. Each time a message is forwarded by a node which has the same ID as that of the message, the message's counter is incremented by one. Thus, the counter in a message is a rough estimate of the frequency of its ID in the ring. Whenever a node can determine that its ID is not unique, it becomes passive. In order for the leader to be uniquely defined, the IDs of the nodes are used as a tie-breaker. If an active node P forwards a token with a larger ID which has the same counter value, and that value is at least 1, then P knows that it is not the leader, and thus becomes passive.

3.4.2.2 Formal Description

Each node P has the following variables.

- $P.id$, of unspecified *label type*, which does not change. Labels can be compared.
- $P.init$, Boolean, initially TRUE, which becomes FALSE at the first step.
- $P.active$, Boolean, which indicates that P is *active*. If $\neg P.active$, we say P is *passive*.

Initially all nodes are active, and when the UNIQUE_k is done the leader is the only active node. A passive node never becomes active.

- $P.count$, an integer in the range $0 \dots k+1$. Initially, $P.count = 0$.

- *P.leader*, of label type, initially *P.id*. When UNIQUE_k is done, $P.leader = L.uid$ for each *P*, where *L* is the leader.
- *P.is_leader*, Boolean, initially FALSE for all *P*. Eventually, *P.is_leader* becomes TRUE and remains TRUE. *P.is_leader* remains FALSE for the entirety of the execution if $P \neq L$.
- *P.leader_elected*, Boolean, initially FALSE for all *P*. Eventually $P.leader_elected = \text{TRUE}$ for all *P*. *P.leader_elected* means that *P* knows a leader has been elected; once TRUE it never becomes false.

UNIQUE_k algorithm uses only one kind of message of the form $\langle x, c \rangle$ where,

- *x* – ID of the original node which generated the message.
- *c* – An integer counter in the range $0 \dots k+1$. Incremented each time the message is forwarded by a node whose ID is equal to *P.id*.

Following is an action table which formally describes the UNIQUE_k algorithm. In the Distributed Algorithm Simulator, this is used as the base for the coding of the algorithm.

Table 3: Action table of UNIQUE_k algorithm.

Action Number	Action Name	Condition	Action
A1	Start	P.init	Send <P.id, 0> P.init ← FALSE
A2	Passive Forward	! P.active Read <x, c> x ≠ P.id c ≤ k	Send <x, c>
A3	Active Forward	P.active Read <x, c> x ≠ P.id P.count = 0 or c > P.count	Send <x, c>
A4	Deactivate	P.active Read <x, c> x ≠ P.id c < P.count	Send <x, c> P.active ← FALSE
A5	Forward Inferior	P.active Read <x, c> x < P.id c = P.count ≥ 1	Send <x, c>
A6	Forward Superior	P.active Read <x, c> x > P.id c = P.count ≥ 1	Send <x, c> P.active ← FALSE
A7	Terminate Message	! P.active Read <x, c> x = P.id	(nothing)
A8	Increment Message	P.active Read <x, c> x = P.id c = P.count ≤ k-1	Send <x, c+1> P.count ← c+1
A9	Elect Leader	P.active Read <x, k> x = P.id P.count = k	Send <x, k+1> P.is_leader ← TRUE P.leader_elected ← TRUE P.count ← k+1
A10	Acknowledge Leader	! P.active Read <x, k+1>	Send <x, k+1> P.leader ← x P.leader_elected ← TRUE
A11	Finish	P.active Read <x, k+1> x = P.id P.count = k+1	(nothing)

Chapter 4

Simulator Application – Design

In this chapter we introduce and explain the main components of the Distributed Algorithm Simulator. Then, using class diagrams we describe the structure of the application, and then we use sequence diagrams to explain the core processes of the application.

4.1 INTRODUCTION

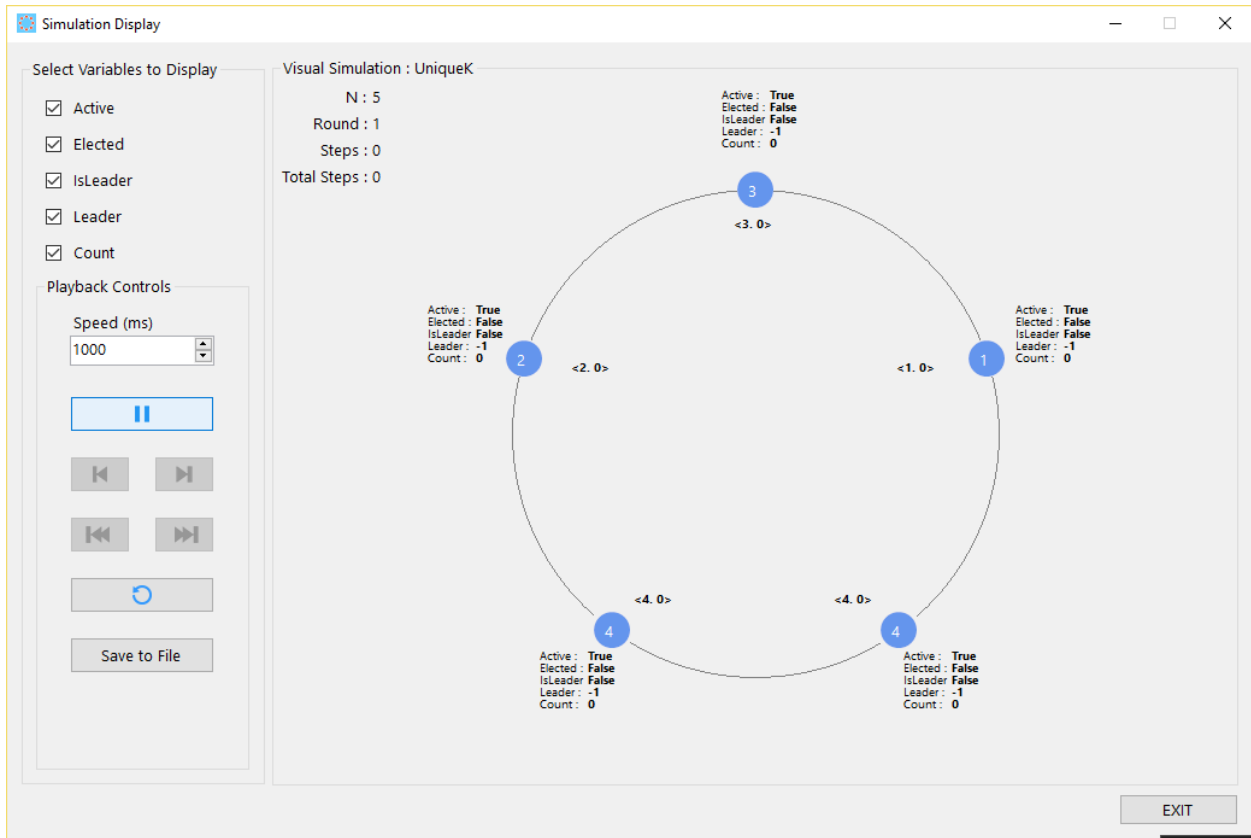
The Distributed Algorithm Simulator is a desktop application that simulates the execution of distributed leader election algorithms in ring networks. It consists of two main components, namely, the Visual Simulator and the Textual Simulator. The application is developed using Microsoft .NET Framework 4.5, and as such can only be run on Microsoft Windows platforms.

4.1.1 Visual Simulator

The Visual Simulator, as the name suggests, simulates the execution of the algorithm visually. The Visual Simulator can simulate the execution on networks containing up to 18 nodes. The restriction is purely due to limitations imposed by the screen size. If the number of nodes is greater than 18, the application will prompt the user with the option of running the Textual Simulator

instead. At the end of the execution the results can be saved to a text file. Following is a screen capture of an intermediate stage of the execution of $UNIQUE_k$ algorithm described previously.

Figure 1: Execution of $UNIQUE_k$ algorithm in the Visual Simulator.



The application lets the user either manually input the IDs of the nodes in the ring into a textbox, or read them from a CSV (Comma Separated Values) file. In either case the IDs must be in a clockwise orientation. Then the user can choose the algorithm to be run and execute it.

This opens up the window shown in the above figure, which prompts the user for more actions. The nodes in the ring are represented by circles. The node IDs are displayed on top of the circles. The nodes are displayed in a clockwise orientation. The Visual Simulator can display the status of the variables used in the algorithm as well as the messages being passed in each step, and the user has the freedom to choose which variables to be displayed. The user can also choose the speed with which the algorithm is executed, the default value of which is 1 second. The [Play/Pause] toggle button begins the execution of the algorithm, which also lets the user pause the execution. [Reset] stops the algorithm and returns to the initial status so a new execution can be started. [Next] and [Previous] lets the user execute the algorithm forward or backward, one step at a time, while [First] and [Last] buttons let the user navigate to the first and last steps of the execution respectively. During the execution, if a node is active and is still in candidacy to be the leader, it is displayed in a blue color. If a node becomes inactive and no longer a candidate to be the leader, it turns red. If a node is elected as the leader, it turns green.

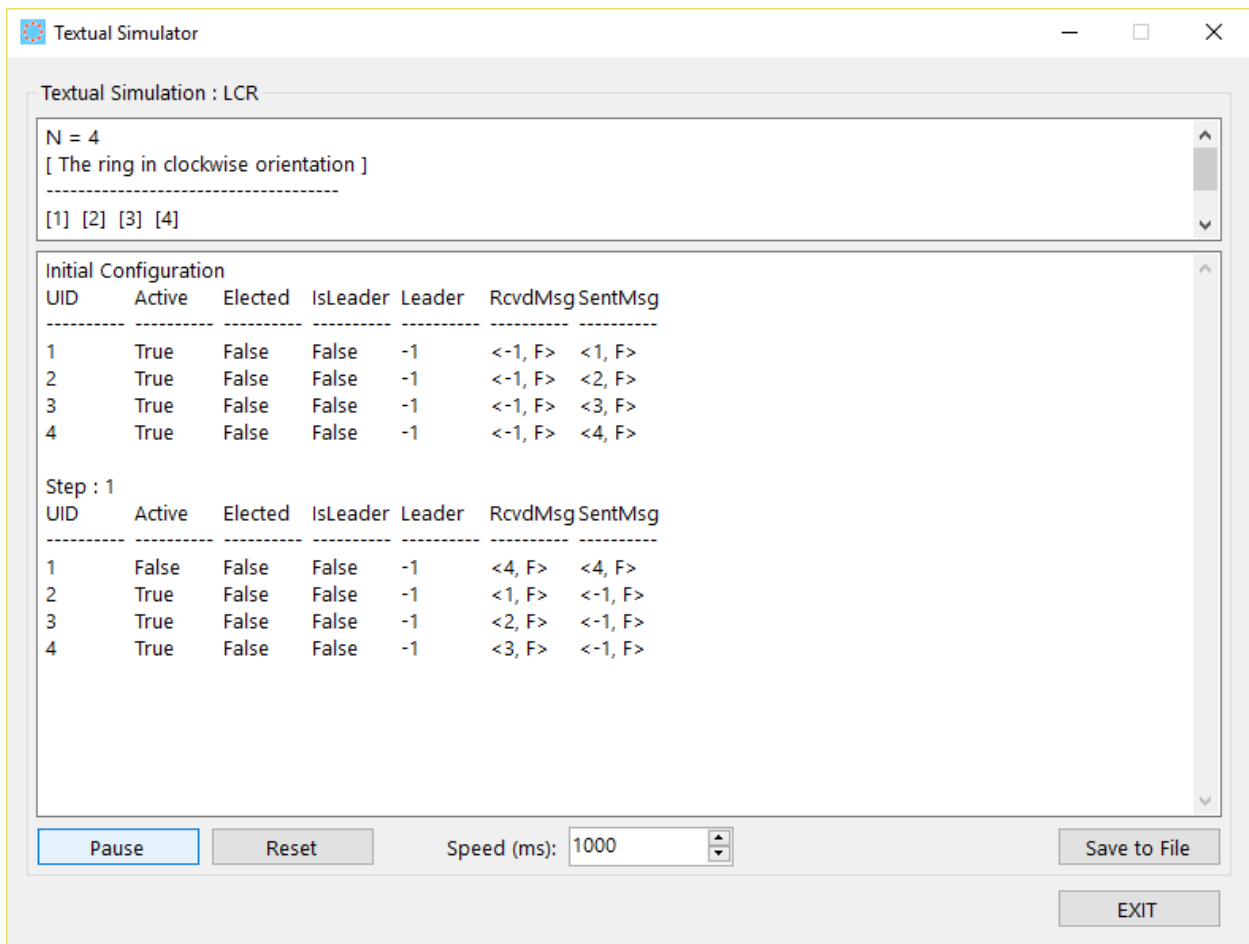
4.1.2 Textual Simulator

The Textual Simulator can execute the algorithm on networks containing any number of nodes. However, due to the time it may take, it is advisable to limit the size of the network to a reasonably small n .

Once the IDs of nodes are either entered by the user or read from a file, the user can check the [Textual] radio button to run the Textual Simulator. The textual simulation is displayed on a new window that opens up. It displays the contents of all the variables in the algorithm, at each

step. The [Run/Pause] toggle button lets the user run and pause the simulation. The simulation can be reset at any stage during the execution using the [Reset] button. Following is a screen capture of an intermediate stage of the execution of LCR algorithm described previously.

Figure 2: Execution of LCR algorithm in the Textual Simulator.



4.2 SYSTEM REQUIREMENTS

The Distributed Algorithm Simulator has been tested on the following operating systems and works correctly.

- Microsoft Windows 8.0
- Microsoft Windows 8.1
- Microsoft Windows 10.0

It has been tested on the following frameworks and works correctly.

- Microsoft .NET 4.5 framework
- Microsoft .NET 4.0 framework

4.3 DESIGN

The program adopts a modular design which keeps different components independent of each other. The Distributed Algorithm Simulator is comprised of 3 namespaces which are listed below.

4.3.1 Namespaces

4.3.1.1 DistributedAlgorithmSimulator Namespace

This is the 'main' namespace of the application. All the System.Windows.Forms form classes that describe the Distributed Algorithm Simulator application windows, and the supplementary classes which are required for their functionality are categorized under this namespace. This includes the Distributed Algorithm Simulator main window, the Visual Simulator, and the Textual Simulator windows.

Classes in this namespace are unaware of the detailed implementation of the actual distributed leader election algorithms found in the DistributedAlgorithms namespace. This modularity gives both the applications and the algorithms a high degree of independence, so the application can be changed with minimal changes to the algorithms and vice versa. The modularity is achieved by making use of the WrapperClasses namespace which acts as a template for the classes in DistributedAlgorithms namespace.

4.3.1.2 WrapperClasses Namespace

This contains abstract classes that are inherited by the classes in DistributedAlgorithms namespace. The DistributedAlgorithmSimulator namespace uses these classes to send and receive messages, execute the algorithms, and get the current status of the execution of the algorithms. These classes dictate the common elements and the rules to which coded algorithms in the DistributedAlgorithms namespace must adhere.

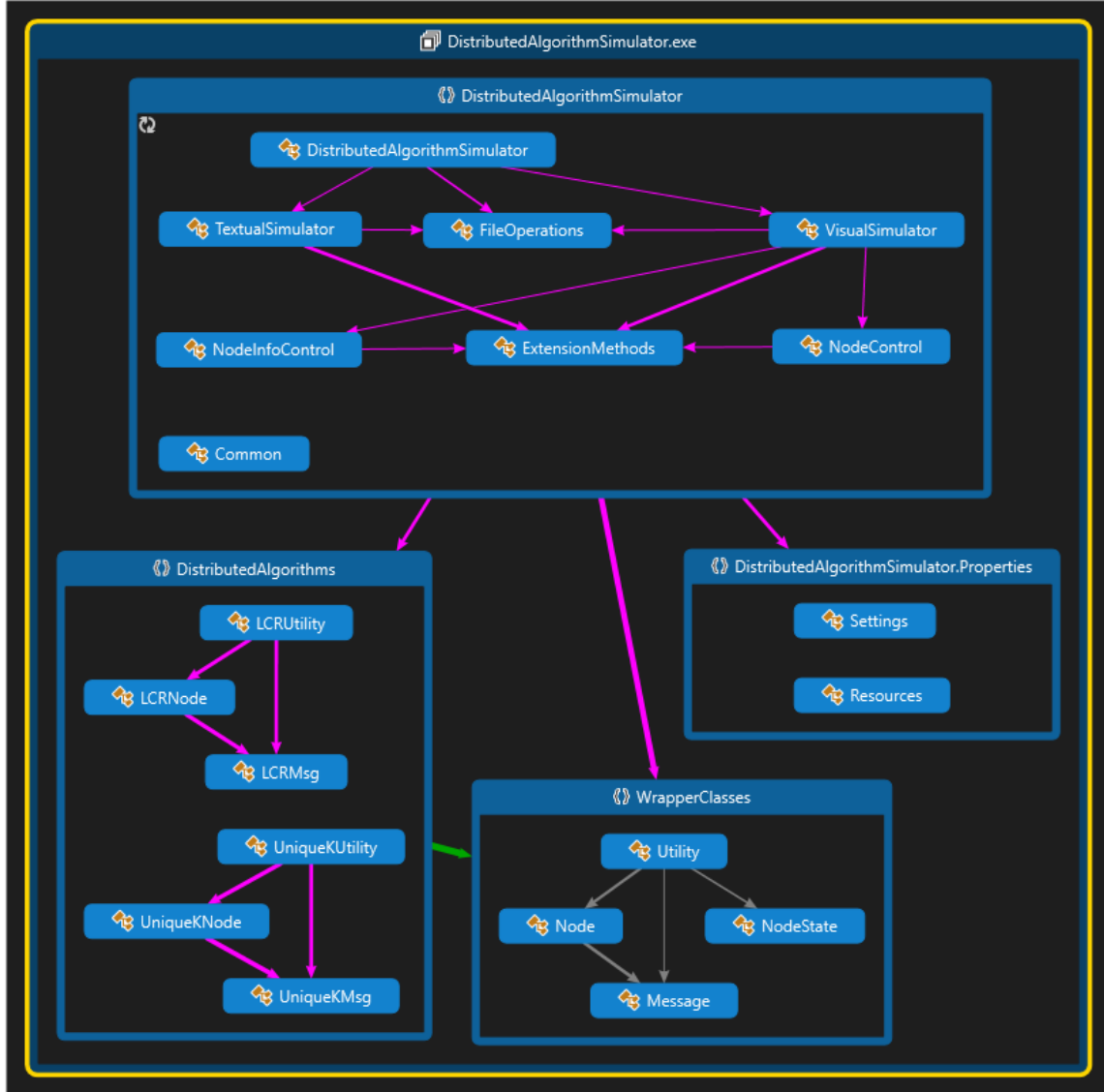
4.3.1.3 DistributedAlgorithms Namespace

Classes in this namespace describe the actual leader election algorithms. It inherits the classes in the WrapperClasses namespace, and then defines each algorithm's unique behavior.

4.3.2 Code Map

The following Code Map illustrates the relationship between namespaces and classes within them.

Figure 3: Code Map of Distributed Algorithm Simulator



A pink arrow denotes a namespace/class calling another namespace/class, with the origin of the arrow denoting the calling party and the arrowhead the called party. A green arrow denotes inheritance with origin of the arrow representing the sub class and arrowhead the super class.

The DistributedAlgorithmSimulator class inside the like named namespace calls upon VisualSimulator and the TextualSimulator classes, while making use of other classes. Those 3 classes together call both WrapperClasses and the DistributedAlgorithms namespace, and in turn, classes within them.

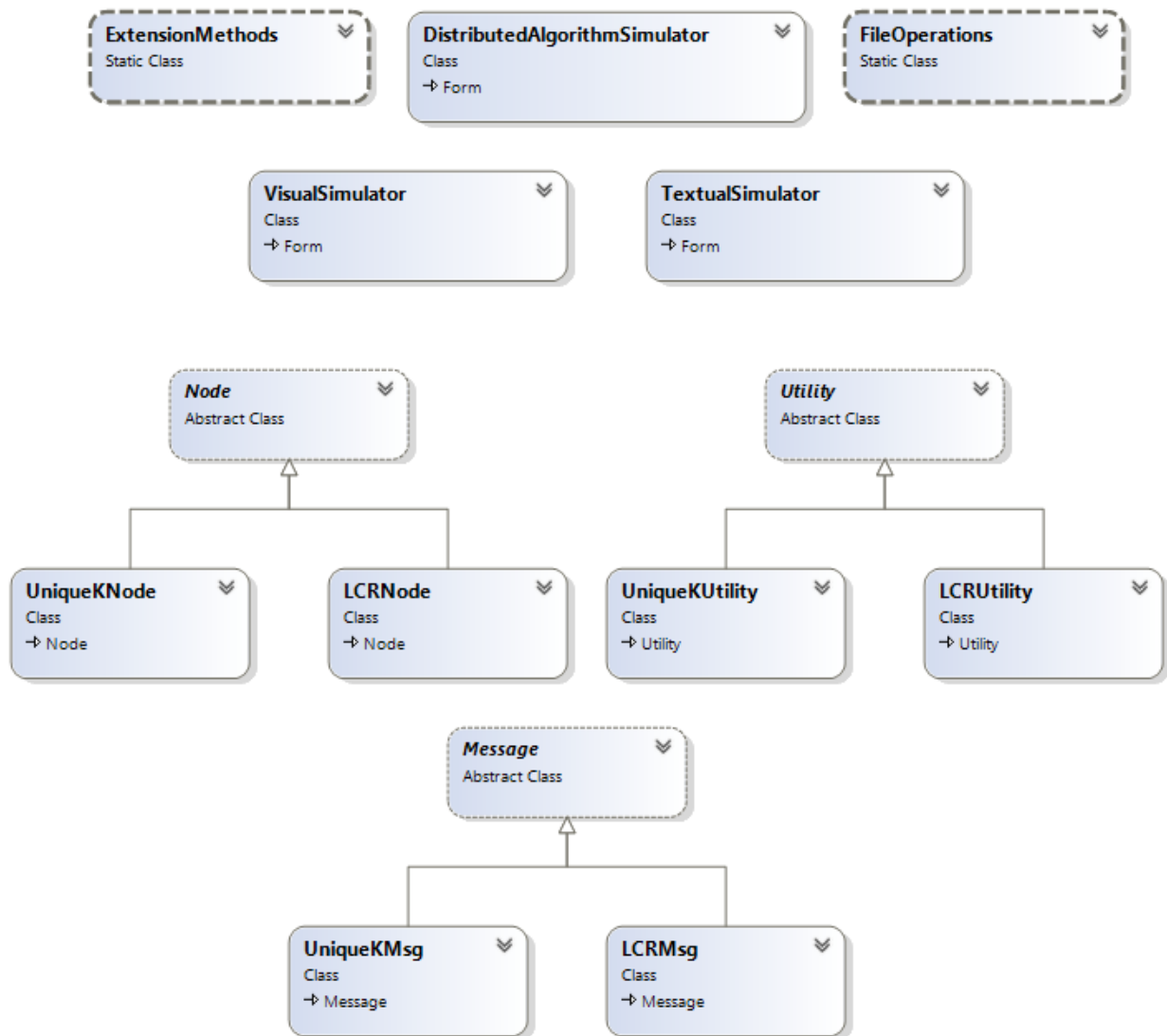
4.4 UML DIAGRAMS

4.4.1 Class Diagrams

4.4.1.1 System Overview

The following class diagram represents the system as a whole.

Figure 4: Class diagram of the entire system.



