NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

DisJ - Reactive Distributed Simulation in Java

by

Nothapol Piyasin, B.C.S.

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

Department of Computer Science

Carleton University

Ottawa, Ontario

April, 2012

©Copyright

Nothapol Piyasin, 2012

UMI Number: MR93707

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.

Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

The designing of reactive distributed algorithms has many challenges, both theoretical and in implementation. Theoretical challenges include the computing environment being different than single process computing. Implementation challenges include the necessity of handling various constraints of the underlying systems. A simulation is an important tool for the algorithm designer to overcome these challenges and be able to solely focus on the design of the algorithm. Many simulation systems have been built, but they have many limitations

This thesis presents DisJ, a reactive distributed simulation system written in Java. DisJ provides a full cycle of algorithm development from writing a protocol to generating simulation results. DisJ supports three different types of distributed communication models: Message Passing, Mobile Agent with Whiteboard, and Mobile Agent with Token. Also, DisJ provides an adversary mechanism for the designer to validate the correctness of the algorithm.

To all mankind who are righteous

Acknowledgments

In the name of God, Most Gracious, Most Merciful

This thesis would not have been possible without the help and support of many people around me, to only some of whom it is possible to give particular mention here.

Above all, my parents who always give me their unequivocal support throughout. I will never be able to repay them and find words to express my appreciation for their love and support.

I am grateful for the advice and guidance of my principle supervisor, Professor Nicola Santoro, and my second supervisor, Associate Professor Doron Nussbaum. Without their knowledge and kindness I would never have succeeded in this study.

I owe my deepest gratitude to Associate Professor Irwin Reichstein, P' Chadchalee Sinhaseni, P' Anuwath Sinhaseni, and P' Ratana Nawong for their encouragement and support when I most needed it.

Also, I would like to thank my colleagues, Hein Mar, Prab Dhaliwal, and Badr Asiri for their support of my studies and research. Last but not least, special thanks to my proofreaders, Brian Foster, Gerrit De Vynck, and Marijke Large. Without them my thesis writing would have been very difficult to achieve.

Table of Contents

Abstract	iii
Acknowledgments	vi
Table of Contents	vii
List of Tables	xii
List of Figures	xiii
1 Introduction	1
1.1 Framework	1
1.2 Existing Solutions and Problems	2
1.3 Contributions of This Thesis	3
1.4 Thesis Layout	5
2 Background	6
2.1 Overview	6
2.2 DAJ - Distributed Algorithms in Java	9
2.2.1 Introduction	9
2.2.2 Environment	9
2.2.3 Visualization	9

2.2.4	Limitations	10
2.3	T-DAJ - Toolkit for Distributed Algorithms in Java	11
2.3.1	Introduction	11
2.3.2	Environment	11
2.3.3	Visualization	12
2.3.4	Limitations	14
2.4	DAP - Distributed Algorithms Platform by University of Patras, Greece	14
2.4.1	Introduction	14
2.4.2	Environment	14
2.4.3	Visualization	16
2.4.4	Limitations	16
2.5	SinAlgo - Simulation of Network Algorithm	16
2.5.1	Introduction	16
2.5.2	Environment	17
2.5.3	Visualization	17
2.5.4	Limitations	18
2.6	DisASter - Distributed Algorithm Simulation Terrain	19
2.6.1	Introduction	19
2.6.2	Environment	20
2.6.3	Visualization	20
2.6.4	Limitations	21
3	Distributed Environments and Models	22
3.1	Distributed Computing Environments	22
3.1.1	Entities	22
3.1.2	Network Environments	23

3.1.3	Communications	26
3.1.4	Knowledge	28
3.1.5	Cost and Complexity	29
3.2	The Communication Models in the Netscape Environment	30
3.2.1	The Message Passing Model	30
3.2.2	The Mobile Agent with Whiteboard Model	31
3.2.3	The Mobile Agent with Token Model	32
4	Simulation System Requirements	34
4.1	System Requirements Overview	35
4.1.1	The Input Requirements	35
4.1.2	The Output Requirements	38
4.1.3	System Requirements Partition	39
4.2	The Primary Inputs Subsystem	40
4.2.1	Protocol Editor Component	41
4.2.2	Topology Editor Component	45
4.3	The Runtime Execution Subsystem	49
4.3.1	Runtime Execution Overview	49
4.3.2	Simulator Engine Component	51
4.3.3	Logger Component	52
4.3.4	Playback Component	54
4.3.5	Adversary Control Component	55
4.3.6	Replay Component	60
4.4	The Presentations Subsystem	61
4.4.1	Types of Information	62
4.4.2	Live Presentation	64