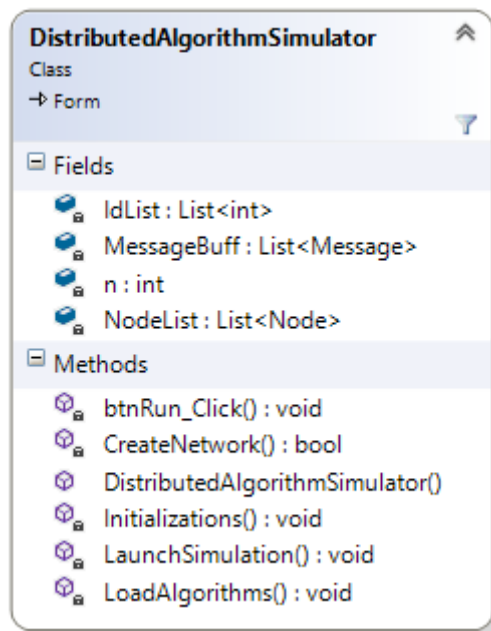
DistributedAlgorithmSimulator, VisualSimulator, and TextualSimulator classes are the three main classes of the application program, each of which is a form window. They make use of ExtensionMethods and FileOperations classes.

Node, Utility, and Message abstract classes and their derived classes offer the functionality of the actual algorithms. The three main classes mentioned above create instances of these classes during the execution of the algorithm.

4.4.1.2 DistributedAlgorithmSimulator Class

Following is the DistributedAlgorithmSimulator class, which is a derived class of the .NET Forms class. It represents the Distributed Algorithm Simulator main window.

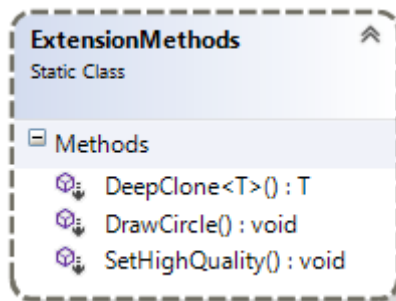
Figure 5: Class diagram of DistributedAlgorithmSimulator class.



4.4.1.3 ExtensionMethods Class

The ExtensionMethods class encompasses the supplementary methods used by the Distributed Application Simulator.

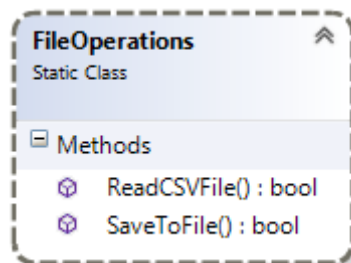
Figure 6: Class diagram of ExtensionMethods class.



4.4.1.4 FileOperations Class

The FileOperations class contains methods for file read/write operations.

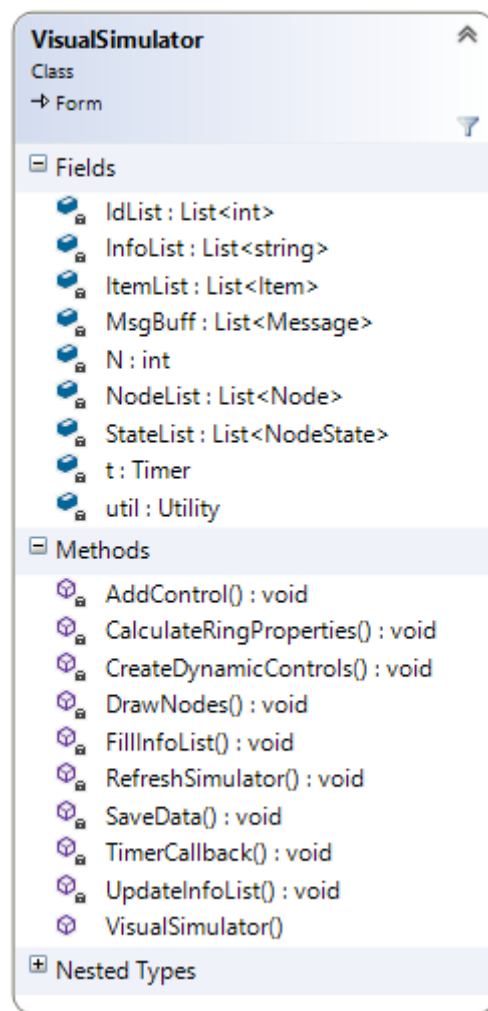
Figure 7: Class diagram of FileOperations class.



4.4.1.5 VisualSimulator Class

The Visual Simulator window class, which is a derived class of the .NET Forms class.

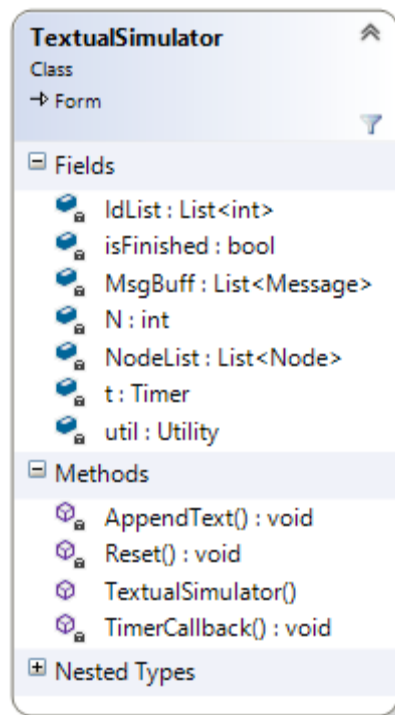
Figure 8: Class diagram of VisualSimulator class.



4.4.1.6 Textual Simulator Class

The Textual Simulator window class is a derived class of the .NET Forms class.

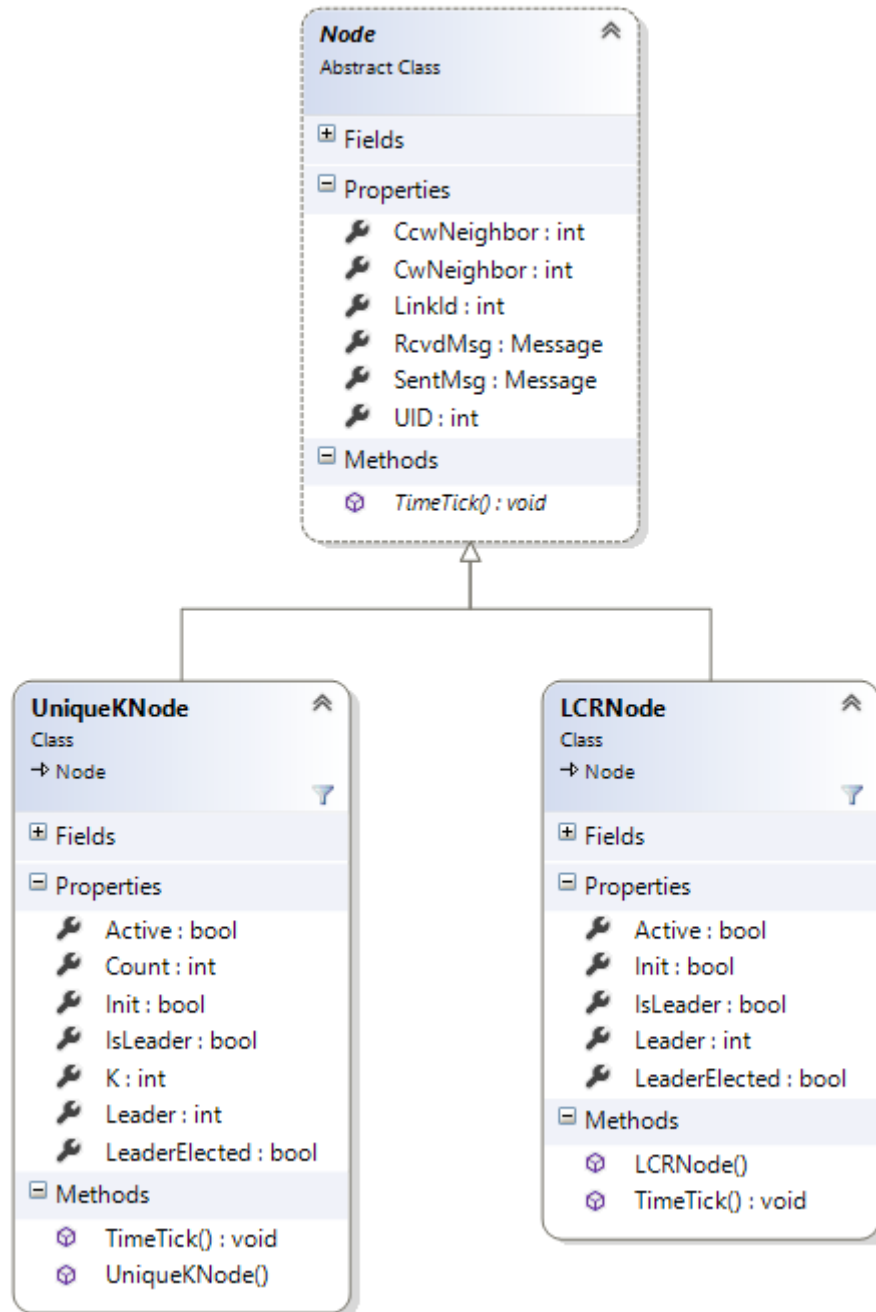
Figure 9: Class diagram of TextualSimulator class.



4.4.1.7 Node Classes

The abstract Node class and its derived classes.

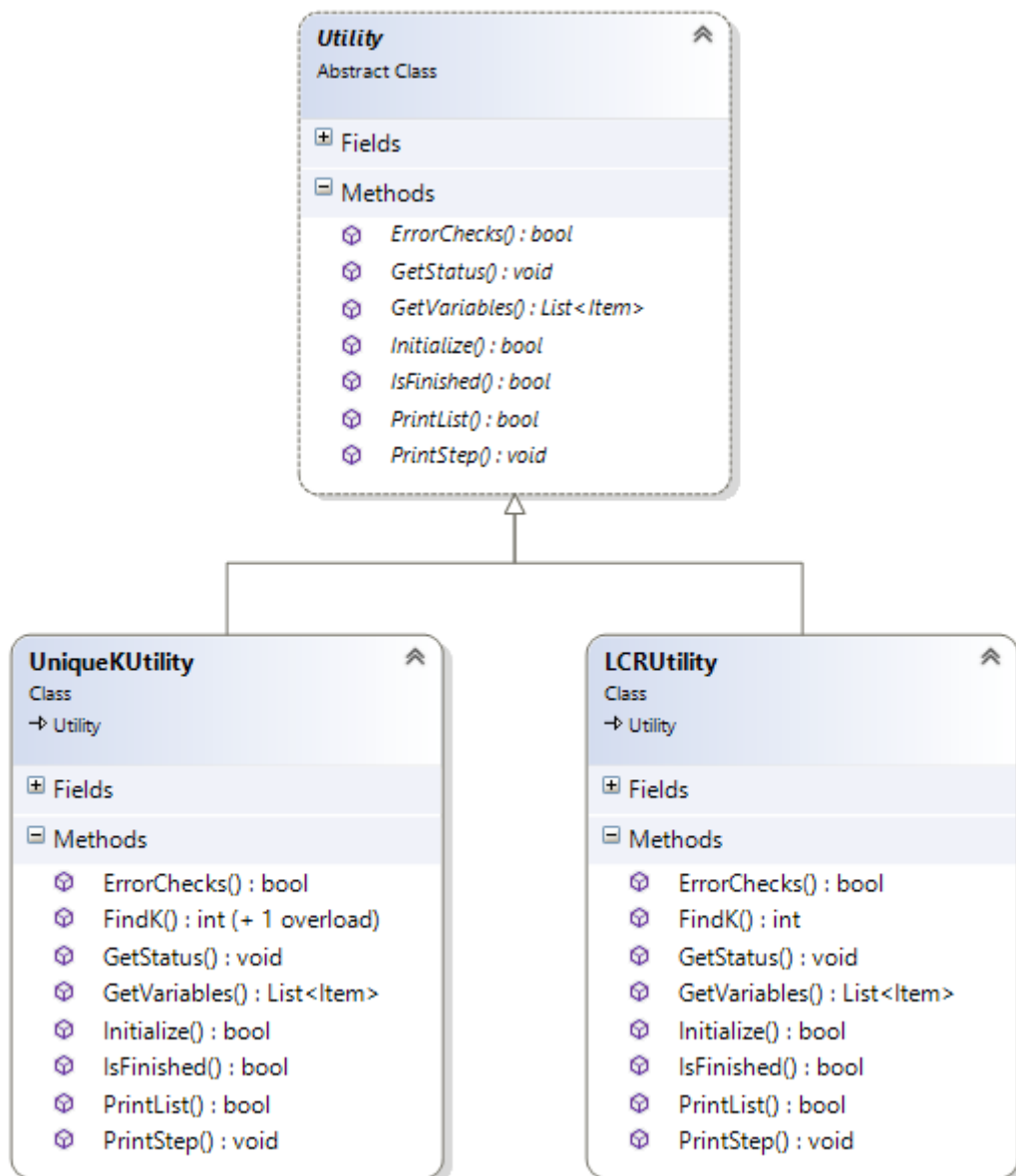
Figure 10: Class diagram of Node class and its derived classes.



4.4.1.8 Utility Classes

The abstract Utility class and its derived classes.

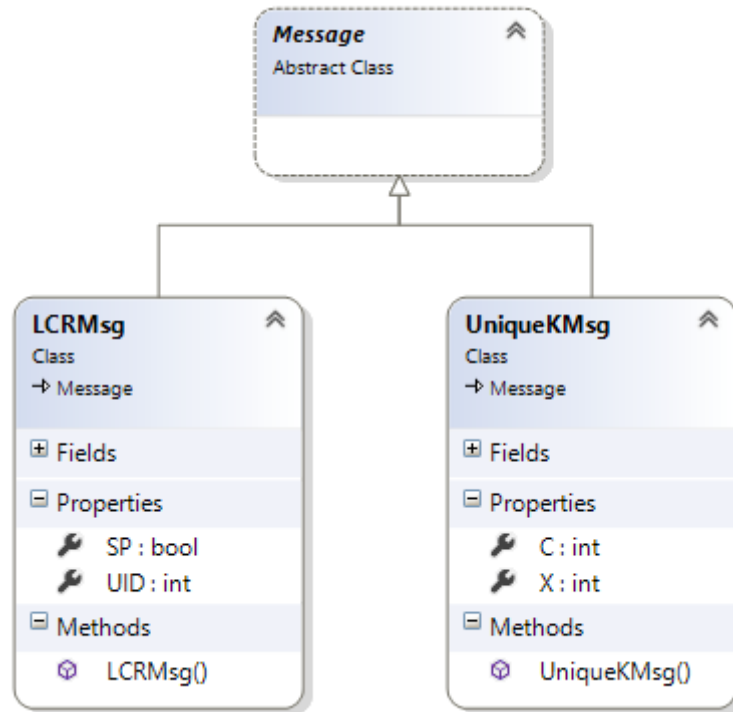
Figure 11: Class diagram of Utility class and its derived classes.



4.4.1.9 Message Classes

The abstract Message class and its derived classes.

Figure 12: Class diagram of Message class and its derived classes.

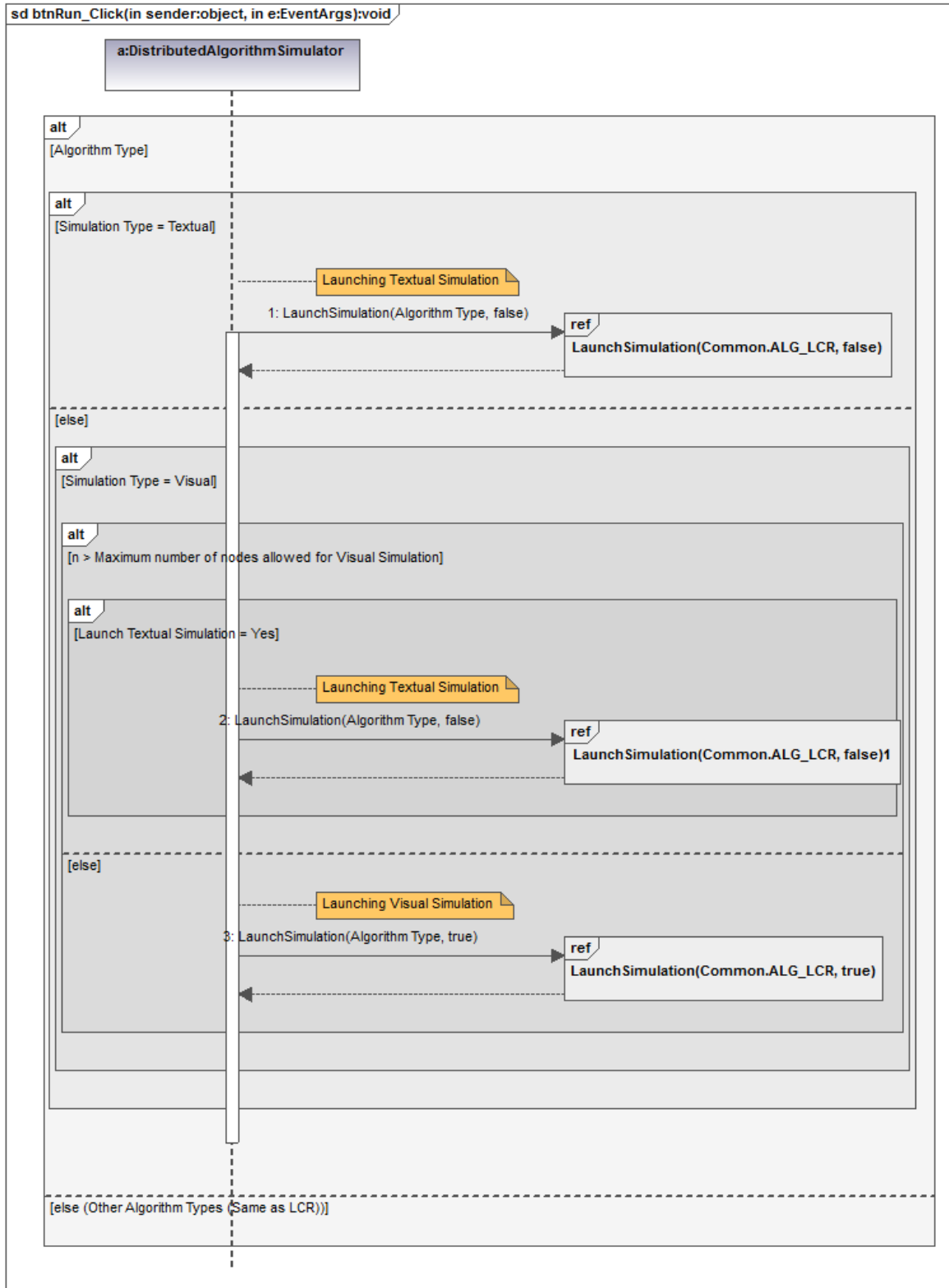


4.4.2 Sequence Diagrams

4.4.2.1 Launching Simulator

The following illustrates the launching sequence of Visual Simulator and Textual Simulator. Depending on user choices the application selects the appropriate algorithm and launches the simulation.

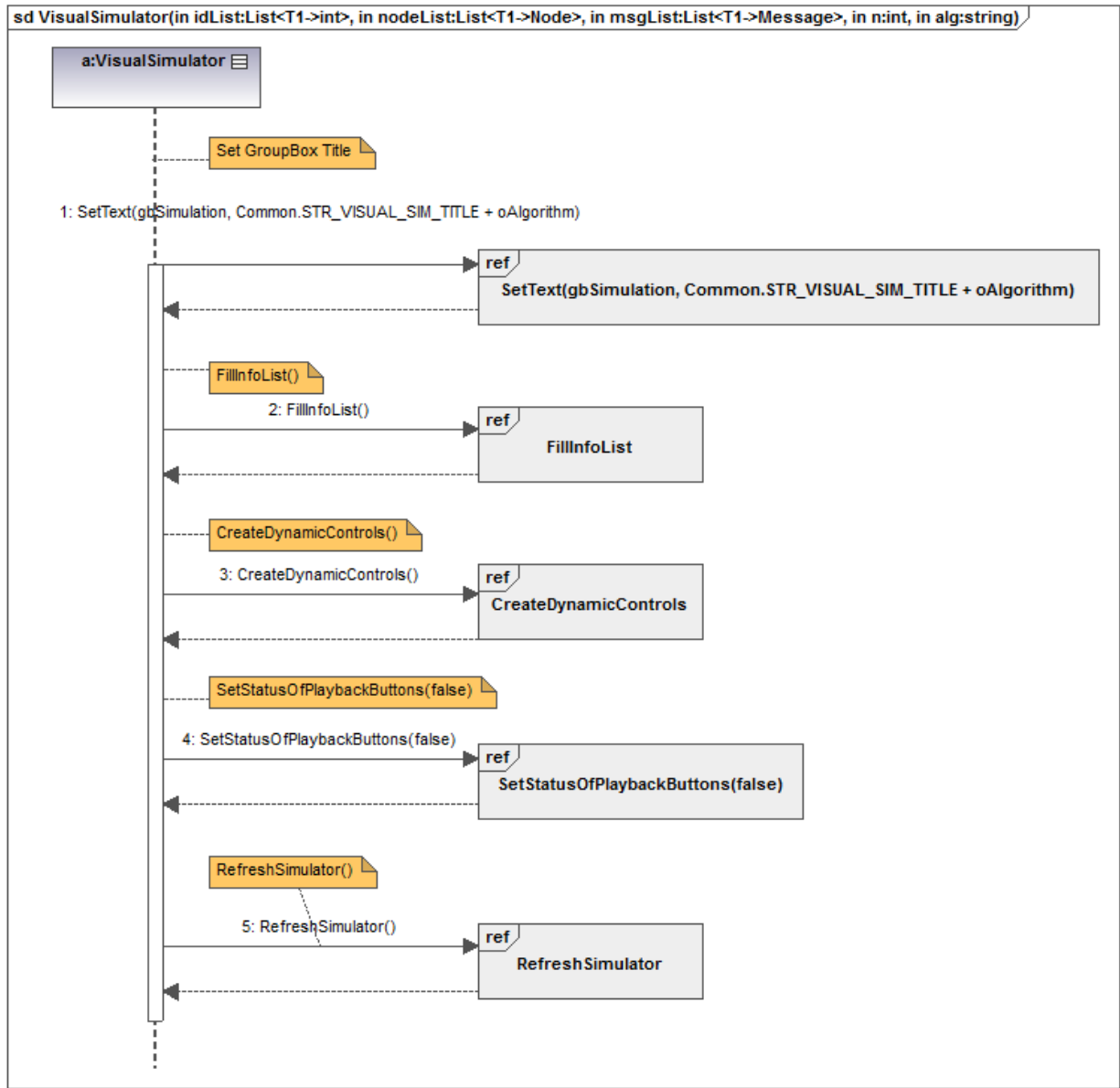
Figure 13: Sequence diagram of simulation launch process.



4.4.2.2 Initializing Visual Simulation

The following sequence diagram illustrates the initiating sequence of the Visual Simulator.