4.4.3	Final Presentation	65
5	Architecture and Implementation	67
5.1	A Simulation System Design Overview	68
5.2	Software Development Methodology and Approach	70
5.3	DisJ Simulation System Design and Architecture	73
5.3.1	DisJ Simulation System Workflows	73
5.3.2	Loading Group Architecture	75
5.3.3	Runtime Group Architecture	88
5.3.4	Infrastructures	103
5.4	DisJ Implementation	114
5.4.1	Coding Structure and Deployment	114
5.4.2	Reusing Existing Applications and Codes	118
5.4.3	Using Software Design Patterns	121
6	Case Studies	125
6.1	The Ring Election algorithm for the Message Passing Model	126
6.1.1	Problems	126
6.1.2	The Assumptions of the Network Environment	126
6.1.3	The Pseudo Code	127
6.1.4	The Implementation of the As Far Protocol in Java with DisJ	129
6.1.5	The Simulation Results	132
6.1.6	Worst Case Scenario of the As Far Protocol	133
6.1.7	Writing an Adversary Script for the As Far Protocol	134
6.1.8	The Results Comparison	137
6.2	Black Hole Search algorithm for the Agent with Whiteboard Model	138
6.2.1	Problems	138

6.2.2	The Assumptions of the Network Environment	138
6.2.3	The Pseudo Code	139
6.2.4	The Implementation of Co-Locate Agent Black Hole Search Protocol in Java with DisJ	141
6.2.5	The Simulation Results	146
6.2.6	The Results Analysis	148
6.3	A Black Hole Search algorithm for the Agent with Token Model . . .	151
6.3.1	Problems	151
6.3.2	The Assumptions of the Network Environment	151
6.3.3	The Pseudo Code	152
6.3.4	The Implementation of Ping Pong Protocol in Java with DisJ	156
6.3.5	The Simulation Results	159
6.3.6	The Results Analysis	162
7	Conclusion	165
	List of References	168
	Appendix A As Far Protocol in Java	176
	Appendix B Co-Locate Black Hole Search Protocol in Java	183
	Appendix C Ping Pong Protocol in Java	198
	Appendix D DisJ Plug-in Implementation in Java	213
	Appendix E DisJ User Manual: Cookbook	214

List of Tables

2.1	Comparison of some features and aspects of existing simulation systems with the proposed DisJ Simulation System	7
2.2	cont.. Comparison of some features and aspects of existing simulation systems with the proposed DisJ Simulation System	8
2.3	cont.. Comparison of some features and aspects of existing simulation systems with the proposed DisJ Simulation System	8
5.1	The Organization of the Simulation System	68
5.2	Another Possible Organization of the Simulation System	69

List of Figures

2.1	The DAJ Simulation Control Screen Snapshot [6]	10
2.2	The T-DAJ Screen Snapshot form Java Applet [72]	13
2.3	The DAP GUI Architecture [16]	15
2.4	The SinAlgo Screen Snapshot [40]	18
2.5	The DisASTer Global Message Queue View Screen Snapshot [62] . . .	21
3.1	An Example of Different Devices form the Netscape Environment . .	24
3.2	The Mobile Network Environment Deployed in Planar with Obstacles	24
3.3	The Distributed Communication Models in the Netscape Environment	30
4.1	Overall requirements for a Reactive Distributed Simulation System .	36
4.2	The Simulation System and its Subsystems	40
4.3	The Protocol Editor Component Requirements	42
4.4	The Topology Editor Requirements	46
4.5	The Runtime Execution Subsystem	50
4.6	The Simulator Engine Lifecycle	52
4.7	The Types of Collected Information	62
4.8	The Information Presentations Formats	65
5.1	Use case for the user input data directly into the system	75
5.2	Use case for the user that uses the system to produce data and then input the data back into the system	75

5.3	The User Inputs Workflow	76
5.4	Outputs Workflow	77
5.5	Execution Workflow	78
5.6	The components grouped by the time of activity	79
5.7	Topology Editor Class Relationship with GEF	81
5.8	The Topology Editor Internal Components	81
5.9	The Protocol Editor Internal Components with Java Implementation	84
5.10	Distributed Communication Model Libraries that DisJ Supports . . .	86
5.11	The Flow of Runtime Information Process	88
5.12	Simulator Engine Internal Components	89
5.13	Simulator Engine Internal Components Relationship	90
5.14	DisJ Simulation Log Content	93
5.15	DisJ Log Class Relationship	93
5.16	DisJ Log and Network Element Classes Relationship	94
5.17	Playback Animation and Report are Visual Display of the DisJ Simu- lation System	95
5.18	The MVC Architecture for DisJ Views	97
5.19	Relationship between Adversary Script Instructions, Default Instruc- tions, and Simulator Engine	99
5.20	Radom Number Generator and Adaptation of External Distribution .	100
5.21	The Replay Main Workflow	102
5.22	The DisJ Plug-in with the Eclipse Platform [73]	104
5.23	Mapping the infrastructures that the DisJ Simulation System Compo- nents reused from the Eclipse Platform	105
5.24	The System Log View when system error occurs in the DisJ simulation system	107

5.25	Protocol Editor with JDT plug-in	108
5.26	Topology Editor with a Palette on the Righthand Side	109
5.27	The Eclipse Workbench that Contains Some of Default GUI Components	111
5.28	The DisJ View that extends Eclipse View	111
5.29	The Outline of the Topology Editor in the Eclipse Outline View . . .	112
5.30	The Outline of the Protocol Editor in the Eclipse Outline View . . .	113
5.31	The Properties View is Displaying a Topology Information	114
5.32	GUI for DisJ Build Configuration and Deployment	117
5.33	The Eclipse IDE Provides Plug-in Export GUI Wizard	118
5.34	GUI Configuration Plug-in Dependency for the DisJ Simulation System	119
5.35	XML Scripts Describes the Extension Plug-ins in Eclipse	120
5.36	The Factory Method Pattern used in the DisJ System	121
5.37	The Faade Pattern used in DisJ	122
5.38	The Template (Algorithm) Pattern used in DisJ	123
6.1	A snapshot of DisJ during the simulation of the As Far protocol. . . .	132
6.2	The Ring Topology Connected in the Worst Case Scenario	134
6.3	A snapshot of the User Adding the Adversary Script into DisJ	136
6.4	A snapshot of DisJ during the simulation of the As Far protocol with the FifoDelay Adversary Script	136
6.5	A snapshot of DisJ during the simulation of the Co-Locate Black Hole Search protocol	145
6.6	A snapshot of the Agent View during the simulation of DisJ	145
6.7	A Graph Comparison Results of the Protocol with Three Different Locations of Black Hole	147

6.8	A snapshot of the DisJ view that show the Whiteboard of a last safe node of the dead agent in an ordered ID Ring topology of size 50, where a home base is at n7 and a back hole is exact opposite the home base, n32	149
6.9	A snapshot of the DisJ view that show the Whiteboard of a last safe node of the dead agent in an ordered ID Ring topology, where a home base is at n7 and a back hole is besides the home base, n6	150
6.10	A snapshot of the DisJ view that show the Whiteboard of a last safe node of the dead agent in an ordered ID Ring topology of size 50, where a home base is at n7 and a back hole is at n26	151
6.11	A snapshot of DisJ during the simulation of the Ping Pong protocol .	159
6.12	A snapshot of the Node View during the simulation	160
6.13	A snapshot of graphical report during the simulation	160
6.14	A Graph Comparison Results of the Protocol with Three Different Locations of Black Hole	163

Chapter 1

Introduction

1.1 Framework

In modern technology, there are many systems that use reactive distributed algorithms such as communication services, security controls, system coordination, pervasive applications etc. These systems are commonly used and are important to many applications in the modern world. A distributed system is made up of a large number of entities located in physically distributed environments such as search engine servers, telephone switches, desktop computers, mobile devices etc. These entities require communication exchanges amongst themselves in order to achieve a common goal. The communication exchanges are done through a network of communication media, both wired and wireless that, by nature, are asynchronous and unreliable, which makes these systems very complex. The complexity of these systems makes the development of distributed algorithms perhaps the most difficult type of algorithms to successfully work with. For example, routing telephone calls in a telephone network requires constructing virtual networks and routing tables in order to forward the calls more efficiently. The virtual networks and the routing tables are required to be updated periodically due to the stability of entities in the telephone network.

The construction of the virtual networks and routing table is a very challenging task because there are many factors to consider, such as the size of the network, the different types of priority of telephone calls, the dynamic of the network structure due to the mobility of devices, and the loss of connections due to power shortages, the device shutdown or the device malfunctions, etc. Therefore, the construction requires sophisticated algorithm, which the performance of the algorithm may affect the quality of service. The routing table construction has been researched and used in various network environment [59] [2] [48] [41].