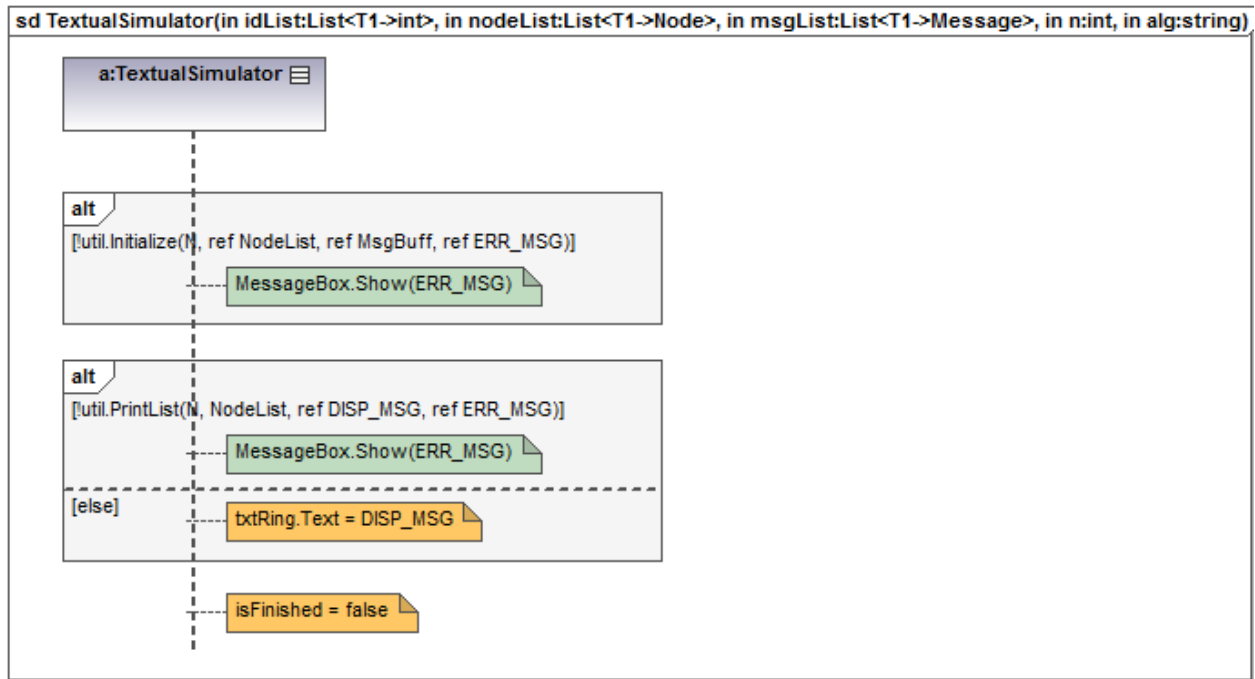
Figure 14: Sequence diagram of the initialize process of Visual Simulator.



4.4.2.3 Initializing Textual Simulation

The following sequence diagram illustrates the initiating sequence of the Textual Simulator.

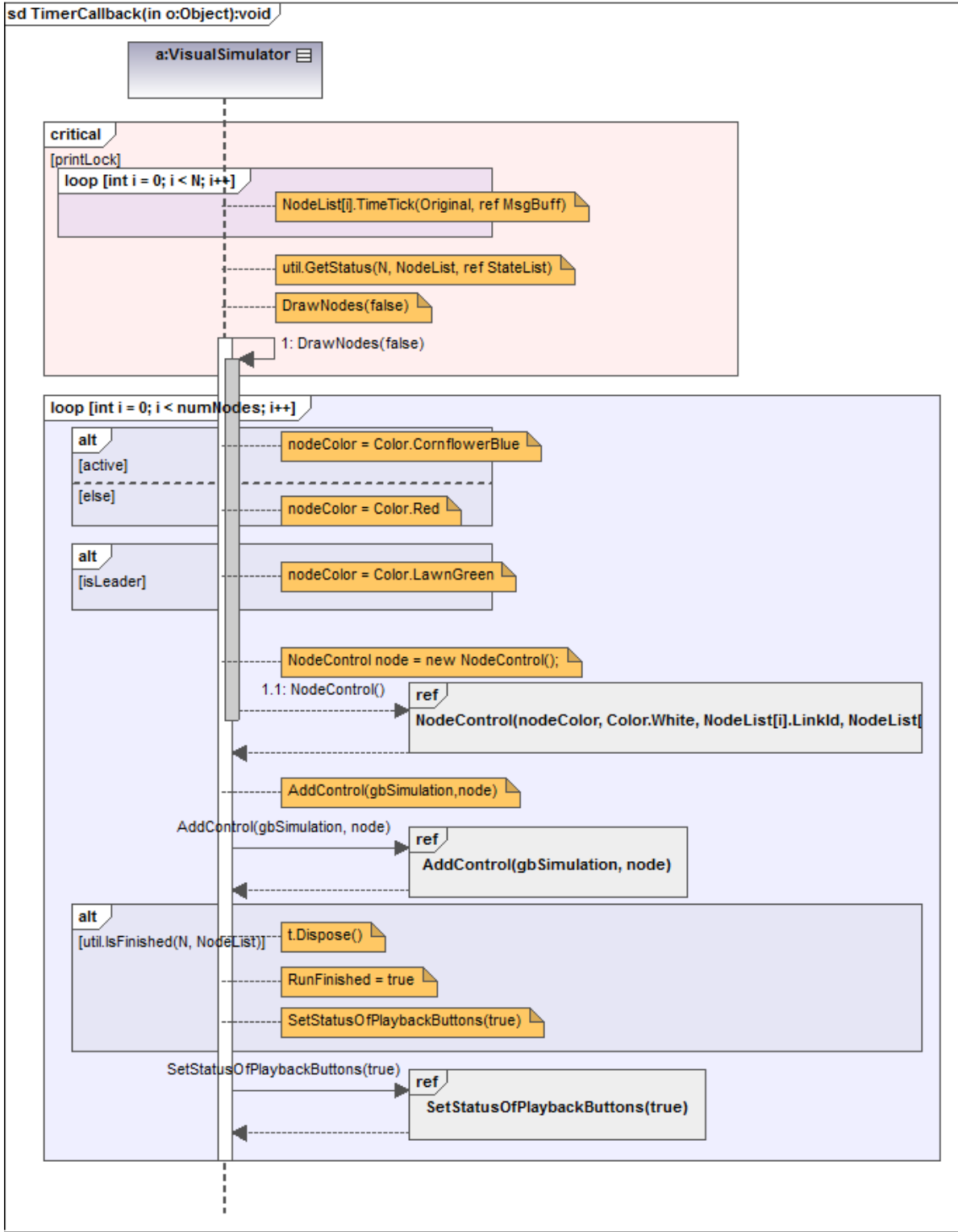
Figure 15: Sequence diagram of the initialize process of Textual Simulator.



4.4.2.4 TimerCallback

The following diagram illustrates the `TimerCallback()` method sequence. It is the method executed at each time-step of the algorithm.

Figure 16: Sequence diagram of TimerCallback process.



Chapter 5

Simulator Application – Implementation

In this chapter we describe in detail the implementation of Distributed Algorithm Simulator. We present documentation of all the namespaces, classes, methods, and properties used in implementing the system. Then, we focus on core components of the system and give a detailed description using illustrations and code samples.

5.1 SOURCE CODE DOCUMENTATION

In this section, we present a list of namespaces, classes, methods, and fields – along with descriptions – used in the Distributed Algorithm Simulator.

5.1.1 DistributedAlgorithmSimulator Namespace

Description: This namespace includes classes used for development of the application.

Table 4: Classes of DistributedAlgorithmSimulator namespace.











	Class	Description
	Common	Contains common fields.
	DistributedAlgorithmSimulator	Main Windows of the Distributed Algorithm Simulator application.
	ExtensionMethods	Extension methods used for various tasks.
	FileOperations	Static class offering file read/write operations tailored to Distributed Algorithm Simulator.
	History	Keeps a record of state of all the variables of the algorithm. One History object refers to one step in the execution of the algorithm.
	NodeControl	A control that represents a node in the Visual Simulator. Consists of a circular shaped graphic that represents a node and text on it that represents node IDs.
	NodeInfoControl	A control that represents variables to be displayed alongside each node control.
	TextualSimulator	Textual Simulator window.
	VisualSimulator	Visual Simulation window.

Table 5: Enumerations of DistributedAlgorithmSimulator namespace.

	Enumeration	Description
	Algorithm	Algorithm Type

5.1.1.1 Algorithm Enumeration

Description: Denotes algorithm type.

Table 6: Algorithm enumeration syntax.

C# Syntax
<code>public enum Algorithm</code>

Table 7: Members of Algorithm enumeration.

Member name	Value	Description
LCR	0	LCR Algorithm
UniqueK	1	UniqueK Algorithm





















5.1.1.2 Common Class

Description: Contains common fields.

Table 8: Common class syntax.

C# Syntax
<code>public static class Common</code>

Table 9: Fields of Common class.

	Name	Description
 	CAPTION_ERROR	Caption: ERROR! Used as general error message MessageBox caption.
 	CAPTION_FILE_ERROR	Caption: FILE ERROR. Used as file error message MessabeBox caption.
 	CAPTION_FILE_WRITE	Caption: FILE WRITE. Used as file write error message MessageBox caption.
 	CAPTION_INVALID_ID	Caption: INVALID ID ERROR. Used as caption for invalid ID error messages MessageBox.
 	CAPTION_TEXT_SIM	Caption: TEXTUAL SIMULATION. Used as caption for prompt for textual simulation MessageBox.
 	ERR_MSG_SUCCESS	Success message.
 	STR_ALG	Sting: Algorithm
 	STR_ALG_LCR	String: LCR
 	STR_ALG_UNIQUE_K	String: UniqueK
 	STR_DEFAULT_SAVE_FILE	String: SimulationResults (Default output file name without extension)
 	STR_DIR_CCW	String: Counter Clockwise
 	STR_DIR_CW	String: Clockwise
 	STR_FINISHED	String: Finished!
 	STR_TEXT_SIM_TITLE	String: Textual Simulation
 	STR_VISUAL_SIM_TITLE	String: Visual Simulation

5.1.1.3 DistributedAlgorithmSimulator Class

Description: The main windows of the Distributed Algorithm Simulator application.

Table 10: DistributedAlgorithmSimulator class syntax.

C# Syntax
<code>public class DistributedAlgorithmSimulator : Form</code>

Table 11: DistributedAlgorithmSimulator class constructor.














	Name	Description
	DistributedAlgorithmSimulator	Initializes a new instance of the DistributedAlgorithmSimulator class.

Table 12: DistributedAlgorithmSimulator class methods.

	Name	Description
	btnExit_Click	Exits the application.
	btnOpenFile_Click	Launches a FileOpen dialog which lets the user select the input file.
	btnRead_Click	Reads the node IDs and creates a list of IDs. Depending on the user's selection, reads either from a user selected file or takes input from a text box.
	btnRun_Click	Executes the simulation depending on user preference of Visual or Textual simulation. If the number of nodes is greater than 18, prompts the user to launch a Textual Simulation.
	CreateNetwork	Creates a ring network which can pass messages in either clockwise or counter-clockwise directions Nodes are linked to their CW and CCW neighbors using a private id called LinkID (which isn't a part of the algorithm)
	Initializations	Performs initializations such as setting initial values of controls.
	InitializeComponent	Required method for Designer support - do not modify the contents of this method with the code editor.
	LaunchSimulation	Launches either a Visual Simulation or a Textual Simulation of an algorithm of user's choice.
	LoadAlgorithms	Loads available algorithms into "Algorithm Type" combo box.
	rbInputIDs_CheckedChanged	Specifies user preference of reading input from a textbox.
	rbReadFromFile_CheckedChanged	Specifies user preference of reading input from a file.
	SimulatorMain_Load	Form load event of the main window.




5.1.1.4 ExtensionMethods Class

Description: Extension methods used for various tasks.

Table 13: ExtensionMethods class syntax.

C# Syntax
<code>public static class ExtensionMethods</code>

Table 14: ExtensionMethods class methods.

	Name	Description
	DeepClone(T)	Makes a new copy of an object without keeping a reference.
	DrawCircle	Draws a circle using given parameters on a Graphics object.
	SetHighQuality	Sets high quality parameters to a Graphics object.



5.1.1.5 FileOperations Class

Description: Static class offering file read/write operations tailored to Distributed Algorithm Simulator.

Table 15: FileOperations class syntax.

C# Syntax
<code>public static class FileOperations</code>

Table 16: FileOperations class methods.

	Name	Description
	ReadCSVFile	Reads a CSV file and store values in a list of integers.
	SaveToFile	Opens a SaveFileDialog to allow user to save the file.

5.1.1.6 History Class

Description: Keeps a record of states of all the variables of the algorithm. One **History** object refers to one step in the execution of the algorithm.





Table 17: History class syntax.

C# Syntax
<code>public class History</code>

Table 18: History class constructor.

	Name	Description
	History	Initializes an instance of the History class.

Table 19: History class properties.

	Name	Description
	RoundNum	Round number in the execution.
	RoundSteps	Step number in the current round of execution.
	StateList	List of NodeState objects representing the current step.
	TotalSteps	Step number in the overall execution.

5.1.1.7 NodeControl Class

Description: A control that represents a node in the Visual Simulator. Consists of a circular shaped graphic that represents a node and text on it that represents node IDs.

Table 20: NodeControl class syntax.

C# Syntax
<code>public class NodeControl : Control</code>

Table 21: NodeControl class constructor.




	Name	Description
	NodeControl	Initializes an instance of a NodeControl class.

Table 22: NodeControl class methods.

	Name	Description
	SetNodeColor	Sets the node color.
	SetTextColor	Sets the node ID text color.


5.1.1.8 NodeInfoControl Class

Description: A control that represents variables to be displayed alongside each node control.

Table 23: NodeInfoControl class syntax.

C# Syntax
<code>public class NodeInfoControl : Control</code>

Table 24: NodeInfoControl class constructor.

	Name	Description
	NodeInfoControl	Initializes an instance of a NodeInfoControl class.

5.1.1.9 TextualSimulator Class

Description: The Textual Simulator window.

Table 25: TextualSimulator class syntax.

C# Syntax
<code>public class TextualSimulator : Form</code>

Table 26: TextualSimulator class constructor.











	Name	Description
	TextualSimulator	Initializes an instance of the TextualSimulator class.

Table 27: TextualSimulator class methods.

	Name	Description
	btnExit_Click	Exits the Textual Simulation window.
	btnResetTextSim_Click	Resets the execution.
	btnRunTextSim_Click	Runs the execution.
	btnSaveTextSim_Click	Saves execution results to a file.
	InitializeComponent	Required method for Designer support - do not modify the contents of this method with the code editor.
	numUpDownSpeed_ValueChanged	Change the execution speed based on NumericUpDown control value.
	Reset	Resets all variables to initial state.
	TimerCallback	Performs one step of the execution of the algorithm. Called from the Timer, at each time tick.
	ToggleRunPause	Toggles Run/Pause button.

5.1.1.10 VisualSimulator Class

Description: The Visual Simulation window.

Table 28: VisualSimulator class syntax.

C# Syntax
<code>public class VisualSimulator : Form</code>

Table 29: VisualSimulator class constructor.


	Name	Description
	VisualSimulator	Initializes an instance of the VisualSimulator class.

Table 30: VisualSimulator class methods.

	Name	Description
	btnExit_Click	Exits the Visual Simulator window.
	btnFirst_Clicked	Jumps to the first step of the execution.
	btnLast_Clicked	Jumps to the last step of the execution.
	btnNext_Clicked	Advances one step forward through the history.
	btnPlayPause_Clicked	Play/Pause button click. Runs/Stops the execution.
	btnPrev_Clicked	Advances one step backward through the history.
	btnReset_Clicked	Resets all variables and counters.
	btnSaveToFile_Clicked	Saves the result of the execution to a file.
	CalculateRingProperties	Calculates the XY-coordinates and the radius of the ring.
	chkBox_CheckedChanged	Update the list of variables to be displayed based on check status of check boxes.
	CreateDynamicControls	Dynamically creates controls.
	DrawNodes	Draws nodes in the ring.
	FillInfoList	Gets the variables associated with the algorithm.
	gbSimulation_Paint	Draws the circle representing the ring.
	InitializeComponent	Required method for Designer support - do not modify the contents of this method with the code editor.
	numUpDownSpeed_ValueChanged	Updates the execution speed depending on the user selected value.
	RefreshSimulator	Resets all variables to initial state.
	SaveData	Saves the result of the execution to a file.
	SetCheckStatus	Sets the checked status of check boxes.
	SetCouners	Sets the values of TotalSteps, RoundNumber, and Steps in the Round counters.
	SetStatusOfPlaybackButtons	Sets the enabled statues of playback buttons.
	TimerCallback	Performs one step of the execution of the algorithm. Called from the Timer, at each time tick.
	UpdateInfoList	Updates the list of variables to be displayed depending on check status of check boxes.

5.1.2 WrapperClasses Namespace

Description: Wrapper classes that facilitate a template API for distributed algorithms implemented in the DistributedAlgorithms namespace.

Table 31: Classes of WrapperClasses namespace.






	Class	Description
	Message	MESSAGE super class. Used in the SimulatorMain. Each algorithm will derive from this to define its own message.
	Node	Node super class. Used in the SimulatorMain. Each algorithm's libraries will derive from this to implement the node.
	NodeState	Represents a state of variables of a node during the execution of the algorithm.
	Utility	Common Utility class. Classes of different algorithms must override these and perform appropriate changes.

Table 32: Structures of WrapperClasses namespace.

	Structure	Description
	NodeState.Item	Represents a variable of a node.

5.1.2.1 Message Class

Description: Message super class. Used in the SimulatorMain. Each algorithm will derive from this to define its own message.

Table 33: Message class syntax.

C# Syntax
<pre>[SerializableAttribute] public abstract class Message</pre>

5.1.2.2 Node Class

Description: Node super class. Used in the SimulatorMain. Each algorithm's libraries will derive from this to implement the node.

Table 34: Node class syntax.

C# Syntax
<pre>[SerializableAttribute] public abstract class Node</pre>

Table 35: Node class properties.








	Name	Description
	CcwNeighbor	LinkID of the counter-clockwise neighbor.
	CwNeighbor	LinkID of the clockwise neighbor.
	LinkId	The ID used to create the network. Different from node IDs.
	RcvdMsg	Received message at each step.
	SentMsg	Sent message at the end of each step.
	UID	Node IDs of nodes.

Table 36: Node class methods.

	Name	Description
	TimeTick	Actions performed at each time step. This is the coded Actions Table.

5.1.2.3 NodeState Class

Description: Represents a state of variables of a node during the execution of the algorithm.

Table 37: NodeState class syntax.

C# Syntax
[SerializableAttribute] public class NodeState

Table 38: NodeState class constructor.


	Name	Description
	NodeState	Initializes a new instance of NodeState class.

Table 39: NodeState class properties.






	Name	Description
	CcwId	LinkID of the counter-clockwise neighbor.
	CwId	LinkID of the clockwise neighbor.
	ItemList	List of variables.
	LinkId	The ID used to create the network. Different from node IDs.

Table 40: NodeState class methods.

	Name	Description
	AddItem	Adds an item to the Item List.



5.1.2.4 NodeState.Item Structure

Description: Represents a variable of a node.

Table 41: NoteState.Item structure syntax.

C# Syntax
<code>public struct Item</code>

Table 42: NodeState.Item structure properties.

	Name	Description
	ItemName	Variable name.
	ItemValue	Variable value. Converted to object type.

5.1.2.5 Utility Class

Description: Common Utility class. Classes of different algorithms must override these and perform appropriate changes.

Table 43: Utility class syntax.

C# Syntax
<code>public abstract class Utility</code>

Table 44: Utility class methods.








	Name	Description
	ErrorChecks	Checks for errors.
	GetStatus	Returns (as a reference) the state of all the nodes as a NodeState list.
	GetVariables	Gets a list of variables that the algorithm uses.
	Initialize	Performs required initializations.
	IsFinished	Checks whether the algorithm execution is finished.
	PrintList	Prints the ring orientation in clockwise order.
	PrintStep	Returns (as a reference) the state of all the nodes as a single string.







Table 45: Utility class fields.

	Name	Description
	ERR_MSG_SUCCESS	Successful operation.

5.1.3 DistributedAlgorithms Namespace

Description: This namespace includes the coded distributed leader election algorithms in ring networks that are used in this simulator. Namely, LCR algorithm and UNIQUE_ k algorithm.

Table 46: Classes of DistributedAlgorithms namespace.

	Class	Description
	LCRMsg	Message prototype for messages used in LCR algorithm.
	LCRNode	Contains LCR Algorithm actions.
	LCRUtility	A utility class that performs actions such as initializations and error checks. It acts as an interface to the main application which obtains the status of the algorithm at each step.
	UniqueKMsg	Message prototype for messages used in UNIQUE_ k algorithm.
	UniqueKNode	Contains UniqueK Algorithm actions.
	UniqueKUtility	A utility class that performs actions such as initializations and error checks. It acts as an interface to the main application which obtains the status of the algorithm at each step.

5.1.3.1 LCRMsg Class

Description: Message prototype for messages used in LCR algorithm.



Table 47: LCRMsg class syntax.

C# Syntax
<pre>[SerializableAttribute] public class LCRMsg : Message</pre>

Table 48: LCRMsg class constructor.

	Name	Description
	LCRMsg	Initializes message values.

Table 49: LCRMsg class properties.

	Name	Description
	SP	Indicates whether a message is a special message or not. True: special message, False: regular message.
	UID	Unique IDs of nodes. Non-negative values.

5.1.3.2 LCRNode Class

Description: Contains LCR Algorithm actions.

Table 50: LCRNode class syntax.

C# Syntax	
[SerializableAttribute]	<code>public class LCRNode : Node</code>

Table 51: LCRNode class constructor.


	Name	Description
	LCRNode	Initializes a new instance of the LCRNode class.

Table 52: LCRNode class properties.















	Name	Description
	Active	Represents whether the node is active or not.
	CcwNeighbor	LinkID of the counter-clockwise neighbor. (Inherited from Node.)
	CwNeighbor	LinkID of the clockwise neighbor. (Inherited from Node.)
	Init	Represents whether algorithm is initialized or not.
	IsLeader	True: if the node is the leader. False: otherwise.
	Leader	Elected leader's ID
	LeaderElected	True: the nodes know that a leader has been elected by the algorithm. False: otherwise.
	LinkId	The ID used to create the network. Different from node IDs. (Inherited from Node.)
 	NOMESSAGE	An LCR algorithm message containing NOMESSAGE as its X value represents a non-message. i.e., the same as no message being sent through the channel.
	RcvdMsg	Received message at each step. (Inherited from Node.)
	SentMsg	Sent message at the end of each step. (Inherited from Node.)
	UID	Node IDs of nodes. (Inherited from Node.)

Table 53: LCRNode class methods.

	Name	Description
	TimeTick	Actions performed at each time step. This is the coded Actions Table. (Overrides Node.TimeTick(List(Message), List(Message)).)

5.1.3.3 LCRUtility Class

Description: A utility class that performs actions such as initializations and error checks. It acts as an interface to the main application which obtains the status of the algorithm at each step.

Table 54: LCRUtility class syntax.

C# Syntax
<code>public class LCRUtility : Utility</code>

Table 55: LCRUtility class constructor.











	Name	Description
	LCRUtility	Initializes a new instance of the LCRUtility class

Table 56: LCRUtility class methods.

	Name	Description
	ErrorChecks	Checks for errors. (Overrides Utility.ErrorChecks(Int32, List(Int32), String).)
 	FindK	Finds the maximum number of repeating IDs in a list of integers
	GetStatus	Returns (as a reference) the state of all the nodes as a NodeState list. (Overrides Utility.GetStatus(Int32, List(Node), List(NodeState)).)
	GetVariables	Returns a list of variables used in the algorithm. (Overrides Utility.GetVariables().)
	Initialize	Performs required initializations (Overrides Utility.Initialize(Int32, List(Node), List(Message), String).)
	IsFinished	Checks whether the algorithm execution is finished. (Overrides Utility.IsFinished(Int32, List(Node)).)
	PrintList	Prints the ring orientation in clockwise order. (Overrides Utility.PrintList(Int32, List(Node), String, String).)
	PrintStep	Returns (as a reference) the state of all the nodes as a single string. (Overrides Utility.PrintStep(Int32, Int32, List(Node), String, String).)