The development of any algorithm involves more than just writing a protocol. It also involves a complex process of testing and verifying correctness of that protocol. The challenges of developing a protocol for distributed systems are compounded by the fact that such systems are physically distributed and consist of a large number of entities. Therefore, one of the most important problems that any developer must overcome when working on these systems is achieving cost effectiveness during development cycle. This can be achieved by having an efficient tool or mechanism that can help in developing, testing and verifying algorithms in a large number of distributed entities and various network environment configurations.

1.2 Existing Solutions and Problems

There are several existing technologies that provide communication tools and infrastructure for developing reactive distributed algorithms including PMV [33] [78], MPI [28] [27], RMI [39] [88], COBRA [63] etc. Unfortunately the learning curve for these technologies is very steep and the libraries provided are mainly for the network communication layer. The developer is individually responsible for creating higher layers of libraries specifically designed for applications and testing reactive

distributed computing. Furthermore, these technologies do not provide a test and verification mechanism for the algorithms, meaning that the developer must create such a tool or use some existing network simulation tools to test and verify the algorithms. However, as the name makes clear, network simulation tools are used to simulate the network communication and data transmission aspects of the algorithms not the algorithms in and of themselves. There are some existing tools and simulations that have been built to simulate reactive distributed algorithms but these systems are also limited in term of usability, capability, and efficiency etc. For example, every system we found during the research supported only the Message Passing model and use only bi-directional link for communication, but do not have adversary supports or uni-directional links. Moreover, there are very few systems that provide the ability to configure the network and provide statistical reports, and simulation replay etc., which are very important for testing and verifying correctness of the algorithms. The discussion of existing systems will be elaborated more in next chapter.

1.3 Contributions of This Thesis

The development of reactive distributed algorithms is different and more troublesome than the development of single process algorithms because of the distributed environment properties such as a large number of asynchronous processes, limitation of data sharing, unreliable network communication, etc., are factors which highly increase the complexity of the algorithms. Hence, it is necessary to have a simulation system that is able to model a distributed environment and execute distributed algorithms in order to test and verify correctness. Moreover, a simulation system should support various distributed communication models and be easy to use. Therefore, this thesis proposes a single processor simulation for distributed algorithms in Java named DisJ,

to fulfill the above requirements in developing a reactive distributed algorithm.

DisJ is a discrete event simulation system meant for developing, simulating, testing and verifying a reactive distributed algorithm in Java, which is executed on a single processor computer. The DisJ simulation system provides a full development cycle of algorithm development tool that supports every step from writing codes to generating simulation results. The objective of the DisJ simulation system is to help develop, test and verify the reactive distributed algorithms more efficiently and cost effectively, helping the developer to be more productive when working on a reactive distributed algorithm.

The DisJ simulation system currently supports three communication models for a fixed Netscape environment e.g. the Mobile Agent with Whiteboard model, the Mobile Agent with Token model and the Message Passing model. The DisJ simulation system is an Eclipse Plug-in that includes the Java IDE and the topology editor, which allows the user to develop a reactive distributed algorithm in the Java language, and to create the network topology needed for a simulation of the algorithm. During a simulation of an algorithm, DisJ provides animation playback that shows the progress of the algorithm, and allows the user to inject an adversary interruption program to verify the algorithm. At the end of the simulation, DisJ generates reports of the simulation for the user to review. DisJ also allows the user to replay past simulations from recorded files without any alteration of the simulation.

Finally, DisJ provides reactive distributed communication API libraries for the user so that the user can fully concentrate on developing reactive distributed algorithms instead of being distracted by low level infrastructures and configurations such as network communication, network configuration, and hardware configuration issues etc.

1.4 Thesis Layout

The thesis consists of seven chapters that explore the background of our works detail proposed contributions, to this field. Chapter 2 presents some of the existing work on reactive distributed algorithm simulation systems. Chapter 3 provides an introduction to aspects of a particular reactive distributed environment. Chapter 4 discusses a simulation system model for a reactive distributed algorithm and its requirements. Chapter 5 explores the architecture and implementation of the DisJ simulation system while Chapter 6 shows some examples and case studies of how to develop reactive distributed algorithms with DisJ. Chapter 7 concludes with a discussion of the future directions and work on DisJ.

Chapter 2

Background

This chapter discusses real world distributed systems development difficulties that motivate the development of distributed simulation systems, then briefly discusses about some interesting distributed simulation systems that have been developed.