5.1.3.4 UniqueKMsg Class

Description: Message prototype for messages used in UniqueK algorithm.

Table 57: UniqueKMsg class syntax.

C# Syntax
[SerializableAttribute] public class UniqueKMsg : Message

Table 58: UniqueKMsg class constructor.




	Name	Description
	UniqueKMsg	Initializes message values.

Table 59: UniqueKMsg class properties.

	Name	Description
	C	Counter.
	X	IDs of the originating node.

5.1.3.5 UniqueKNode Class

Description: Contains UniqueK Algorithm actions.

Table 60: UniqueKNode class syntax.

C# Syntax
<code>[SerializableAttribute] public class UniqueKNode : Node</code>

Table 61: UniqueKNode class constructor.


	Name	Description
	UniqueKNode	Initializes a new instance of the UniqueKNode class.

Table 62: UniqueKNode class properties.


















	Name	Description
	Active	Represents whether the node is active or not.
	CcwNeighbor	LinkID of the counter-clockwise neighbor. (Inherited from Node.)
	Count	P.count variable.
	CwNeighbor	LinkID of the clockwise neighbor. (Inherited from Node.)
	Init	Represents whether algorithm is initialized or not.
	IsLeader	True: if the node is the leader. False: otherwise.
	K	K, the maximum number of times a node ID is repeated in the ring.
	Leader	Elected leader's ID.
	LeaderElected	True: the nodes know that a leader has been elected by the algorithm. False: otherwise.
	LinkId	The ID used to create the network. Different from node IDs. (Inherited from Node.)
 	NOCOUNT	An undefined count state.
 	NOMESSAGE	An LCR algorithm message containing NOMESSAGE as its X value represents a non-message. i.e., the same as no message being sent through the channel.
	RcvdMsg	Received message at each step. (Inherited from Node.)
	SentMsg	Sent message at the end of each step. (Inherited from Node.)
	UID	Node IDs of nodes. (Inherited from Node.)
 	UNDEFINED	Denotes an undefined state.

Table 63: UniqueKNode class methods.

	Name	Description
	TimeTick	Actions performed at each time step. This is the coded Actions Table. (Overrides Node.TimeTick(List(Message), List(Message)).)

5.1.3.6 UniqueKUtility Class

Description: A utility class that performs actions such as initializations and error checks. It acts as an interface to the main application which obtains the status of the algorithm at each step.

Table 64: UniqueKUtility class syntax.

C# Syntax
<code>public class UniqueKUtility : Utility</code>

Table 65: UniqueKUtility class constructor.













	Name	Description
	UniqueKUtility	Initializes a new instance of the UniqueKUtility class.

Table 66: UniqueKUtility class methods.

	Name	Description
	ErrorChecks	Checks for errors. (Overrides Utility.ErrorChecks(Int32, List(Int32), String).)
 	FindK(List(Int32))	Finds the maximum number of repeating IDs in a list of integers.
 	FindK(List(Node))	Finds the maximum number of repeating IDs in a list of nodes.
	GetStatus	Returns (as a reference) the state of all the nodes as a NodeState list. (Overrides Utility.GetStatus(Int32, List(Node), List(NodeState)).)
	GetVariables	Returns a list of variables used in the algorithm. (Overrides Utility.GetVariables().)
	Initialize	Performs required initializations (Overrides Utility.Initialize(Int32, List(Node), List(Message), String).)
	IsFinished	Checks whether the algorithm execution is finished. (Overrides Utility.IsFinished(Int32, List(Node)).)
	PrintList	Prints the ring orientation in clockwise order. (Overrides Utility.PrintList(Int32, List(Node), String, String).)
	PrintStep	Returns (as a reference) the state of all the nodes as a single string. (Overrides Utility.PrintStep(Int32, Int32, List(Node), String, String).)

5.2 EXPLANATION OF CRUCIAL COMPONENTS

In this section, we describe in detail the implementation of some of the crucial core components of the Distributed Algorithm Simulator.

5.2.1 Source Code Organization

Inside the Visual Studio IDE, the source code is arranged under the 'Source' folder according to the following structure.

Figure 17: Source code organization in Visual Studio IDE.



The relationship between namespaces and folders are as follows.

Table 67: Organization of folders according to namespaces.

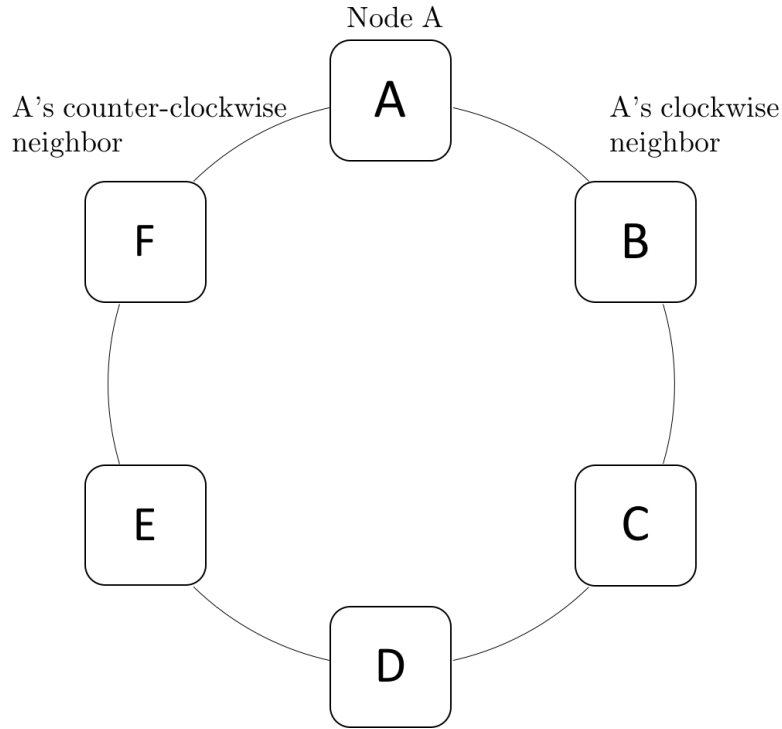
Namespace	Folder(s)
DistributedAlgorithms	Algorithm_LCR
	Algorithm_UniqueK
DistributedAlgorithmSimulator	ApplicationMain
	Common
	TextualSimulator
	VisualSimulator
WrapperClasses	WrapperClasses

5.2.2 Creating the Network

The Distributed Algorithm Simulator simulates the execution of algorithms in a ring network. As such we need a way to denote nodes arranged in a ring orientation, and their relationship to other nodes.

In a ring network, any given node has only two neighbors which we call the clockwise neighbor and the counter-clockwise neighbor, as shown in the following illustration.

Figure 18: Organization of nodes in a ring network.

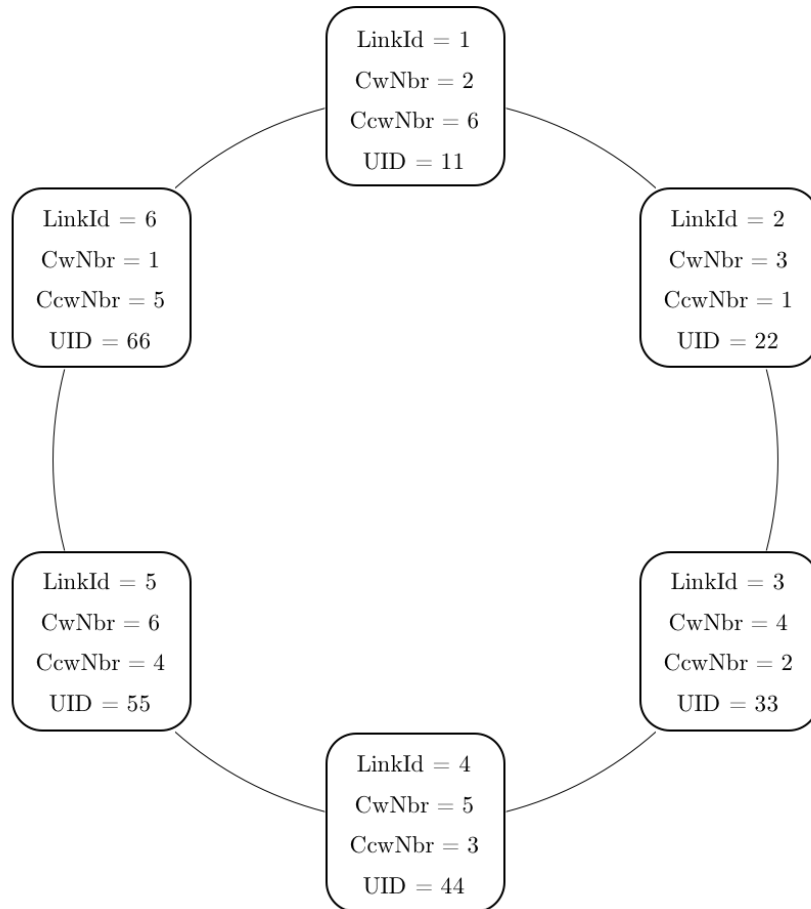


This is implemented by using a list (.NET class: `System.Collections.Generic.List`) of `Node` class objects which we have defined in the `WrapperClasses` namespace. In the `Node` class, we have defined 3 integer type variables for the purpose of implementing the network as follows.

- `LinkId` – A positive non-zero integer assigned at the network creation time, and is used to identify each node's position relative to other nodes in the network. This is different from node IDs used in the algorithms.
- `CwNeighbor` – `LinkID` of the current `Node` object's clockwise neighbor.
- `CcwNeighbor` – `LinkID` of the current `Node` object's counter-clockwise neighbor.

For example, let us consider a network with 6 nodes. Let us also assume the arbitrary node IDs of the nodes are 11, 22, 33, 44, 55, and 66. We assign LinkID = 1 and ID = 11 to the first node, LinkID = 2 and ID = 22 to the second node, so on and so forth. Following figure illustrates the state of variables of each node after the network is created.

Figure 19: Relationships between neighboring nodes in a ring network.



In the `CreateNetwork()` method, we create a list of `Node` objects using a for-loop that runs from 1 to n , n being the size of the network. Each `Node` object is assigned a `LinkID` such that the first node gets the `LinkID = 1`, the second node gets the `LinkID = 2`, so on and so forth. At the same time, each `Node` object's `CwNeighbor` and `CcwNeighbor` values are calculated and assigned. For first and last nodes in the list, when calculating `CcwNeighbor` and `CwNeighbor` respectively, we wrap around. At the same time, we also assign actual node IDs (either read from a file or taken from user input) to each node, in the order they are listed in the file or the user input.

Below is the `CreateNetwork()` method with error handling sections removed to better focus on the core of the method.

Figure 20: CreateNetwork() method.

```
private bool CreateNetwork()
{
    NodeList = new List<Node>(n);

    for (int i = 1; i <= n; i++) {
        // Set values from 1... n to LinkId
        int linkId = i;
        int cw = linkId + 1;
        // Wrapping around
        if (cw > n)
            cw = 1;
        int ccw = linkId - 1;
        // Wrapping around
        if (ccw < 1)
            ccw = n;

        Node newNode = null;
        switch (algorithm) {
            case Common.STR_ALG_UNIQUE_K:
                newNode = new UniqueKNode(linkId, cw, ccw);
                break;
            case Common.STR_ALG_LCR:
                newNode = new LCRNode(linkId, cw, ccw);
                break;
        }

        // Each node is assigned the actual UIDs
        newNode.UID = IdList[i - 1];
        NodeList.Add(newNode);
    }
}
```

Once the CreateNetwork() function finishes execution, we have a list of Node objects that represents the network. Each node is aware of its own position in the network and that of the two

neighbors on either side of it. We use this knowledge when the algorithms are being executed as explained in the next section.

5.2.3 Executing Algorithm Steps

We use a timer (.NET class: `System.Threading.Timer`) object and a callback method to execute the algorithms step-by-step. The callback method is called at every time-tick of the timer and represents the execution of one time-step in the algorithm. Let us take a look at this callback method, once again error handling sections removed.

Figure 21: TimerCallback() method.

```
private void TimerCallback(Object o)
{
    // Critical section.
    // Calls the algorithm's step execution method.
    // 'Original' contains received messages for this step.
    // 'MsgBuff' is updated with messages
    // to be sent after this step.
    lock (printLock) {
        for (int i = 0; i < N; i++) {
            NodeList[i].TimeTick(Original, ref MsgBuff);
        }

        // Gets the status of the variables after above step.
        util.GetStatus(N, NodeList, ref StateList);

        // Draw nodes accordingly.
        DrawNodes(false);
    }

    // Check whether the execution is finished.
    // If true, finish execution.
    if (util.IsFinished(N, NodeList)) {
        t.Dispose();
    }
}
```

The Node class contains a TimeTick() method which executes the actions of the algorithms which are performed at each step. In the TimerCallback() method, we traverse through each Node object in the list of Node objects created in the CreateNetwork() method and execute the TimeTick() method. This is equivalent to each node executing one step in a real distributed system.

The Utility class contains the method `IsFinished()` which checks if all the nodes in the network satisfy the conditions for termination. Therefore, if `IsFinished()` returns true at any given time during the execution, that implies the algorithm has finished its work. In which case the `TimerCallback()` method disposes the timer object and finishes the execution.

5.2.4 Message Communication

The distributed algorithms described in this document use message passing as the mode of communication. In a real distributed system this would be done by some sort of message passing protocol such as MPI. In this application, we use a message buffer to simulate the node.

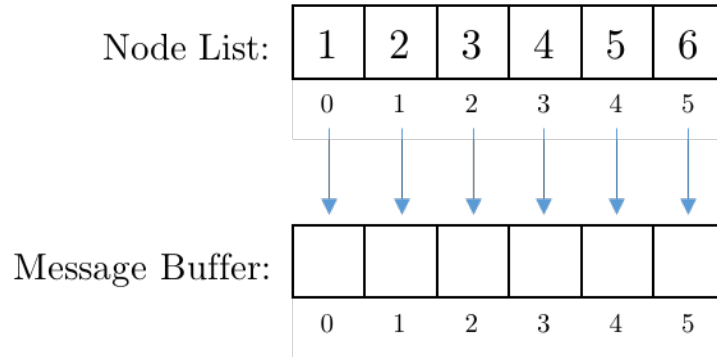
To describe the process, let us consider a ring network consisting of 6 nodes, and let us assume that the nodes are represented by a list of Node class objects as described in the ‘Creating the Network’ section above. The following figure illustrate the configuration, with labels inside the list items indicating each node’s LinkId, and the labels below the list items indicating each node’s zero-based index in the list.

Figure 22: Representation of list of nodes and their LinkIDs.

Node List:	1	2	3	4	5	6
	0	1	2	3	4	5

To hold the messages, we use a message buffer which is a list of Message class objects. Each algorithm (such as the LCR algorithm or the `UNIQUE_k` algorithm) has its own message class which is a derived class of the Message class, and thus defines message types according to that particular algorithm's requirements. The crucial connection here is that in the list of Message class objects, each item corresponds to the like index item of the Node list. The Message Buffer can be thought of as set of mail boxes assigned to each node in the node list; receiving mail must be retrieved from your own mailbox, and sending mail must be put inside intended receiver's mailbox.

Figure 23: Relationship between node list and message buffer.

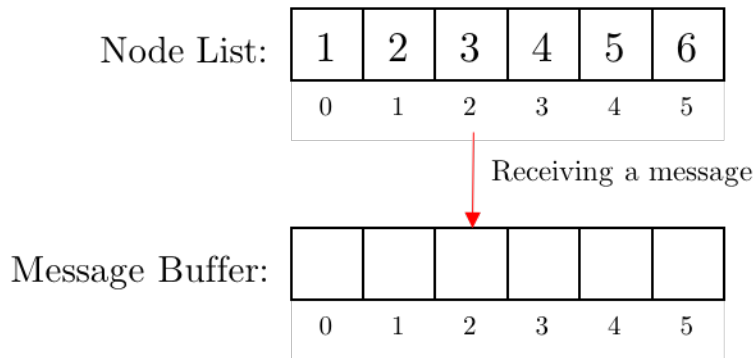


Consider the case where the node with `LinkId = 3` (`index = 2`) receiving and sending messages. As described in the 'Creating the Network' section above, its clockwise and counter-clockwise neighbors are `LinkId = 4` and `LinkId = 2` respectively, and thus the corresponding indices are 3 and 1. Now if we assume the algorithm passes messages in the clockwise direction, it would mean

that a node receives messages from the counter-clockwise neighbor, and it sends messages to the clockwise neighbor.

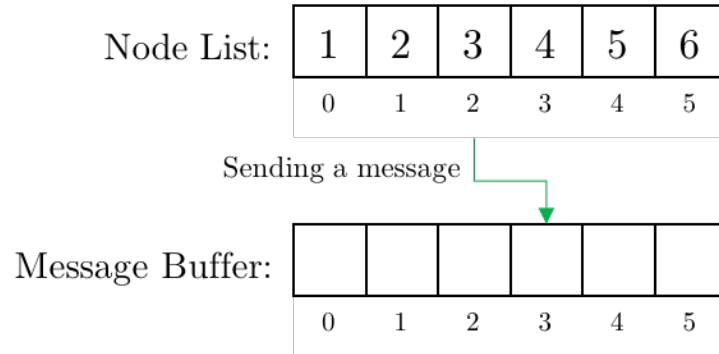
When $\text{LinkId} = 3$ node wants to receive a message, it grabs the message from the item corresponding to its own index in the message buffer. That is, it copies the message from index 2 of message buffer, as illustrated below. As such, regardless of whether the algorithm operates clockwise or counter-clockwise, receiving messages are always retrieved from the index corresponding to the receiver's own index.

Figure 24: Receiving a message from the counter-clockwise neighbor.



Conversely, when sending a message, the $\text{LinkId} = 3$ node copies the message it wants to send to the item corresponding to the index of its clockwise neighbor in the message buffer. In other words, $\text{LinkId} = 3$ node copies a message to the index 3 of message buffer. If the algorithm was operating in the counter-clockwise direction, conversely, sending message must be copied to the index 1 of the message buffer.

Figure 25: Sending a message to the clockwise neighbor.



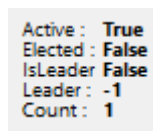
5.2.5 Visual Simulator – Drawing Nodes

The colored circles that represent nodes and the information displayed next to them are displayed using two class objects, namely, NodeControl and NodeInfoControl, which are derived classes of Microsoft .NET Control class.

Figure 26: An instance of a NodeControl control.



Figure 27: An instance of a NodeInfoControl control.



5.2.5.1 NodeControl Class

The NodeControl class which is derived from the .NET Control class contains two major components:

- A fill circle – Represents a node. Drawn using Graphics.FillEllipse() .NET method.
- A string of text – Represents a node ID. Drawn using Graphics.DrawString() .NET method.

The NodeControl class contains two methods, DrawNode() and DrawID(), which are called from the overridden OnPaint() method of the class, which draws the circle representing the node and the text representing the node ID, respectively [18].

Figure 28: Code sample of the NodeControl class.

```
public class NodeControl : Control
{
    /// <summary>
    /// OnPaint event of the object.
    /// </summary>
    /// <param name="e"></param>
    protected override void OnPaint(PaintEventArgs e) {
        G = e.Graphics;
        G.SetHighQuality();
        DrawNode();
        DrawID();
    }

    /// <summary>
    /// Draws a node control.
    /// </summary>
    protected void DrawNode() {
        G.FillEllipse(new SolidBrush(NodeColor),
            new Rectangle(0, 0,
                DefSize,
                DefSize));
    }

    /// <summary>
    /// Draws the node ID text.
    /// </summary>
    protected void DrawID() {
        string text = ID.ToString();
        G.DrawString(text, this.Font,
            new SolidBrush(TextColor),
            DefSize / 4,
            DefSize / 4);
    }
}
```

5.2.5.2 NodeInfoControl Class

The NodeControl is also derived from the .NET Control class, and it contains a number of label controls matching the variables of the algorithm that must be displayed. For instance, the LCR algorithm contains four variables that can be displayed, namely, Active, IsLeader, LeaderElected, and Leader.

We pass a list of strings which contains the names of the variables to be displayed, and a NodeState object which contains the values of those variables, to the constructor. The DrawLables() method iterates through the list and draws two labels for each variable that must be shown; one to display the name of the variable and the other to display the value.

Figure 29: Code Sample of the NodeInfoControl Class.

```
public class NodeInfoControl : Control
{
    protected override void OnPaint(PaintEventArgs e) {
        G = e.Graphics;
        G.SetHighQuality();
        DrawLabels();
    }

    protected void DrawLabels() {
        for (int i = 0; i < infoList.Count; i++) {
            // Label representing variable name.
            Label lblItemName = new Label();
            lblItemName.Text = infoList[i] + " : ";
            this.Controls.Add(lblItemName);

            // Label representing variable value.
            Label lblItemValue = new Label();
            string value = nodeState.ItemList.Find(
                x => x.ItemName.Equals(
                    infoList[i])).ItemValue.ToString();
            lblItemValue.Text = value;
            this.Controls.Add(lblItemValue);
        }
    }
}
```

Chapter 6

User Guide

This chapter is organized as a user guide to the end-user of the Distributed Algorithm Simulator.

We provide step-by-step guides for installation, basic overview of the system, and the usage of the Visual Simulator and the Textual Simulator.

6.1 DOWNLOAD

The Distributed Algorithm Simulator setup file can be downloaded from the following BitBucket repository. In addition, the entire repository can also be obtained at the same link.

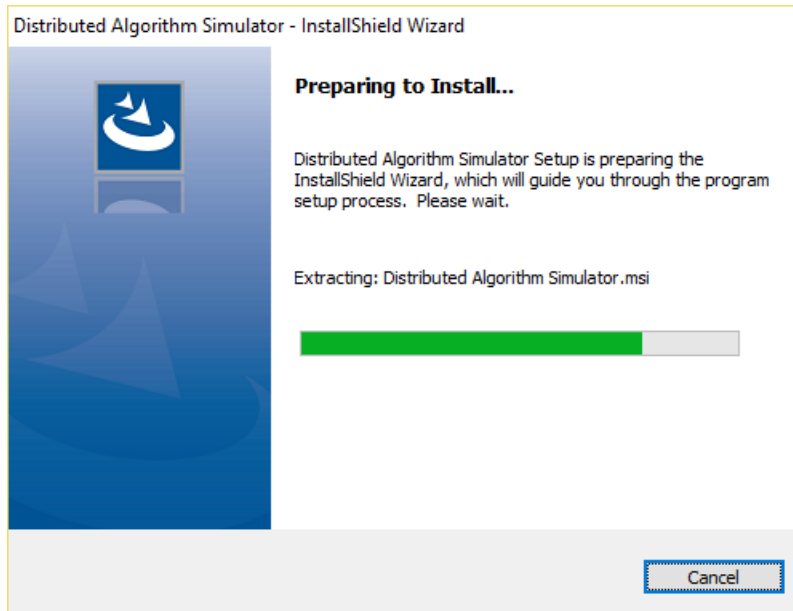
- <https://bitbucket.org/sachintha81/distributedalgorithmsimulator-public/downloads>

6.2 INSTALLATION

Following steps will guide you through the installation process.

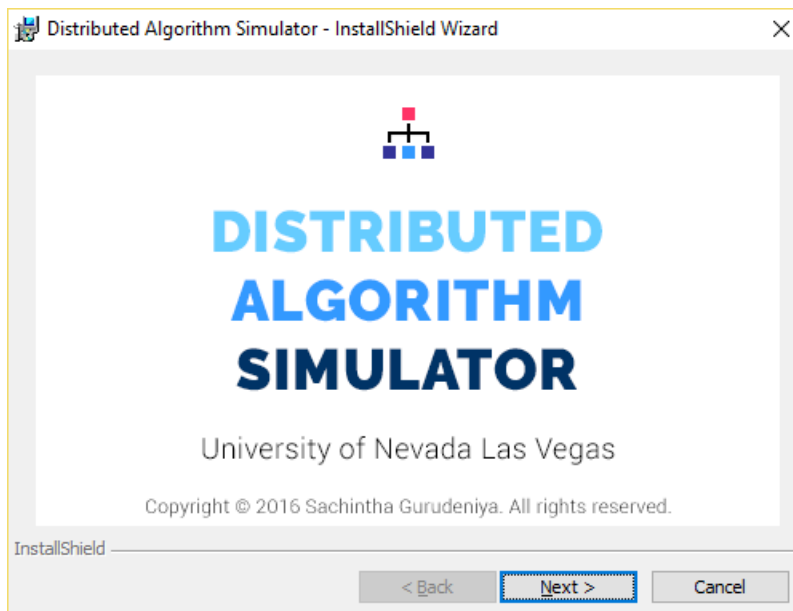
- Download the DistributedAlgorithmSimulatorSetup.zip file and extract it.
- Double-click the DistributedAlgorithmSimulatorSetup.exe file on the extracted folder.
- [Preparing to Install...] screen will be displayed. Wait for the next screen.

Figure 30: [Preparing to Install] screen.



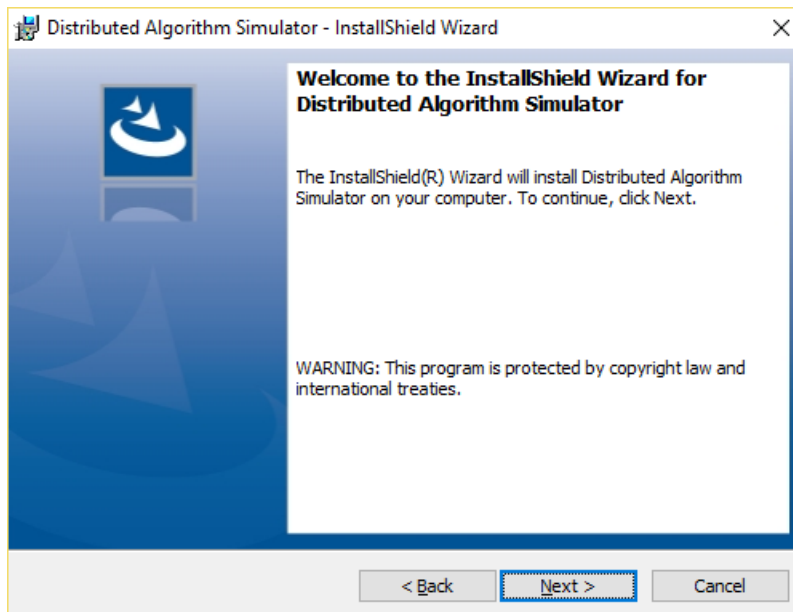
- Click [Next >] at the Distributed Algorithm Simulator splash screen.

Figure 31: Distributed Algorithm Simulator splash screen.



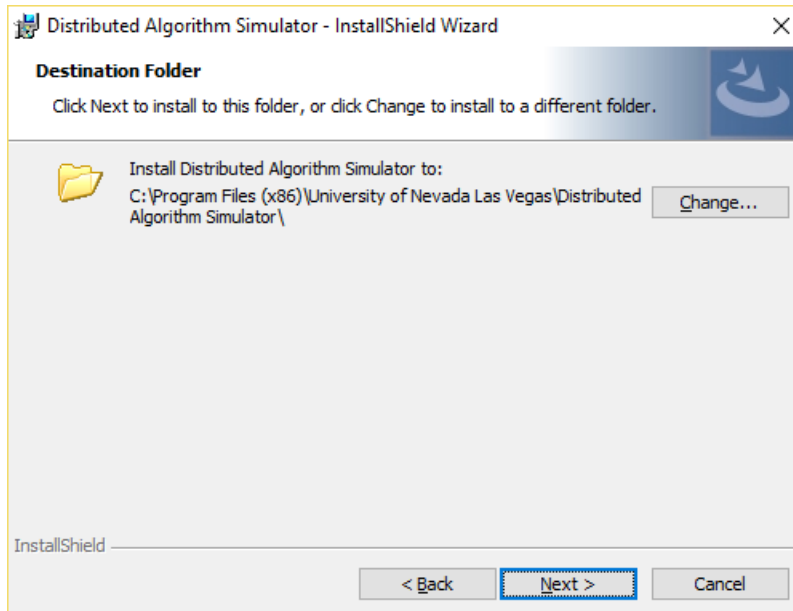
- Click [Next >] at the [Welcome] screen.

Figure 32: [Welcome] screen.



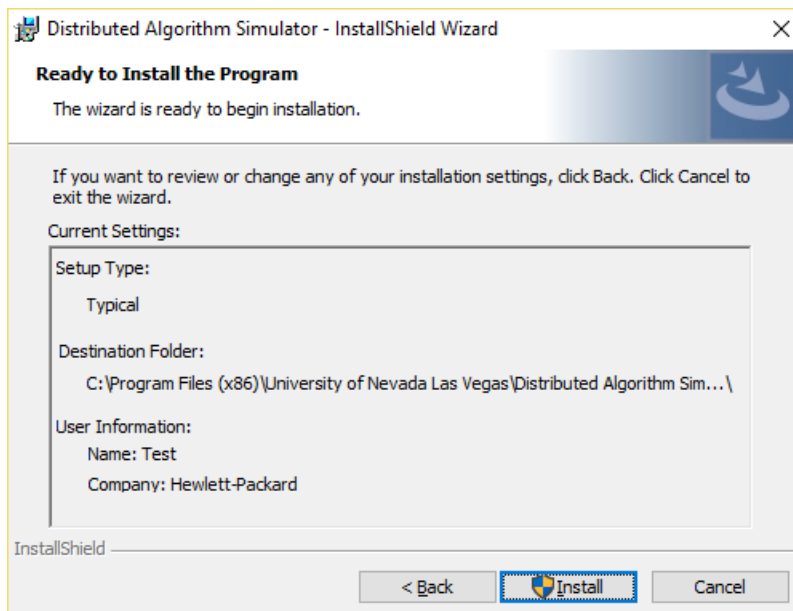
- The [Destination Folder] screen lets the user select the install directory. Either leave the default directory, or click [Change...] to select a different directory. Click [Next >]

Figure 33: [Destination Folder] screen.



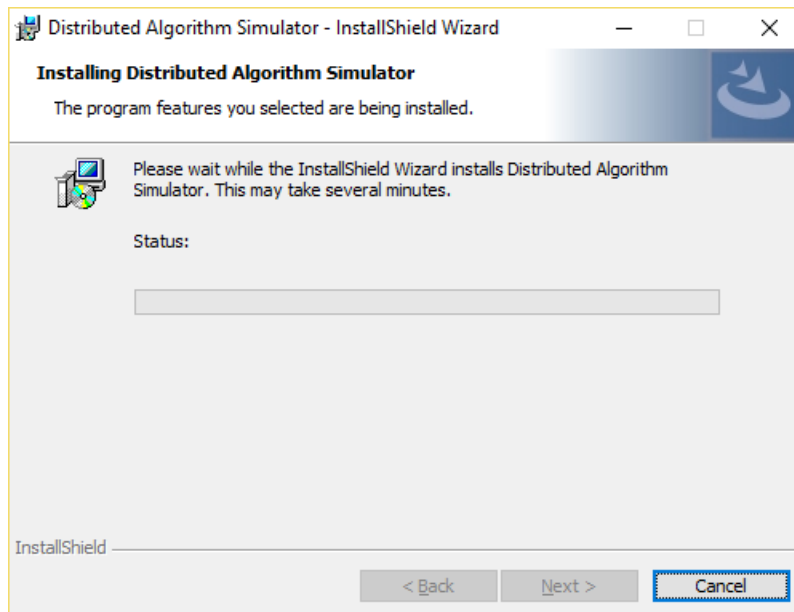
- Click [Install] at the [Ready to Install the Program] screen.

Figure 34: [Ready to Install the Program] screen.



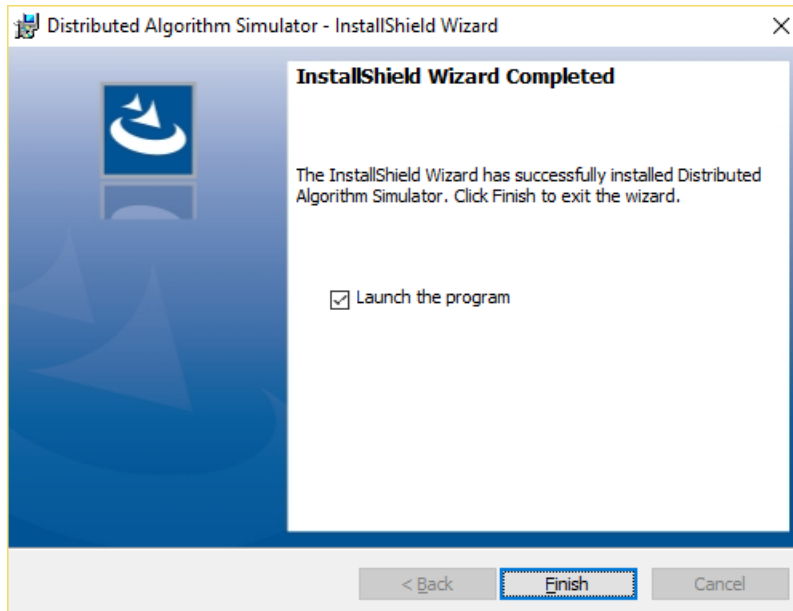
- Wait until the installation is complete.

Figure 35: [Installing Distributed Algorithm Simulator] screen.



- Click [Finish] at the [InstallShield Wizard Complete] screen. If [Launch the program] check box is checked, it will launch the installed Distributed Algorithm Simulator program.

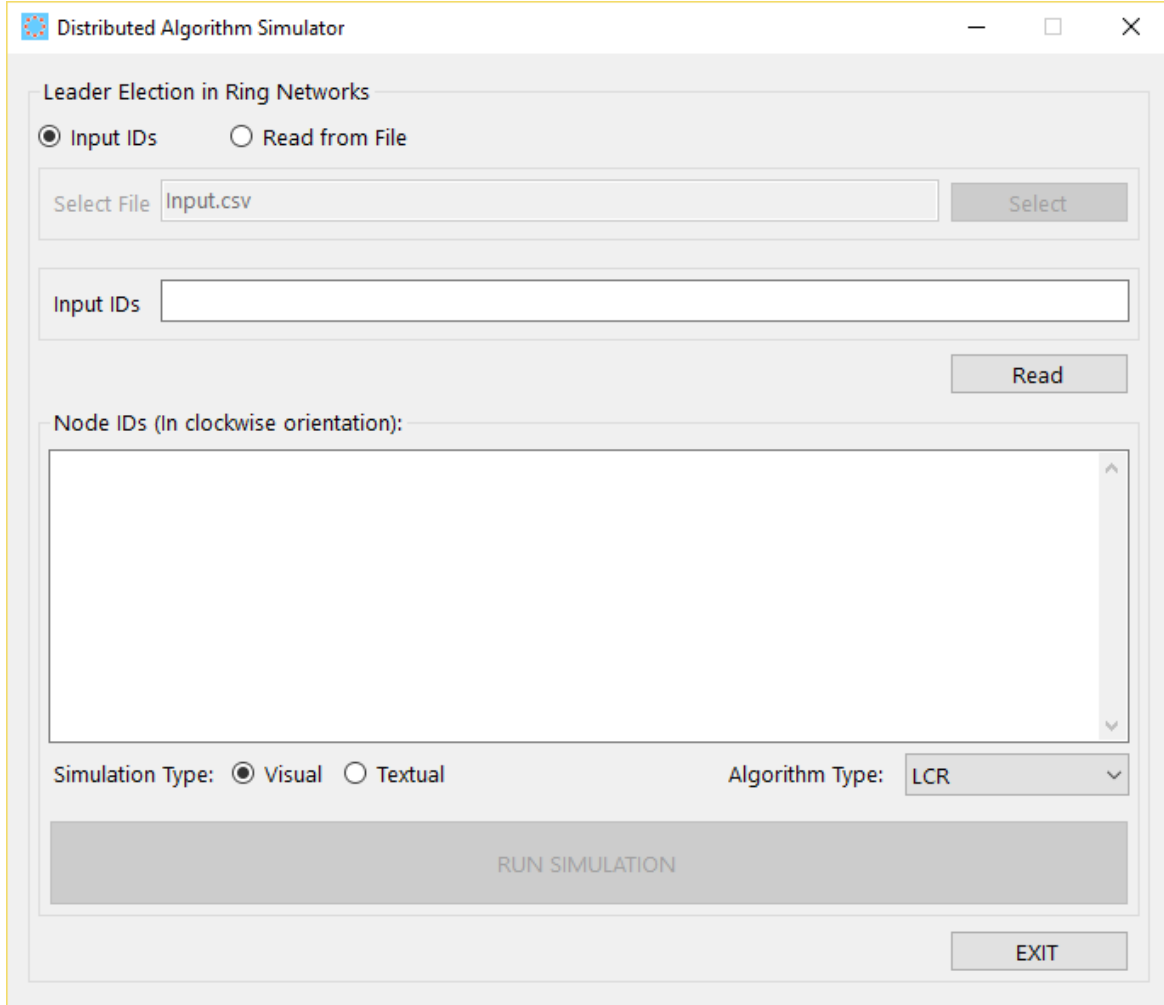
Figure 36: [InstallShield Wizard Complete] screen.



6.3 DISTRIBUTED ALGORITHM SIMULATOR MAIN WINDOW

Launching the application opens the following Distributed Algorithm Simulator main window.

Figure 37: Distributed Algorithm Simulator main window.



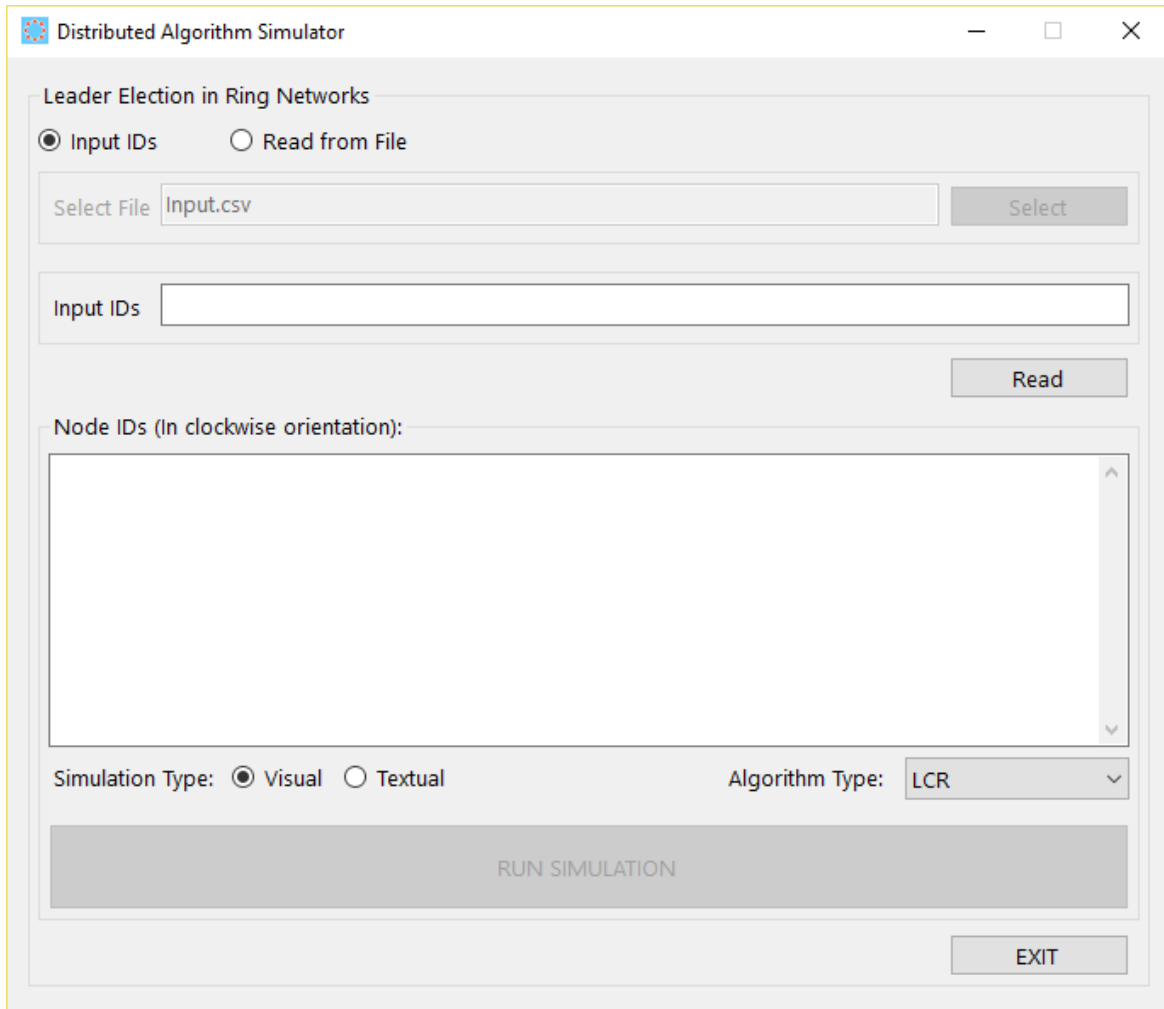
- [Input IDs] and [Read from File] radio buttons let user input the node IDs to the [Input IDs] textbox, or read them from a file.
- By default, [Input IDs] radio button is checked.
- [Node IDs] textbox displays the node IDs, in clockwise orientation, once read.

- [Visual] and [Textual] radio buttons let the user select the type of simulation to run: The Visual Simulation or the Textual Simulation.
- [Algorithm Type] drop down list lets the user select the algorithm to run.
- The [RUN SIMULATION] button is deactivated by default. It becomes activated once the node IDs are read.
- The [EXIT] button lets the user terminate the Distributed Algorithm Simulator.

6.3.1 Input IDs into a Textbox.

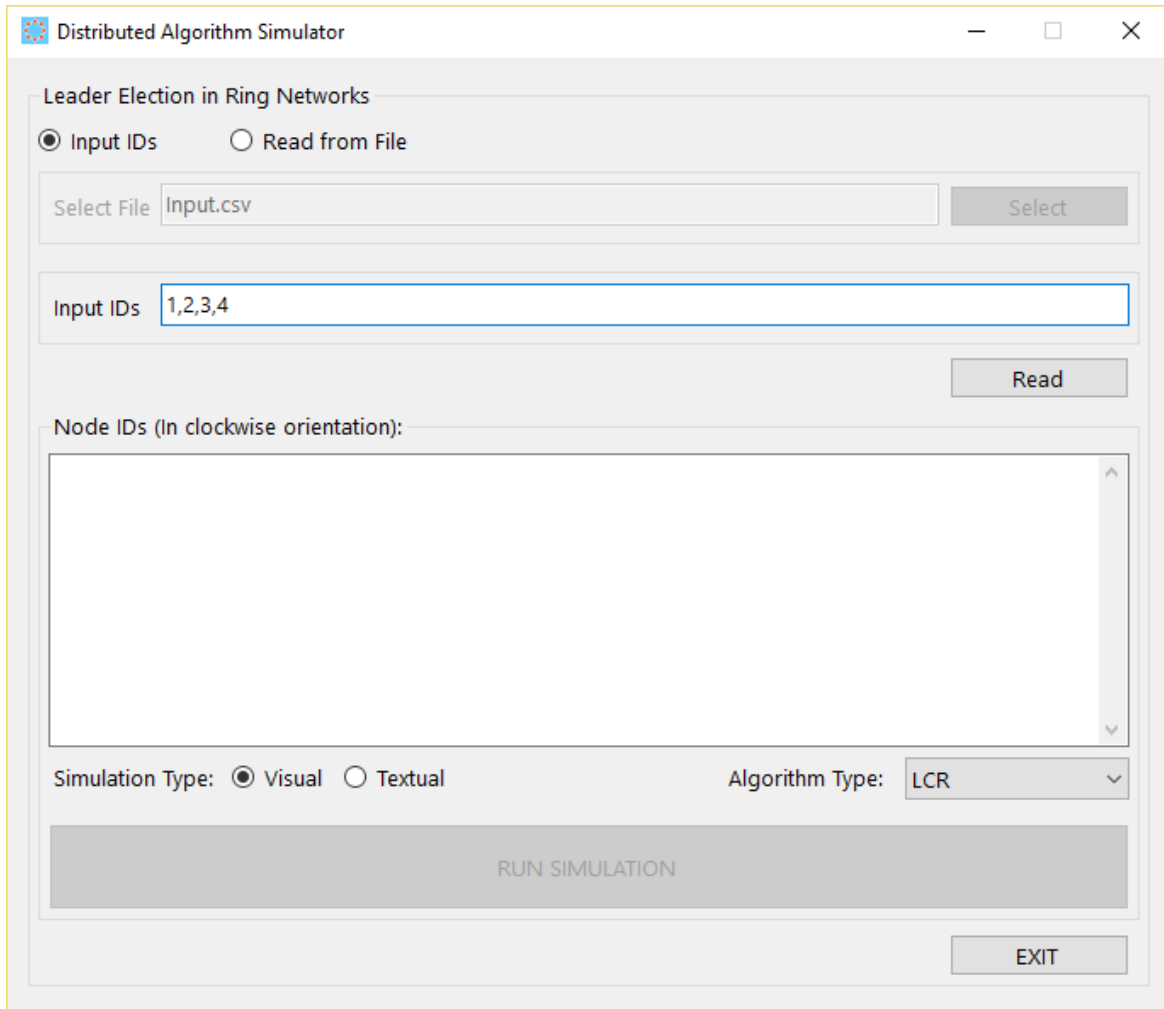
- Select the [Input IDs] radio button. [Input IDs] textbox will be enabled.

Figure 38: Selecting [Input IDs] radio button.



- Type into the [Input IDs] textbox. Input IDs must be non-negative integers, separated by commas. There is no limit on the number of IDs that can be entered.

Figure 39: Entering node IDs.