2.1 Overview

Distributed systems and applications are commonly used in day to day life, from a mobile phone to an internet search engine. The designing of distributed algorithm for distributed systems has many challenges both at the theoretical and implementation levels [75]. At the theoretical level, by nature of distributed environment, there are many aspects that are different from single process computing, such as asynchrony and synchrony computing, limitation of data sharing, and network reliability, etc. At the implementation level, there is an underlining system that the developer has to take into account beyond a distributed algorithm, such as network communication and especially network protocols. Moreover, there is also the configuration of a distributed environment that is not easy to duplicate from a real life environment into a testing environment such as the size of network, data transmission reliability, and

Table 2.1: Comparison of some features and aspects of existing simulation systems with the proposed DisJ Simulation System

	DAJ	T-DAJ
Programming language	Java	Java
Distributed models	Message Passing	Message Passing
Network environment	fixed wired	fixed wired
Environment configuration	n/a	n/a
Topology	complete network only	any
Topology creating	coding	coding
Communication type	bi-directional	bi-directional
Simulation playback	step by step	yes
Replay	yes	n/a
Stats report	n/a	n/a
Adversary support	n/a	code assertion

activation/deactivation of entities within a network, etc.

Therefore, a simulation for distributed algorithms plays an important role for a developer to overcome such limitations and difficulties in order to develop distributed algorithms more effectively. The simulation not only provides an environment for the developer to develop, test, and verify the algorithm, but also helps the developer to understand behaviours and characteristics of the algorithm.

This thesis focuses on reactive distributed computing. We researched several interesting simulators for reactive distributed algorithms that have been developed in the past, and briefly discuss their advantages and disadvantages to the best of our knowledge. Table 2.1, Table 2.2, and Table 2.3, are a brief comparison of some features and aspects from existing simulation systems that we will discuss in later

Table 2.2: cont.. Comparison of some features and aspects of existing simulation systems with the proposed DisJ Simulation System

	SinAlgo	DisASter
Programming language	Java	Java
Distributed models	Message Passing	
Network environment	wired and mobile wireless	fixed wired
Environment configuration	coding	n/a
Topology	any	any
Topology creating	import file and coding	GUI
Communication type	bi-directional	bi-directional
Simulation playback	yes	yes with undo-redo execution
Replay	yes	n/a
Stats report	coding required	n/a
Adversary support	n/a	message queue modification

Table 2.3: cont.. Comparison of some features and aspects of existing simulation systems with the proposed DisJ Simulation System

	DAP	DisJ
Programming language	C++	Java
Distributed models	Message Passing	Message Passing, AW, AT
Network environment	wired and fixed wireless	fixed wired
Environment configuration	basic scenario control	GUI
Topology	any	any
Topology creating	GUI and import XML	GUI and import file
Communication type	bi-directional	uni-directional and bi-directional
Simulation playback	yes	yes
Replay	n/a	yes
Stats report	yes	yes
Adversary support	n/a	yes

sections.

2.2 DAJ - Distributed Algorithms in Java

2.2.1 Introduction

DAJ is an interactive visual aid simulator framework designed by Mordechai Ben-Ari of Weizmann Institute of Science in Israel, in 1997 and updated in 2006. The main goal of the framework is to help the user understand various types of reactive distributed algorithms in a Message Passing model. The simulator interacts with the user in a step-by-step manner during an execution of the algorithm in discrete state changes. The main purpose of the framework is for the user, particularly a student to study, observe, and understand behaviors and characteristics of algorithms [5] [6].

2.2.2 Environment

DAJ is a simulator framework written in Java that enables the user to write reactive distributed algorithms in Java. In this framework, the user has to implement both the algorithm and the simulator as a single code set. This means that each simulator simulates only one specific algorithm and cannot be reused for simulating another algorithm. The framework offers a simulation environment: homogenous entities, only complete network topology, only total reliable bi-directional connectivities, and a very small network with no adversary support.

2.2.3 Visualization

The DAJ framework provides a basic graphical user interface (GUI) in the form of a Java Applet that displays every entity in a simulation network. The framework

Figure 2.1: The DAJ Simulation Control Screen Snapshot [6]

uses a GUI widget Textbox to represent an entity. The Textbox contains the entity information and a set of option buttons, which allow the user to select how to process an algorithm. The number of Textboxes is based on the number of entities in the network and all are shown on screen at the same time in a grid format (this is the reason why the size of the network is very small). The user is required to interact with a simulator in a step-by-step order when processing an algorithm. However, the simulator provides a recording mechanism so that the user can replay the simulation execution over and over without altering the results from the original execution. Figure 2.1 shows a snapshot of the DAJ simulation control during a simulation of an algorithm.