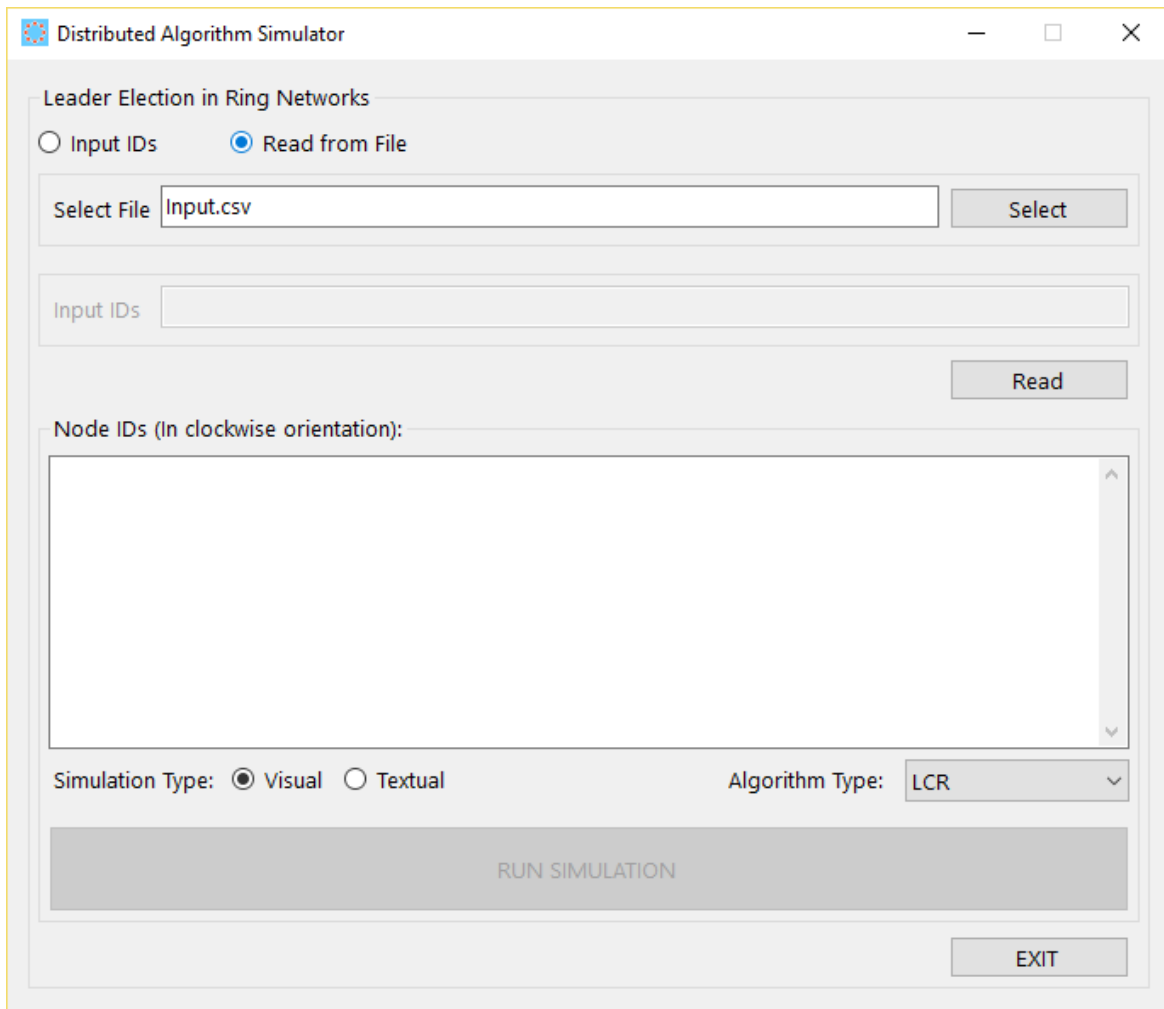


6.3.2 Select an input File

- Select the [Read from File] radio button. [Select File] textbox will be enabled. By default, the textbox will contain the string "Input.csv". If there is a file by the same name in the

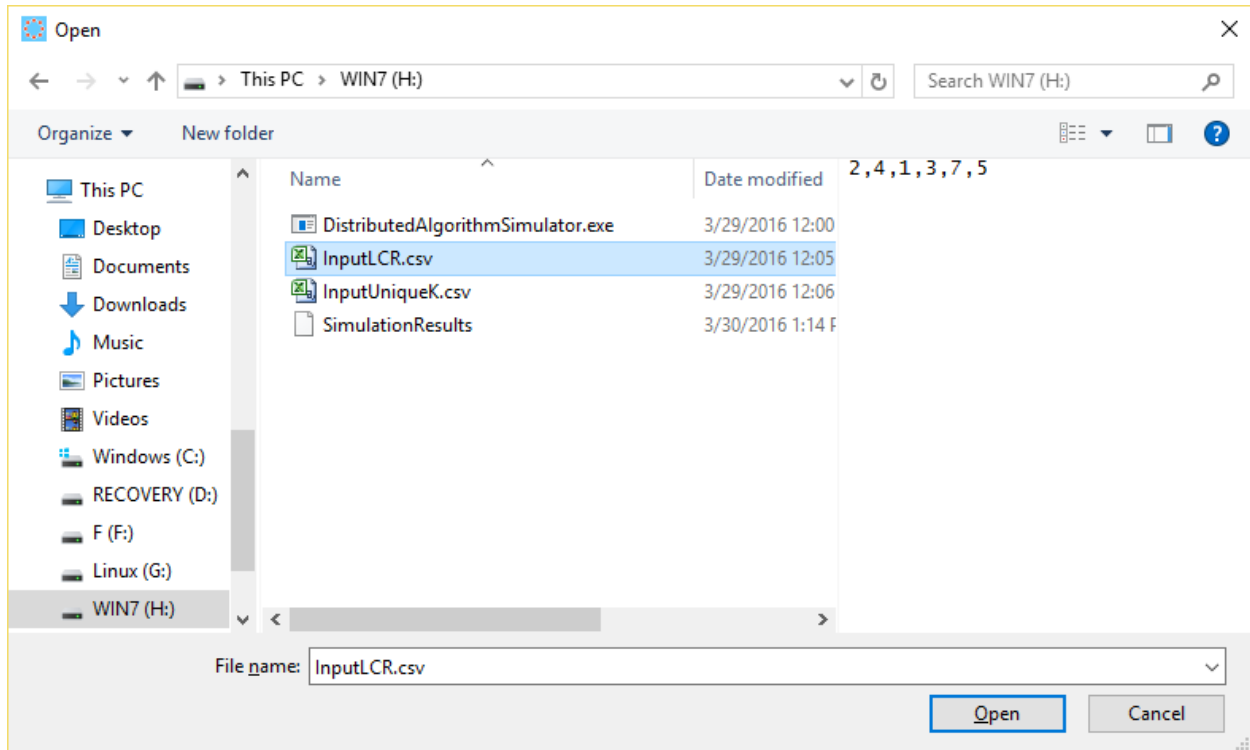
same directory in which the application executable resides, it can be read in. The input file type must be a Comma Separated Values (CSV) file.

Figure 40: Selecting [Read from File] radio button.



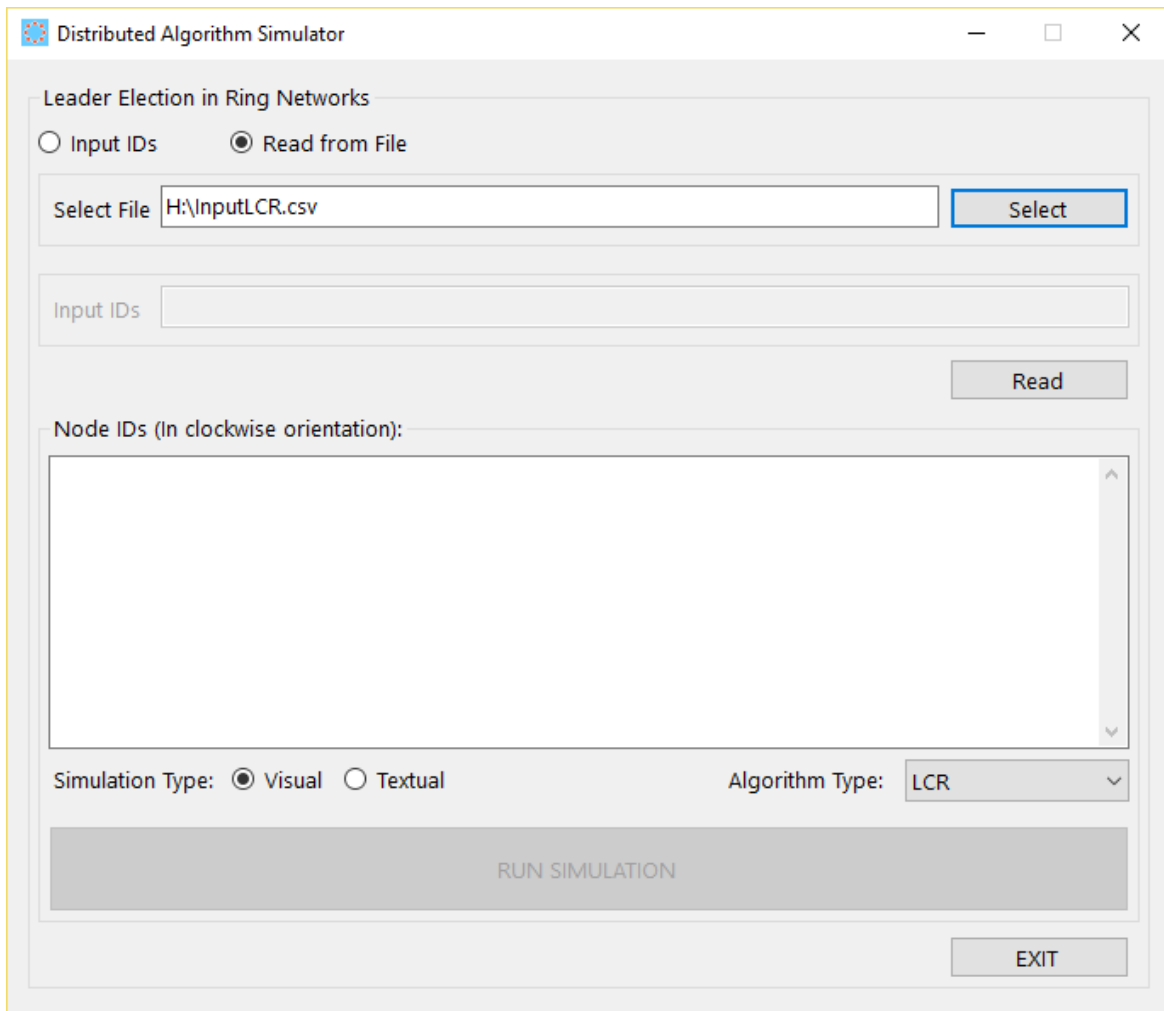
- To select a different file, click the [Select] button. It will open up the [Open] file open dialog box. Select a .CSV file and click the [Open] button.

Figure 41: File open dialog box.



- The file path will be displayed in the [Select File] textbox.

Figure 42: File path of the selected file.

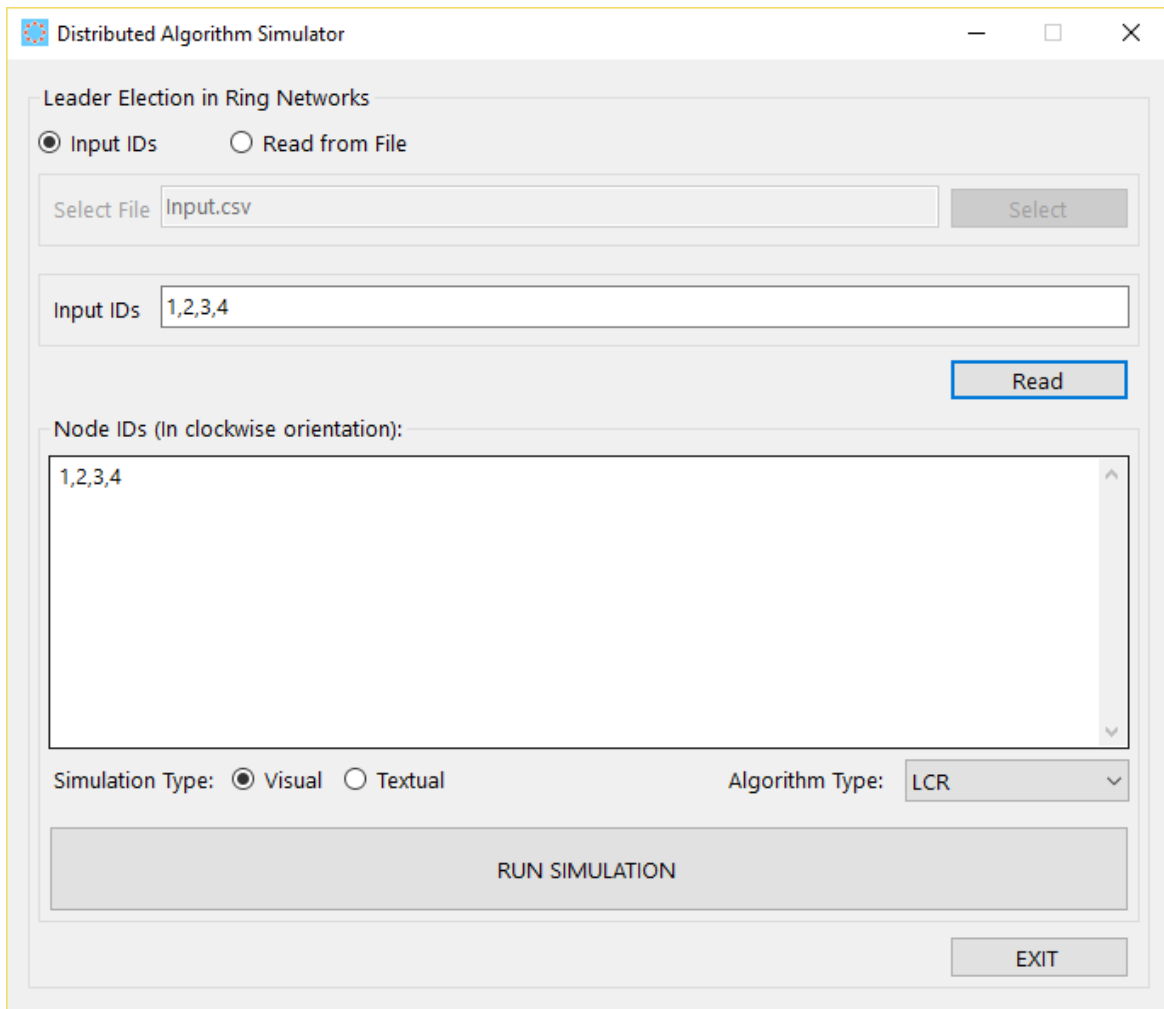


6.3.3 Read Data

- Once either node IDs are typed into the [Input IDs] textbox, or a .CSV file containing the IDs selected, click the [Read] button. The IDs will be read and displayed in the [Node IDs]

textbox, in clockwise orientation, as shown below. The [RUN SIMULATION] button will be enabled at this time.

Figure 43: Reading data from the input source.



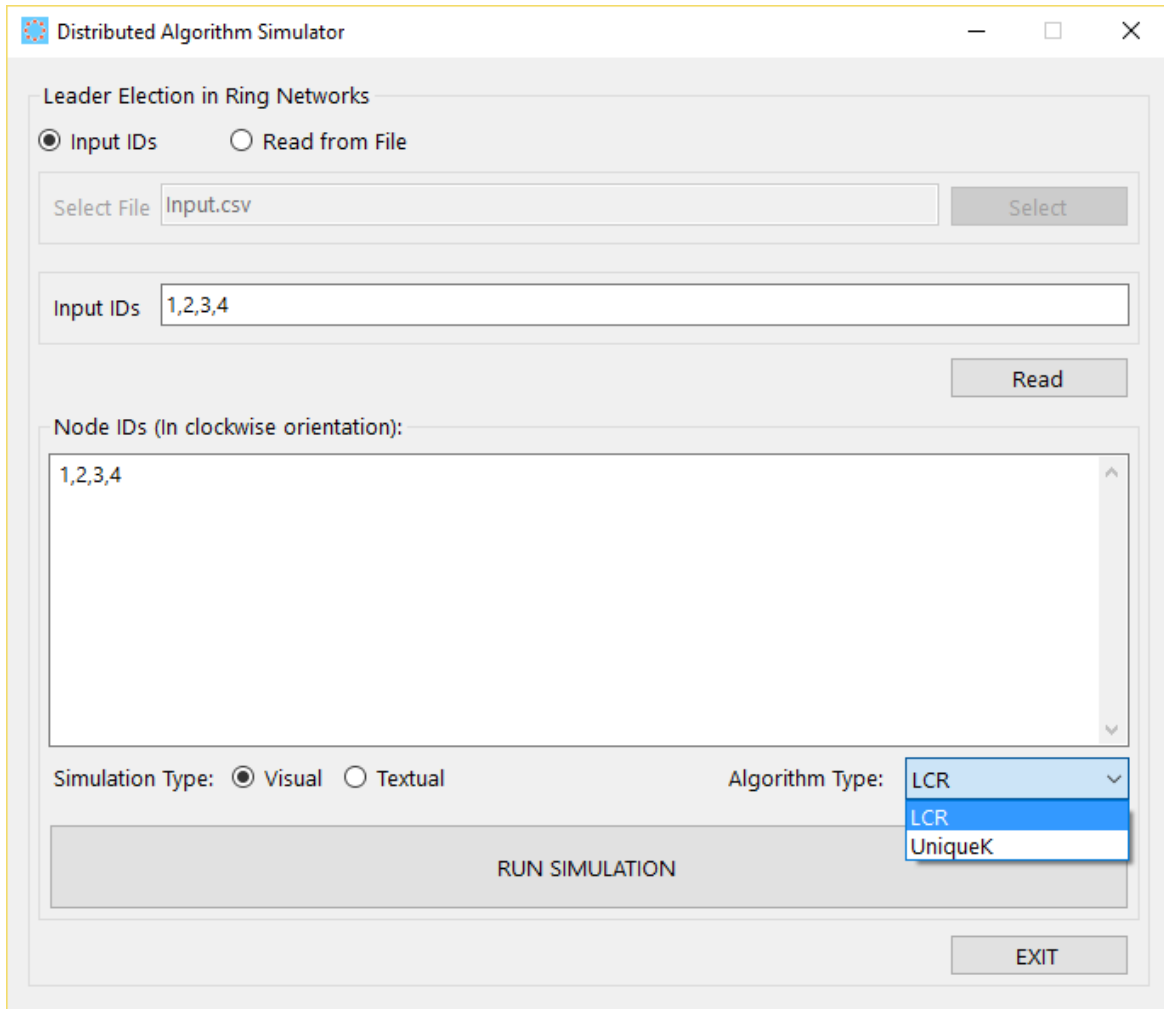
Note: Once the [Read] button is clicked and ID's are displayed, if a modification needs to be done, it must be done either in the [Input IDs] textbox or changes should be made to the .CSV

file, and then read once again using the [Read] button. Editing the displayed IDs in the [Node IDs] textbox will not effect the already read IDs.

6.3.4 Running the Simulation

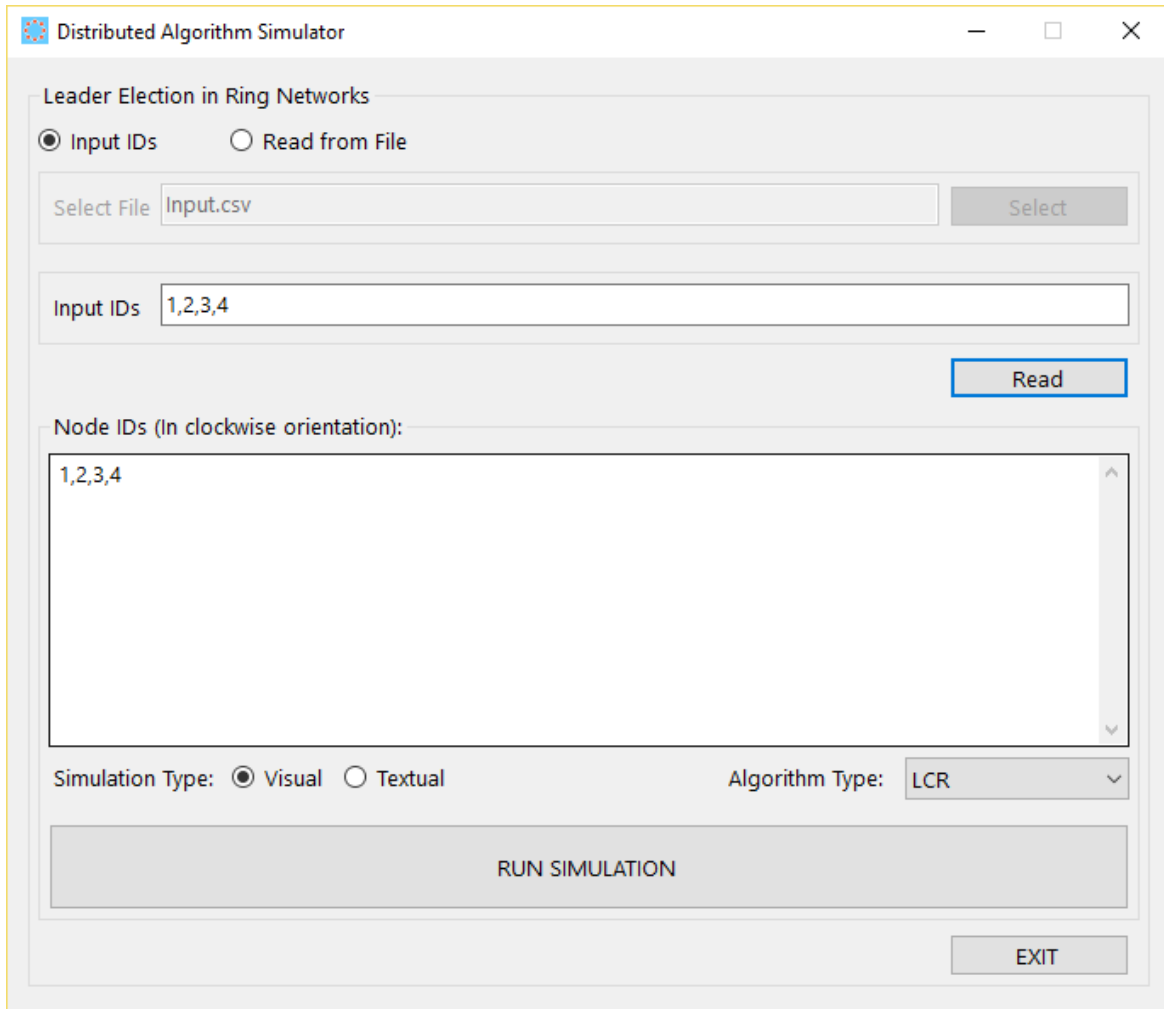
- Read in the node IDs as described in the previous step.
- Select the algorithm from the [Algorithm Type] drop down list.

Figure 44: Selecting an algorithm to simulate.



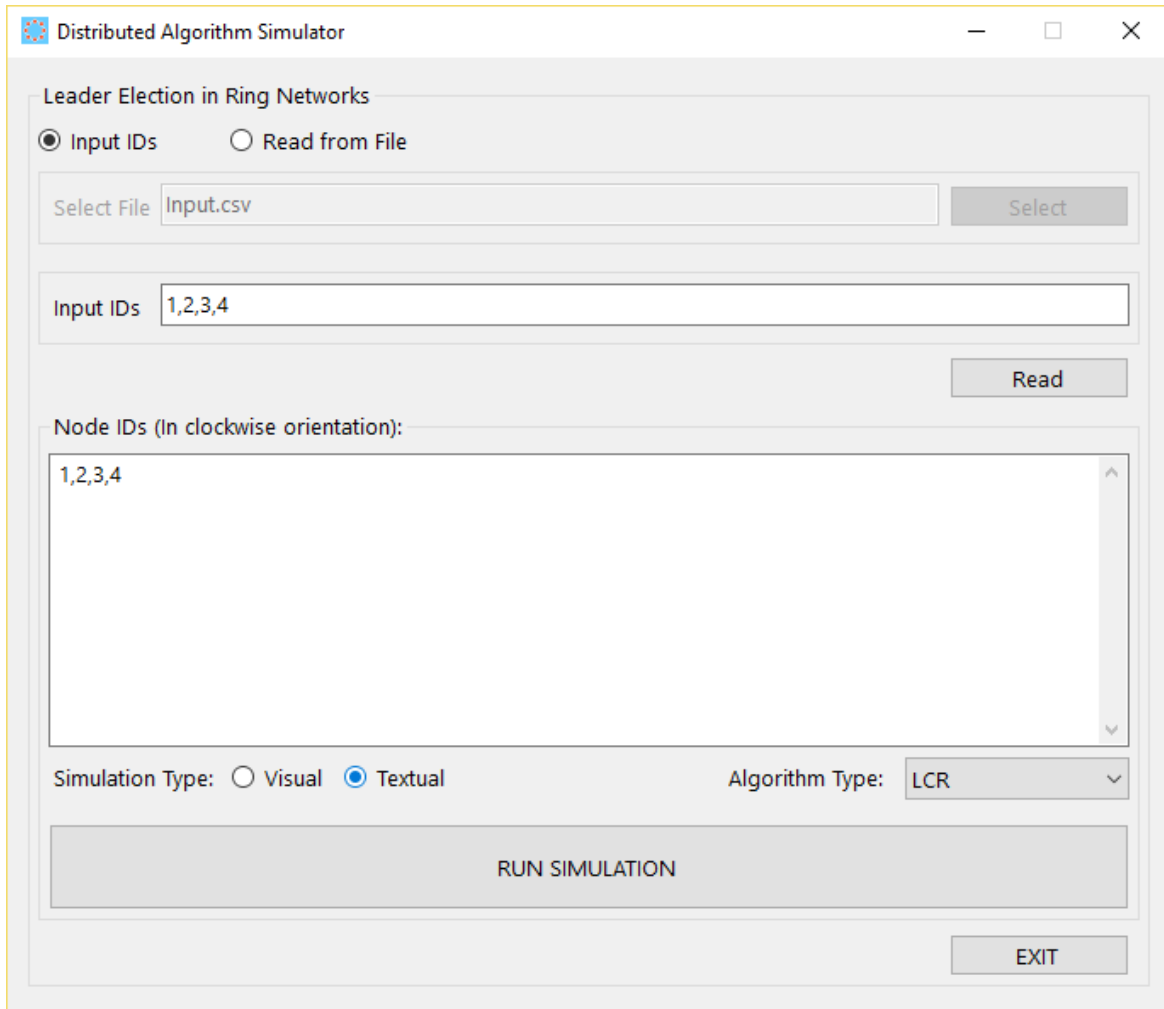
- To run the Visual Simulation, select the [Visual] radio button.

Figure 45: Selecting the Visual Simulation type.



- To run the Textual Simulation, select the [Textual] radio button.

Figure 46: Selecting the Textual Simulation type.

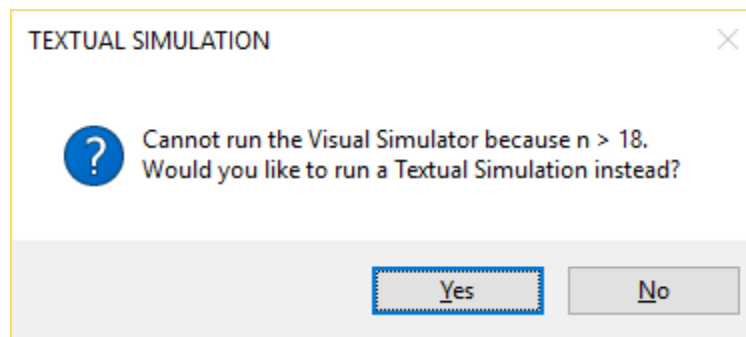


- Click the [RUN SIMULATION] button. It will launch either the Visual Simulation or the Textual Simulation depending on the user choice.

Note: The maximum number of nodes allowed for the Visual Simulation is 18. If the number of nodes in the network is more than that, and the user selected [Visual] radio button, upon clicking

the [RUN SIMULATION] button, user will be prompted to either cancel the simulation or run the Textual Simulation instead.

Figure 47: Prompting user to run a Textual Simulation.



- [Yes] – Runs the Textual Simulation.
- [No] – Cancels the simulation and returns to the main window.

6.4 VISUAL SIMULATOR

Following is the Visual Simulator with the selected algorithm being LCR and the nodes with node IDs 1, 2, 3, and 4 in clockwise orientation, at the initial state. The four navigation icons (Previous, Next, First, and Last) are disabled at first. Navigating through the algorithm can only be done when the simulation is run to the completion at least once, at which point they becomes enabled.

Figure 48: Visual Simulator initial state.



Following legend explains the button icons and their functionality.

Table 68: Visual Simulator button icons and their functionality.

Icon	Meaning	Functionality
▶	Play	Executes the simulation.
⏸	Pause	Pauses the simulation.
▶▶	Next	Advances the simulation by one step in forward direction.
◀◀	Previous	Advances the simulation by one step in backward direction.
▶▶▶	Last	Proceeds to the final step of the simulation.
◀◀◀	First	Proceeds to the first step of the simulation.
🔄	Reset	Resets the simulation.

The four labels at the top left corner of the simulation window displays the round and step numbers. Their meanings are as follows.

Table 69: Descriptions of counters used in Visual Simulator.

Counter	Meaning
N	Number of nodes in the network.
Round	The round number. One round is equivalent to N-steps.
Steps	The number of steps elapsed <i>in the current round</i> .
Total Steps	The number of steps elapsed in the whole execution.

6.4.1 Selecting Variables for Display

The variables to be displayed can be selected using the check boxes in the [Select Variables to Display] group box. Following is an intermediate stage of the execution, with only [Active] and [Leader] variables selected for display.

Figure 49: Displaying only selected variables.



6.4.2 Simulation Speed

The [Speed] numeric up down control lets the user set the speed of the execution. The default value is 1000ms, the minimum allowed is 500ms and the maximum allowed is 5000ms.

6.4.3 Play / Pause

To start the simulation, click the [Play] button. The simulation starts running, and the [Play] button image changes to a [Pause] icon.

Figure 50: Executing the simulation.



Clicking the button again pauses the simulation, and the button icon changes to a [Play] icon.

Clicking a third time resumes the simulation

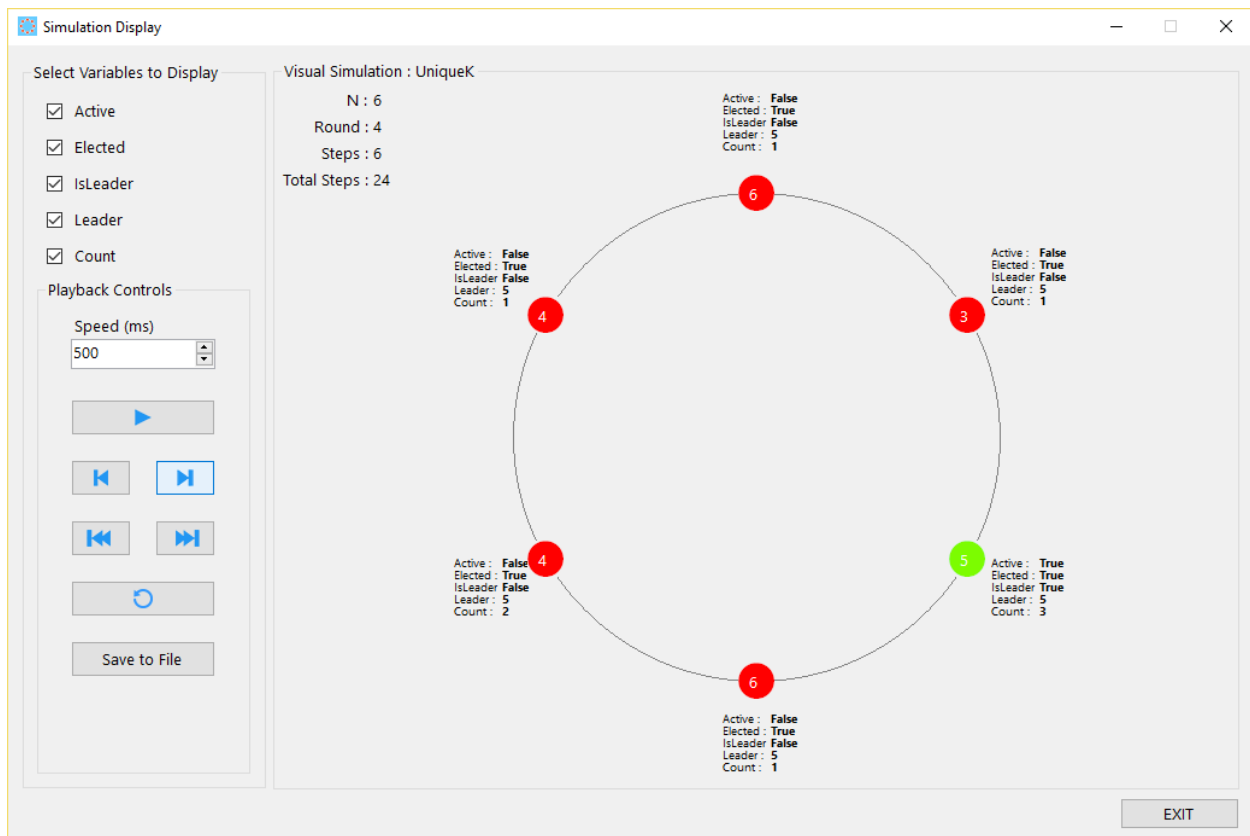
6.4.4 Reset

During the execution, or once the execution is finished, clicking the [Reset] button resets the simulation to its initial state.

6.4.5 Navigation Buttons

The four navigation buttons become enabled once the simulation finishes.

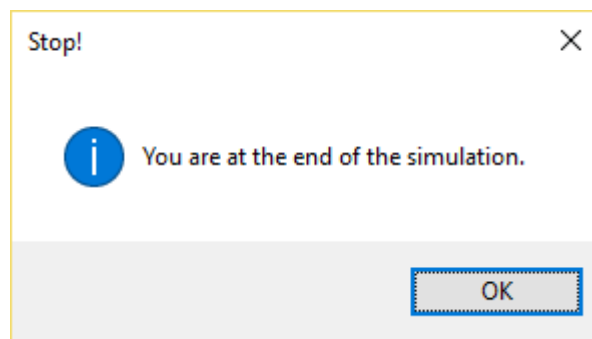
Figure 51: Enabled navigation buttons.



The buttons can be used to navigate through the simulation forward and backward, one step at a time. The status of the variables changes accordingly.

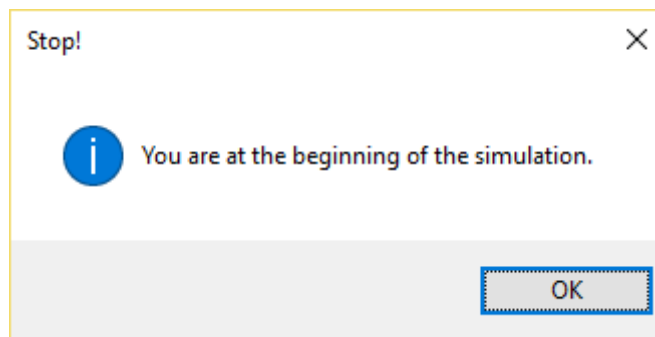
Clicking [Next] or [Last] buttons while at the last step of the simulation generates the following notification.

Figure 52: [End of the Simulation] message.



Clicking [Previous] or [First] buttons while at the first step of the simulation generates the following notification.

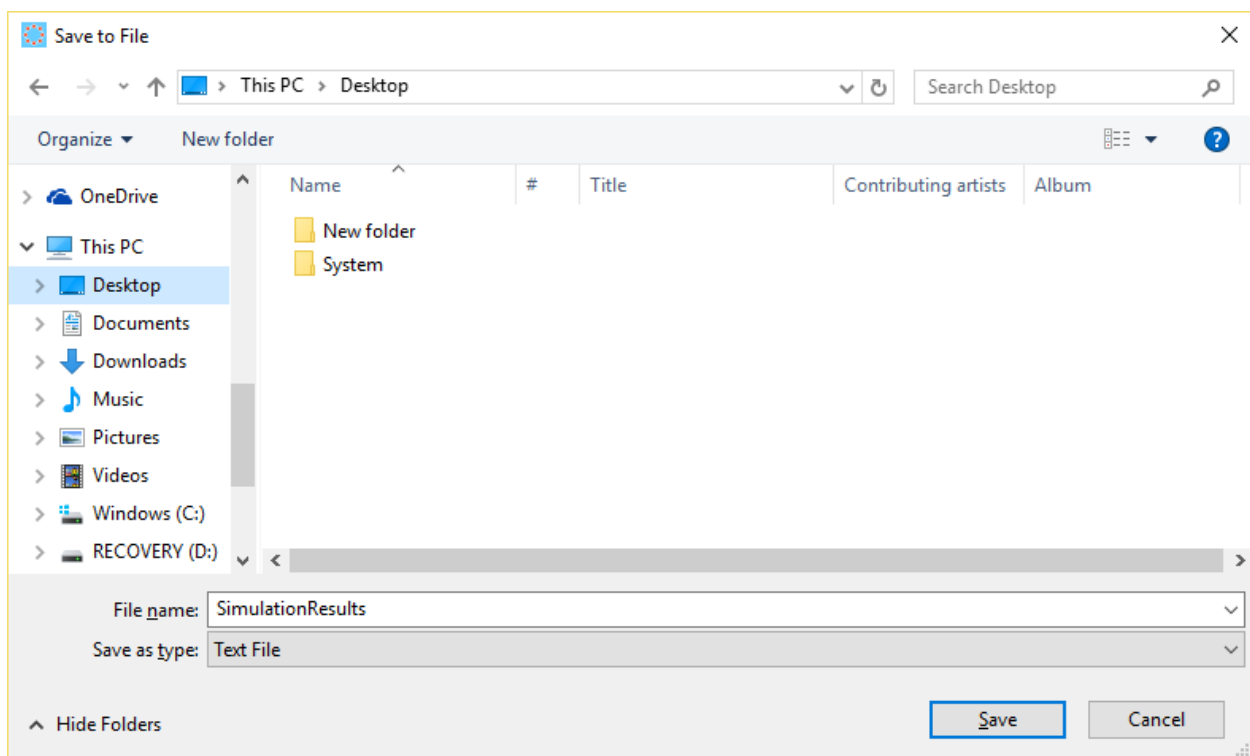
Figure 53: [Beginning of the Simulation] message.



6.4.6 Saving Results to a File

Once the simulation is finished, the results can be saved to a file. Clicking the [Save to File] button opens up the [Save to File] dialog box.

Figure 54: [Save to File] dialog box.



Results can be saved as one of two types of files; .txt or .log. The [Save as type:] drop down lets the user select a type.

Figure 55: Save as a text file.

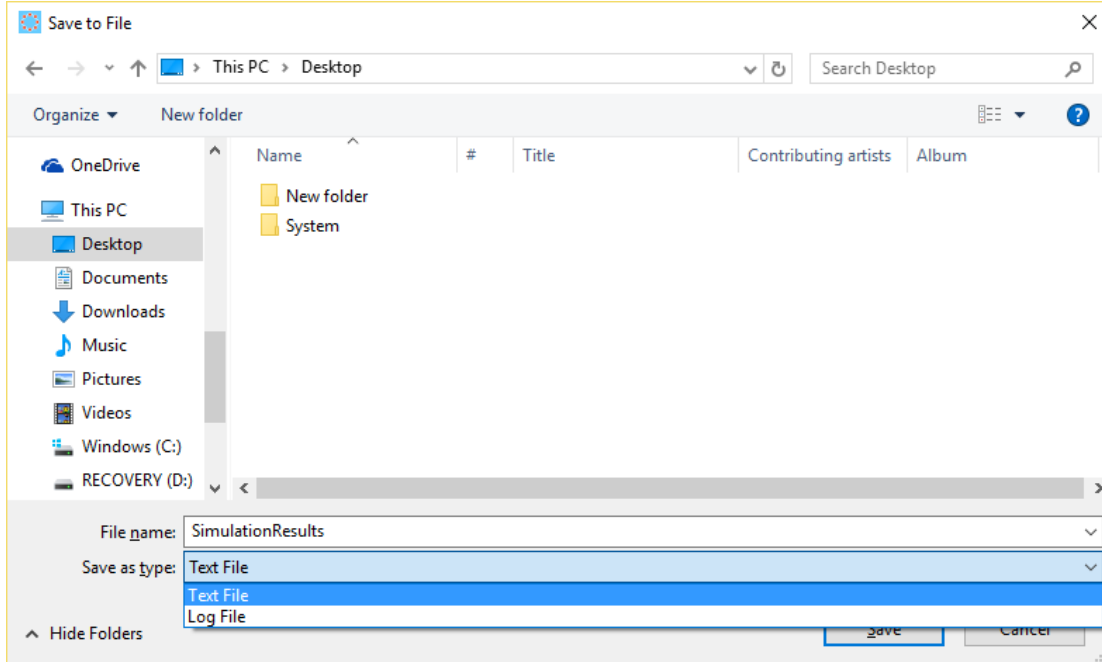
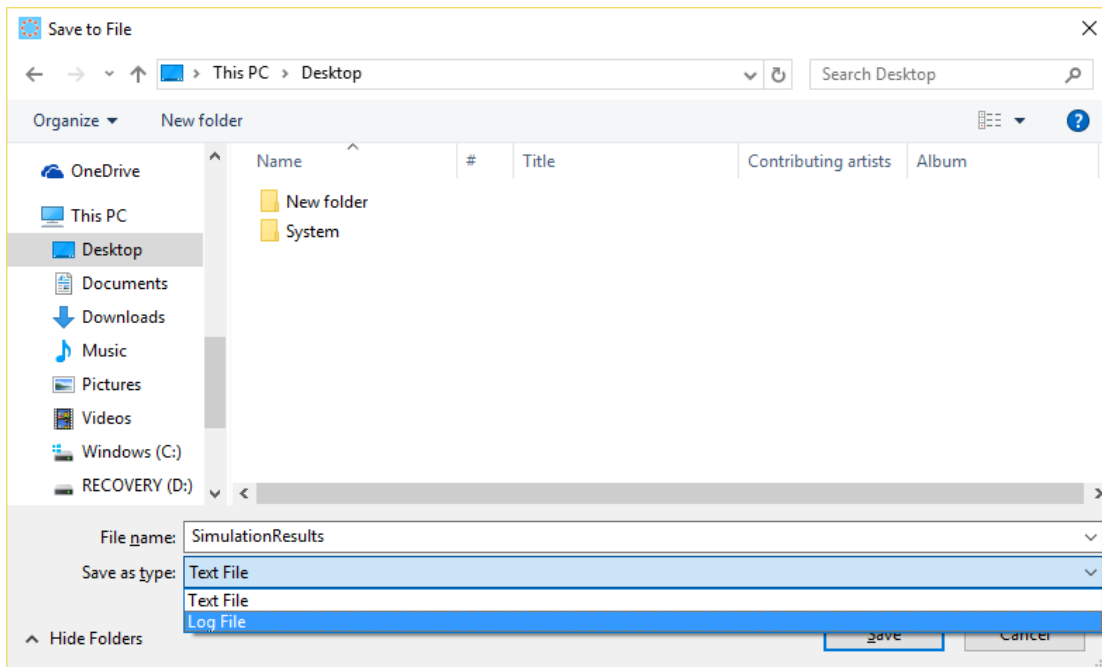


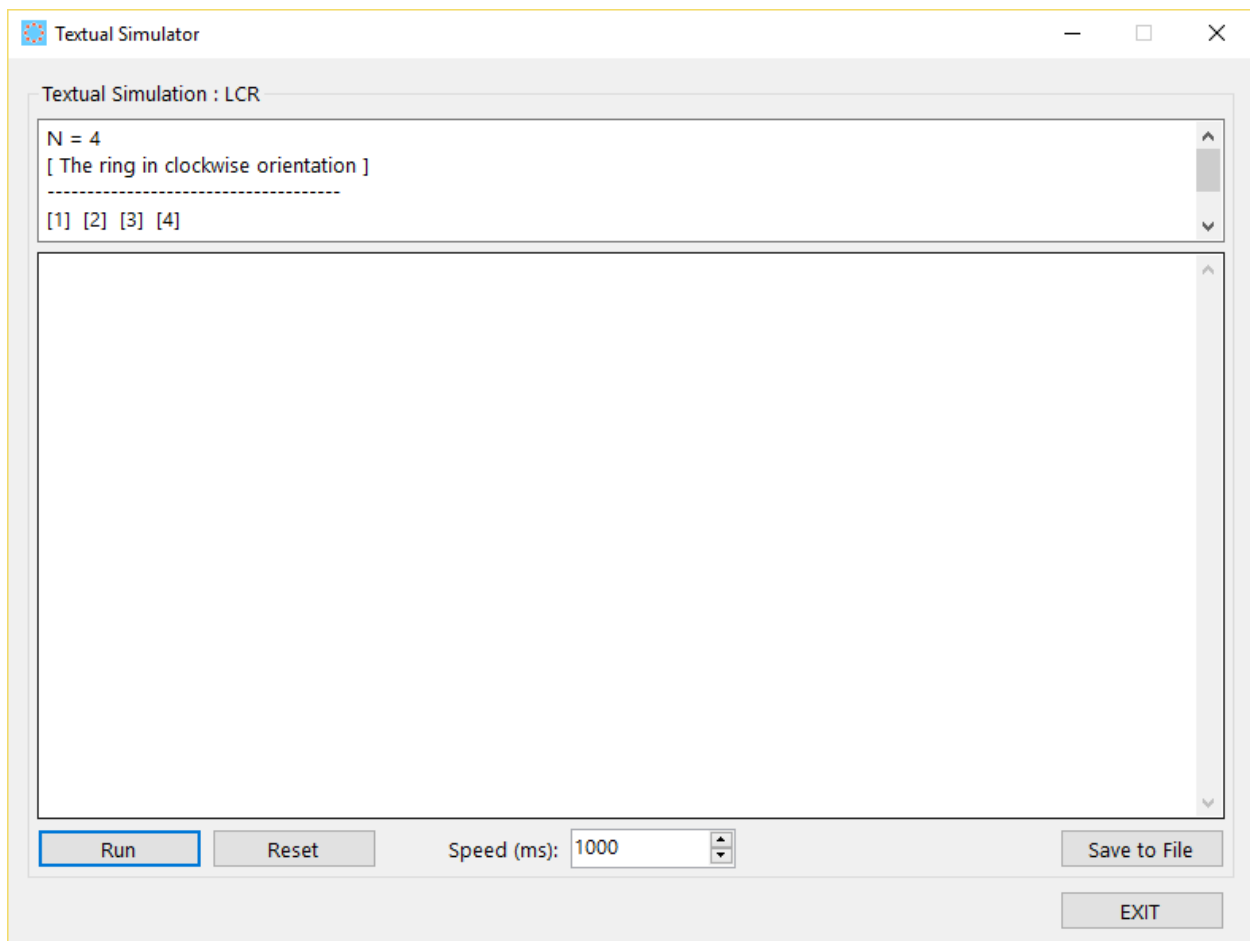
Figure 56: Save as a log file.



6.5 TEXTUAL SIMULATOR

Following is the Visual Simulator with the selected algorithm being LCR and the nodes with node IDs 1, 2, 3, and 4 in clockwise orientation, at the initial state. The topmost textbox displays N, the number of nodes in the network, and the node IDs in clockwise orientation.