2.2.4 Limitations

The DAJ framework has a number of limitations related to the simulation implementation and execution. The user must implement the framework presentation and the

algorithm in a single pack of codes. The framework supports only a Message Passing communication model in an asynchronous environment. A network topology can only be a complete network structure connected with bi-directional links with total reliability. The size of the simulation network is very small and requires user interaction with an entity in every execution (e.g. sending a message). As a result, the complex algorithm and the large amount of interactions prohibited the user from using the system because the number of executions is too large. Furthermore, the framework has no adversary support and neither provides algorithm execution statistics or information.

2.3 T-DAJ - Toolkit for Distributed Algorithms in Java

2.3.1 Introduction

T-DAJ is a Toolkit for Distributed Algorithms¹, which is written in Java, and was designed by Wolfgang Schreiner of Hagenberg University of Applied Sciences in Austria in 1997. The toolkit provides a class library with a programming interface to use in programming distributed algorithms and visualizing their dynamic behaviour [71]. T-DAJ allows users to develop distributed algorithms with a Message Passing communication model and execute it in three different modes, standalone with visualization, standalone without visualization, and Java Applet embedded into a webpage.

¹The original name of T-DAJ Toolkit is DAJ, due to the confusion with another system with the same goals, we give a temporary name in this thesis for the toolkit

2.3.2 Environment

The T-DAJ toolkit provides a basic communication API library for a Message Passing model to be used in an algorithm. The implementation of a simulation in T-DAJ is very straightforward: using their provided Java classes. First, a network topology class - this Java class allows the user to define a network topology using bi-directional links. Second, a protocol class - this Java class allows the user to define and code the distributed algorithm to be simulated. Here the user explicitly codes the behaviour of each entity and iterates through every entity in the network. Therefore, the environment can be non-homogenous because entities can execute different algorithms. Finally, a message content class - this Class defines the content types of message that will be used by an algorithm for communication. T-DAJ also provides a verification mechanism at the coding level with assertion conditions to validate variable values, and boolean outcomes within protocol codes. This means that the protocol codes are mixed with both algorithm and verification, instead of separate algorithm and verification parts. This validation mechanism validates correctness of coding, not correctness of the algorithm.

2.3.3 Visualization

T-DAJ provides a Java class for the user to create network topology. The toolkit provides a network topology view that displays current state of entities in the network such as colouring on a link based on transmitting activities, displaying the accumulated number of messages a node received, and colouring a basic predefined state of a node. T-DAJ predefined three basic states of a node: Ready, Blocked, and Terminated. A Ready state is denoted by a green colour where a node is ready for execution. A Blocked state is denoted by a red colour where a node is blocked due

to various reasons (i.e., sleeping, waiting for messages from other incoming links). A Terminated state is denoted by a blue colour where a node has terminated from executing a protocol. T-DAJ also predefined three colours to represent a link transmitting activities: a gray colour represents that a link currently does not contain any message, a green colour represents that a link currently contains at least one message, and a red colour represents that a link currently does not contain any message but a node is expecting a message. T-DAJ provides a basic playback for the user to control the simulation that includes Run, Walk, Step, Reset, Interrupt, Quit, and Redraw. Run function allows the user to start the simulation, Walk function allows the simulation to execute in slow mode, Step function allows the simulation to perform one execution, Reset function reinitializes the visualization of the simulation, Interrupt function allows the user to pause the simulation, Quit function terminates the simulation, and Redraw function allows the simulation to update the visualization. Figure 2.2 shows a snapshot of the T-DAJ during the simulation of a protocol in a Java Applet mode.

2.3.4 Limitations

The T-DAJ toolkit has some limitations with respect to coding, execution, and replay of an algorithm. A user must code a topology and a distributed algorithm in a single package that cannot be separated. The toolkit supports only a Message Passing communication model and the execution of the algorithm cannot be replayed. A network structure is limited to bi-directional links with total reliability. Furthermore, the toolkit has no adversary support and does not provide algorithm execution statistics and information.

Figure 2.2: The T-DAJ Screen Snapshot form Java Applet [72]

2.4 DAP - Distributed Algorithms Platform by University of Patras, Greece

2.4.1 Introduction

DAP is a platform under development for simulating and developing distributed algorithms in a homogenous environment for wired and wireless networks being designed by the Computer Technology Institute of the University of Patras in Greece from 2000 to the present. DAP is a LEDA extension package written in C++ that provides a topology editor to define network structures for simulation, a library for Message Passing communication model, and statistical reports from simulation results [16]. The DAP platform itself can be physically distributed by executing the simulation in one machine and observing results in another machine.