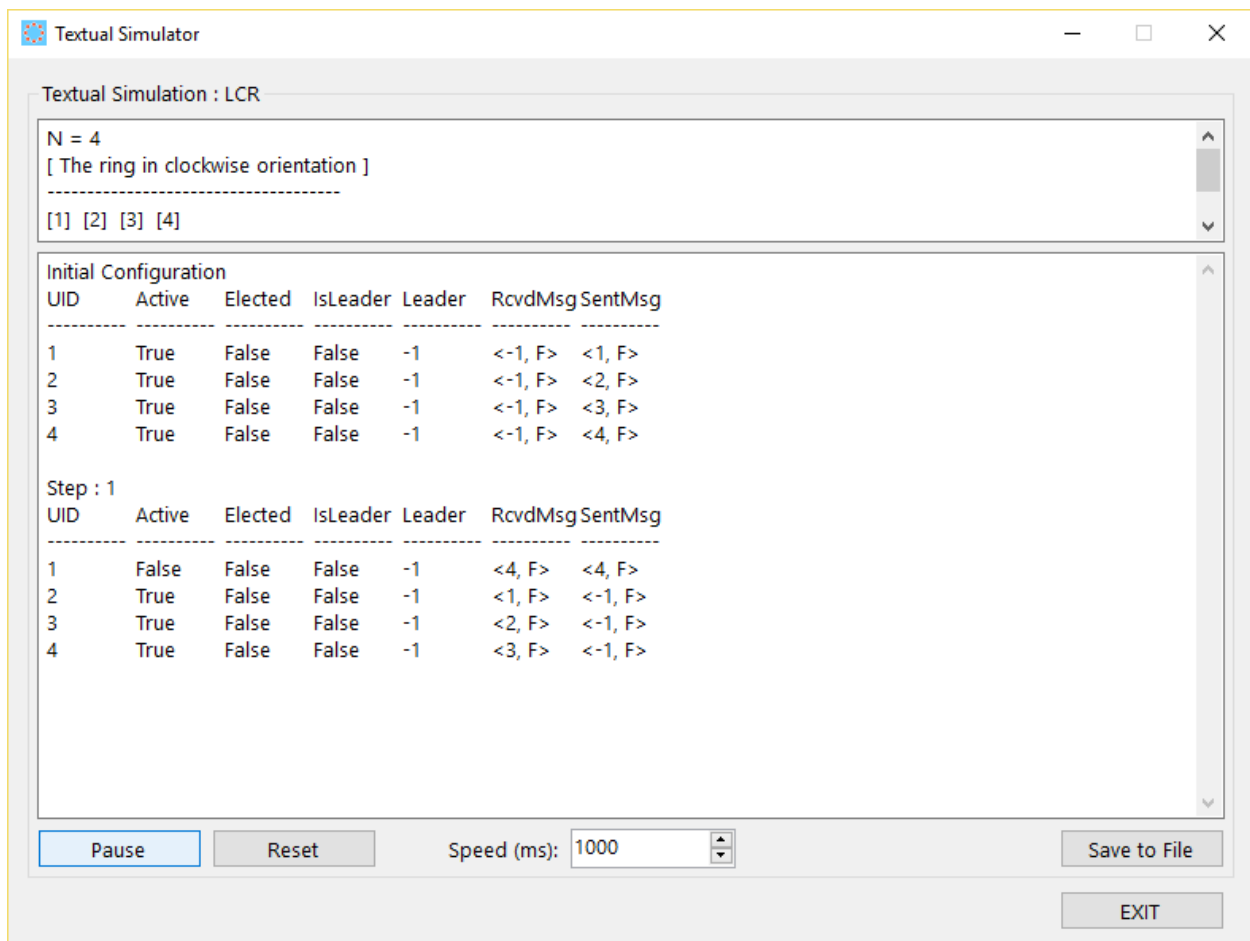
Figure 57: Textual Simulator initial state.



6.5.1 Run / Pause

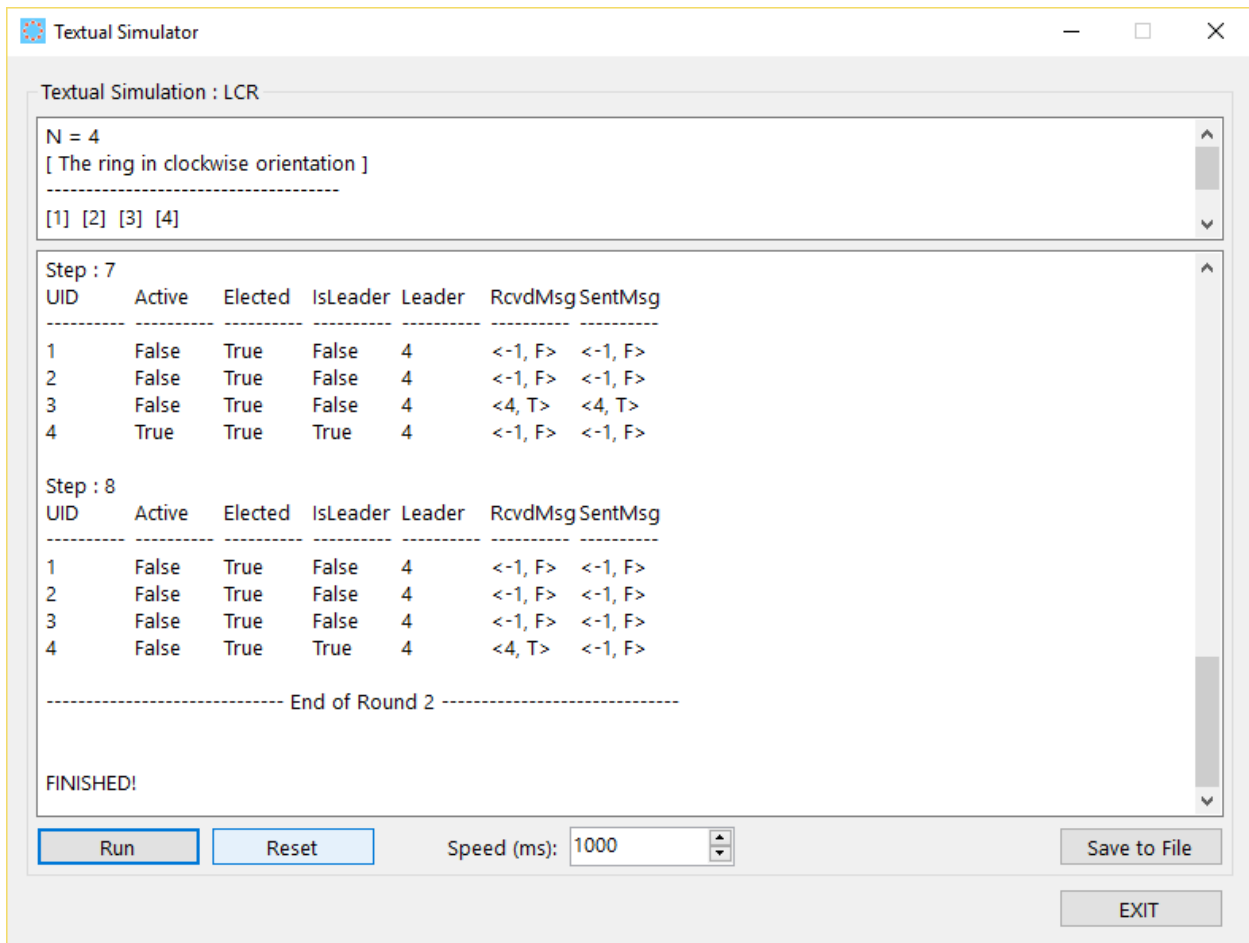
Clicking the [Run] button starts the simulation and changes the button text to [Pause]. Clicking it again pauses the simulation and changes the button text to [Run]. Clicking a third time resumes the simulation.

Figure 58: Running the Textual Simulator.



Once the simulation finishes the execution, the word “FINISHED!” is displayed as the last time of the text box. The [Run / Pause] button text returns to the original [Run] state.

Figure 59: Finished status of the Textual Simulation



6.5.2 Reset

During the execution, or once the execution is finished, clicking the [Reset] button resets the simulation to its initial state.

6.5.3 Simulation Speed

The [Speed] numeric up down control lets the user set the speed of the execution. The default value is 1000ms, the minimum allowed is 100ms and the maximum allowed is 5000ms.

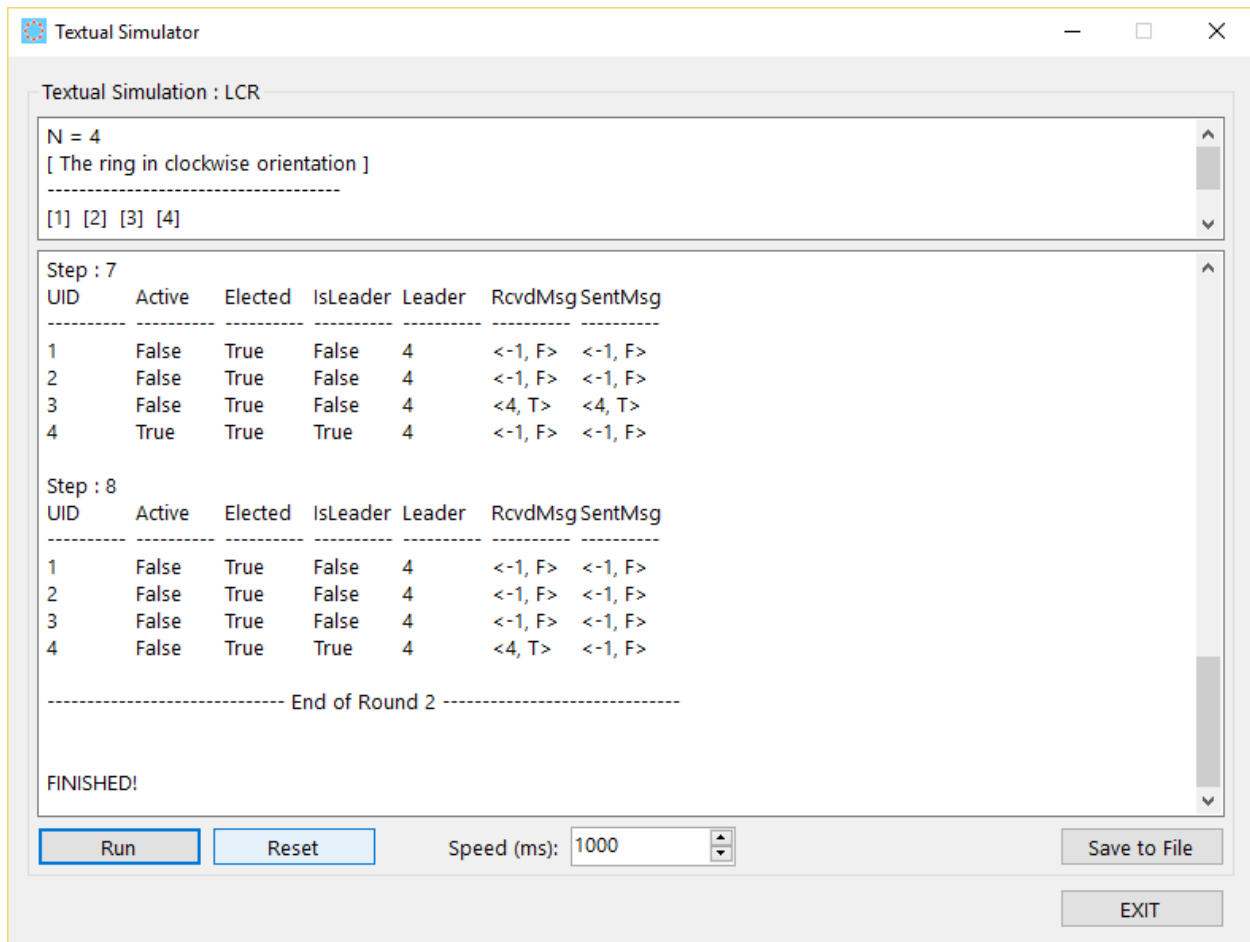
6.5.4 Saving the Results

Once the simulation is finished, the results can be saved to a file by clicking the [Save to File] button. Its functionality is identical to the [Save to File] button in the Visual Simulation.

6.5.5 Output Format of the Textual Simulation

The textual simulation takes the following format.

Figure 60: Output format of the Textual Simulation.



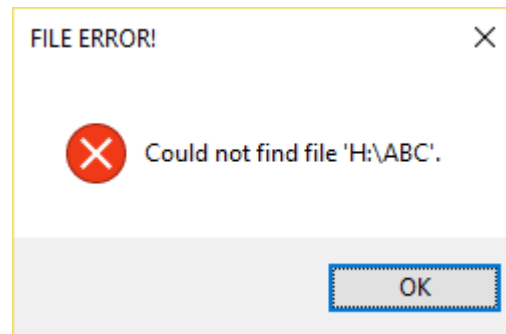
The state of each variable of each node is displayed as a group, under the corresponding step number (total steps in the execution). At the end of each round, the string “End of Round {round number}” is displayed.

6.6 ERROR MESSAGES

6.6.1 File Errors

If the specified file path or the file name cannot be found, the following error message is displayed.

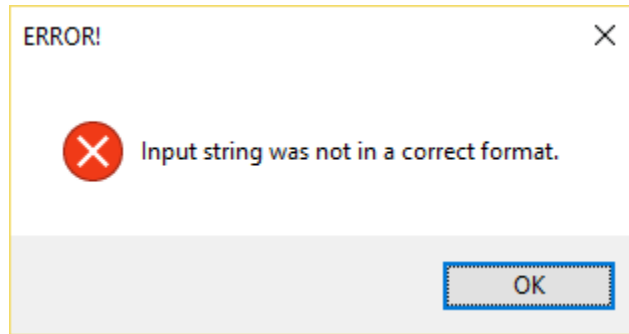
Figure 61: [Could not Find the File] error message.



6.6.2 Input Errors

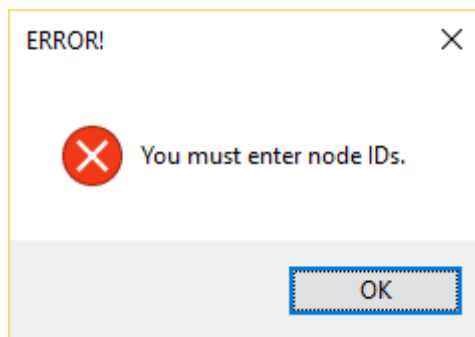
If the user input node IDs in the in the [Input IDs] textbox, or the node IDs listed in the input file are not according to the constraints (must be non-negative integers, separated by commas, in one line), the following message is displayed.

Figure 62: [Incorrect Input String Format] error message.



If the [Read] button is clicked when the [Input IDs] textbox is empty, the following message is displayed.

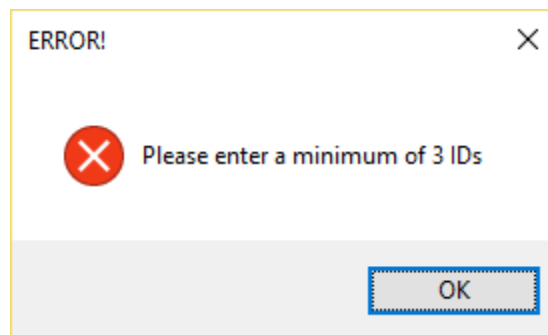
Figure 63: [Node IDs not Entered] error message.



6.6.3 Node ID Errors

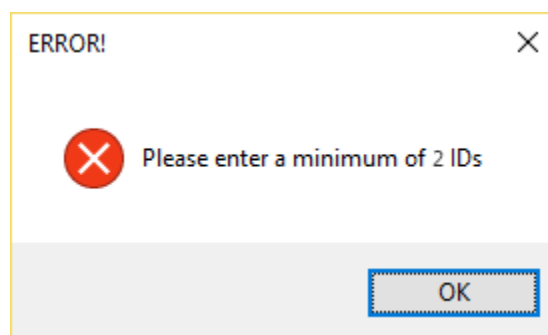
While less than 3 node IDs are entered, if the UNIQUE_ k algorithm is chosen and [RUN SIMULATION] button is pressed, the following error message is displayed.

Figure 64: [Insufficient number of Node IDs] error message for UNIQUE_ k .



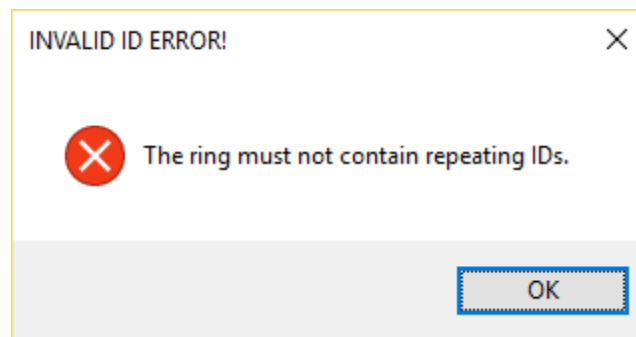
While less than 2 node IDs are entered, if the LCR algorithm is chosen and [RUN SIMULATION] button is pressed, the following error message is displayed.

Figure 65: [Insufficient number of Nodes IDs] error message for LCR.



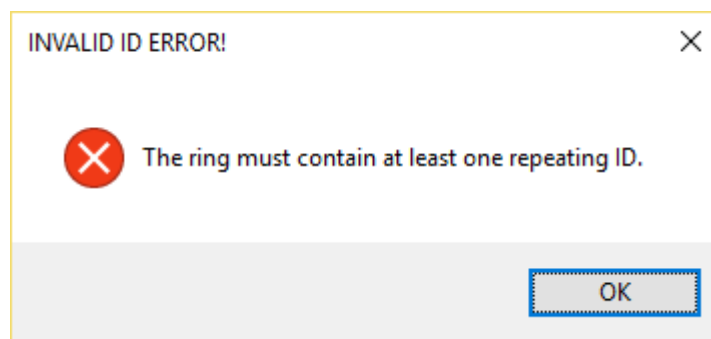
If the LCR algorithm is chosen and the node IDs contain one or more repeating IDs, the following error message is displayed.

Figure 66: [Ring Contains Non-Unique IDs] error message.



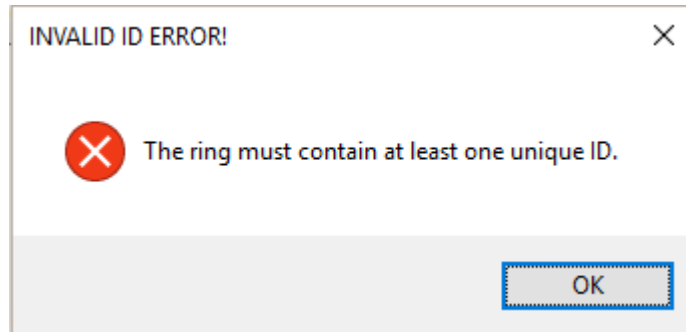
If the `UNIQUE_k` algorithm is chosen and the node IDs contain no repeating IDs, the following error message is displayed.

Figure 67: [Ring does not Contain Repeating IDs] error message.



If the `UNIQUE_k` algorithm is chosen and the node IDs does not contain at least one repeating ID, the following error message is displayed.

Figure 68: [No Unique ID] error message.



Chapter 7

Conclusion and Future Work

This chapter concludes the thesis and offers some suggestions for related future work.

7.1 CONCLUSION

The Distributed Algorithm Simulator is an application program designed to simulate, in a non-distributed environment, the execution of distributed leader election algorithms. The distributed nature of these algorithms sometimes makes it difficult to comprehend, especially when learning them for the first time, and as such we believe this would be a useful tool in the classroom.

Particularly, the ability of the Visual Simulator to step through the execution of an algorithm, not only forward but also backward, could be very useful when analyzing how the algorithms work at each step. The ability to save the results for later analysis is another feature that, we believe, would be a useful teaching tool.

7.2 FUTURE WORK

The Distributed Algorithm Simulator is developed using the Visual C# programming language on the Microsoft .NET framework. As such, it can only be used on Microsoft Windows platforms

(unless a Virtual Machine is being used). It would be more useful if it could be implemented using a platform independent language such as Java, as there are a large number of students and teachers who use other operating systems such as Mac OS or Linux-based systems. It would be even more useful if it can be converted into a web application and/or a mobile app, which would increase its usability.

Finally, the Distributed Algorithm Simulator can only simulate leader election algorithms in ring networks. If it could be extended to simulate leader election algorithms in other network topologies, that would make this a more comprehensive learning tool.

Bibliography

- [1] – Wikipedia contributors. Termite. Wikipedia, The Free Encyclopedia, <https://en.wikipedia.org/wiki/Termite>, March 2016.
- [2] – Hill, Michael. Distributed Computing: An Unstoppable Brute Force. SANS Institute, 2004.
- [3] – Wikipedia contributors. *Beowulf cluster*. Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Beowulf_cluster, January 2016.
- [4] – Peleg, D. Distributed Computing: A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics, 2000.
- [5] – Wikipedia contributors. *Einstein@Home*. Wikipedia, The Free Encyclopedia, <https://en.wikipedia.org/wiki/Einstein@Home>, March 2016.
- [6] – "BOINC Stats – Albert@home". <http://boincstats.com/en/stats/127/project/detail>, Retrieved 2012-02-17.
- [7] – Wikipedia contributors. *Big and Ugly Rendering Project*. Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Big_and_Ugly_Rendering_Project, March 2015.
- [8] – Wikipedia contributors. *Climateprediction.net*. Wikipedia, The Free Encyclopedia, <https://en.wikipedia.org/wiki/Climateprediction.net>, February 2016.
- [9] – "Detailed user, host, team and country statistics with graphs for BOINC". [boincstats.com](http://boincstats.com/en/stats/2/project/detail). <http://boincstats.com/en/stats/2/project/detail>, Retrieved 2010-12-13.

- [10] – Wikipedia contributors. *Folding@home*. Wikipedia, The Free Encyclopedia, <https://en.wikipedia.org/wiki/Folding@home>, February 2016.
- [11] – Pande Lab (2015). "The Science: Protein Folding". Stanford University. <http://folding.stanford.edu/home/the-science>, Retrieved October 9, 2015.
- [12] – Wikipedia contributors. *SETI@home*. Wikipedia, The Free Encyclopedia, <https://en.wikipedia.org/wiki/SETI@home>, March 2016.
- [13] – Wikipedia contributors. *Leader election*. Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Leader_election, March 2016.
- [14] – Attiya, H. and Welch, J., Distributed Computing: Fundamentals, Simulations and Advance Topics, John Wiley & Sons inc., 2004, Chapter 3.
- [15] – Gupta, I., van Renesse, I., and Birman, K. P. A Probabilistically Correct Leader Election Protocol for Large Groups, Technical Report, Cornell University, 2000.
- [16] – Lynch, N. A. Distributed Algorithms. Morgan Kaufmann Publishers, Inc, 1996, Chapter 3.
- [17] – Datta, A. K., Larmore, L. L. Leader Election in Unidirectional Rings. CS780 – Distributed Algorithms class notes, Department of Computer Science, University of Nevada Las Vegas, 2015.
- [18] – Jacot-Descombes, O., StackOverflow contributor. StackOverflow, <http://stackoverflow.com/>, March 2016.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Sugeeswara Gurudeniya

Degrees:

Master of Science in Computer Science, 2016

University of Nevada, Las Vegas

Thesis Title: A Simulator Application for Distributed Leader Election Algorithms

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.

Committee Member, Dr. Yoohwan Kim, Ph.D.

Committee Member, Dr. John Minor, Ph.D.

Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.