2.4.2 Environment

DAP is a Linux based simulator that allows a user to develop distributed algorithms in C++ for a wire and wireless network environment [49]. DAP supports Message Passing communication model and provides a communication API library instead of requiring the user to implement a network communication directly. DAP provides a graph editor that allows the user to create a topology of the network in a graphical way, and every entity in a network is homogenous. However, DAP does not provide an API for accessing local entity information during runtime such as a list of available ports, the number of connected neighbours etc. Instead the user has to store the information for entity in the code. DAP also provides a scenario control as a separate component that allows the user to schedule network environment status and event(s) to access at specific times. The scenario events include entity initiation, entity failure, entity recover, connection lost, connection recover etc. This can be used to verify algorithm behaviour directly, and the scenario can be saved in persistent storage (e.g. an I/O file) and reused later. An interesting thing about the DAP is that the simulation engine and the graphic user interface (GUI) are separated modules based on XML protocol for communication. Figure 2.3 shows an architecture of GUI module in the DAP.

2.4.3 Visualization

DAP uses LEDA libraries [57] to create a detailed visualization of topology with varieties of structures, colors, and labels that can be imported and exported. DAP also provides a simulation playback function to control a simulation execution from GUI and simultaneously display execution data visually [49]. Moreover, the user can observe and modify topology and entity information via GUI while the simulation

Figure 2.3: The DAP GUI Architecture [16]

is executing, and also provides a logging service that logs execution information in XML format and saves it in a persistent storage so the user can study and analyze information offline.

2.4.4 Limitations

The DAP simulation supports only a Message Passing communication model in an asynchronous environment. A network structure is always connected by bi-directional links, message contents can only be a character string type, and entity properties can contain only integer, double or character string types. Furthermore, the platform has no adversary support and does not provide an execution replay.

2.5 SinAlgo - Simulation of Network Algorithm

2.5.1 Introduction

SinAlgo is a network simulation framework that supports Message Passing model for simulating network algorithms in wired and wireless networks. It was designed

and developed by the Distributed Computing Group of the Swiss Federal Institution of Technology in Zurich, Switzerland in 2007. The framework has been developed using Java programming language with an objective to assist a user to prototype and simulate network algorithms. The simulation is a standalone application that provides an extensive view of network topology for a large network in 2D and 3D graphics with acceptable performance [40].

2.5.2 Environment

SinAlgo is a Java-based network simulation framework for homogenous entities in asynchronous and synchronous network environments. The framework supports wired and mobile wireless network environments that allow the user to configure network environments such as connectivity of entity in the network, distribution of entity in the network, reliability of network connection, interference of network connection, and entity mobility to suit their needs. The SinAlgo provides a basic Message Passing communication API for the entities to communicate with each other. SinAlgo also allows the user to customize the entity in order to fit their needs by extending and implementing existing Java codes.

2.5.3 Visualization

The SinAlgo framework allows the user to create topologies with visualization that support a large network in 2D and 3D on a single desktop computer. The framework simulation model requires the user to create entities via a configuration file, implement behaviour of the entity by extending a Node class, and specify a network structure (connectivity) by implementing a connectivity model. The framework provides a playback function that allows the user to control an execution of the simulation. The

Figure 2.4: The SinAlgo Screen Snapshot [40]

user is also able to modify the network structure and environment dynamically during runtime. The framework allows the user to save a simulation execution into an I/O file system for replay and review offline at a later time. Moreover, the framework allows the user to export a topology view to a vector graphic in ESO or PDF format for presentation and publishing. Figure 2.4 is a snapshot of SinAlgo simulating an algorithm simulating a network protocol in 2D graphic viewer.

2.5.4 Limitations

SinAlgo has some limitations for use as a distributed simulation. SinAlgo supports only a Message Passing communication model in a network that is connected with bi-directional links. The creation and configuration of a topology can only be set up in a configuration text file without support of GUI and visualization for editing or creating because in general the user understands the network topology more easily

when it has been represented in visual form. The network environment parameters can be adjusted in Java codes by implementing specific Java classes, which means the user is required to code network environment parameters in Java class in the same way as coding a distributed algorithm instead of using GUI to define the parameters. Therefore, a topology and algorithm have a tight relationship due to the nature of object interactions during the execution, which makes it very difficult to reuse and separate the network topology from the algorithm. Furthermore, the framework has no adversary support and the user is required to write extra Java code explicitly in order to obtain algorithm execution statistics and information.

2.6 DisASter - Distributed Algorithm Simulation Terrain

2.6.1 Introduction

DisASter is a platform for developing distributed algorithms with a Message Passing communication model in Java. The platform was designed and developed by Rainer Oechsle and Tim Gottwald of the Computer Science Department, University of Applied Sciences in Trier, Germany in 2005. The platform is Java based and provides a distributed computing library in Java for a distributed computing programming environment. The platform provides a playback function that allows the user to interact with the simulation during runtime, and observe the simulation through a built-in visualization views. The platform also supports future implementation of application-specific view in order to observe the simulation that the platform currently does not provide [62].

2.6.2 Environment

DisASter is a Java-based simulation platform for Message Passing communication model, where the implementation of the platform is similar to the programming model of Servlets or EJBs [58]. Hence the user has to extend and implement certain classes and methods, and make use of the provided Message Passing API libraries for inter-communication between entities in a network. Communication messages that are used in the platform are serialized and support only a bi-directional transmission link. The user is able to observe and modify these messages in a global message queue. However, each message type must be implemented in a separate Java class. The platform also allows the user to assign different algorithms to different entities in a network topology for different roles in a distributed protocol. As a result, entities may not have the same algorithm to simulate. Moreover, an entity has only two options when sending a message: sending a message to a single connected entity or broadcasting a message to all connected entities. The platform does not provide a mechanism for the user to configure the network environment for the simulation.

2.6.3 Visualization

The DisASter platform provides a topology editor that allows the user to create topologies for a simulation, which can be visually observed algorithm behaviours during the simulation. The platform provides Ring and Torus topologies as ready-made topologies for the convenience of the user. The platform consists of three views of the execution during runtime: Network View, which displays a network structure and states, Sequence View, which displays sequence diagram of message interaction between entities, and Message Queue view, which displays a global message queue of all current messages that are in transmission. The platform also provides a playback

sent	Sender	Receiver	Message	Messagetype
0		MutEx-Agent 2		InitMessage
1	MutEx-Agent 0	MutEx-Agent 3	10	Request
1	MutEx-Agent 0	MutEx-Agent 1	10	Request
1	MutEx-Agent 0	MutEx-Agent 2	10	Request

Figure 2.5: The DisASTER Global Message Queue View Screen Snapshot [62]

function that allows the user to control the simulation during runtime that includes the ability to step an execution forward or backward. This capability allows the user to undo and redo the executions manually and discretely. Moreover, the platform also allows the user to develop new visualizations for the simulation by developing an extension from existing codes. Figure 2.5 shows a snapshot of Global Message Queue view during the simulation of an algorithm.

2.6.4 Limitations

The DisASTER platform supports only a Message Passing communication model and a network structure is always connected by bi-directional links. An entity can only send messages to a single connected target or to all connected targets. The platform does not provide an adversary support except by allowing the user to manually modify a global Message Queue. Furthermore, the platform has no execution replay support nor does provide simulation execution statistics and information.

Chapter 3

Distributed Environments and Models

This chapter discusses reactive distributed computing environments and related communication models. The first section discusses the environments of reactive distributed computing, which includes definitions and terminologies. The second section discusses different types of reactive distributed communication models that have been used in the Netscape environment. From this point forward the term reactive distributed computing is interchangeable with the term distributed computing.

3.1 Distributed Computing Environments

A reactive distributed computing environment consists of a finite number of entities that form a connected network to perform computations. These entities collaborate with each other in order to achieve a common goal. The details of the distributed computing environment are discussed in the following subsections.

3.1.1 Entities

An entity is a core computational unit of the environment. Physically, the entity could be a representation of a computer, a process, a mobile device, or a software

agent, etc. The entity has the following capabilities i) processing power ii) local memory storage iii) ability to communicate with other entities iv) ability to change its status (state) and v) ability to (re)set its alarm clock [69]. The behaviour of the entity is affected by external events (e.g. spontaneous impulse, an internal alarm clock ring, and receipt of external information). Each external event drives the entity to perform one or more actions. Action is defined by the type of the external event and the current status of the entity: $Status \times Event \rightarrow Action$. Therefore, the entity reacts if-and-only-if an event occurs to the entity. The entity in the model has tree properties: i) Homogenous - which means entities in the environment perform the same action if they are in the same status and receive the same event, ii) Anonymous - which means every entity is physically indistinguishable unless it has been stated explicitly in the assumption of the environment e.g. the entity has a unique ID; and iii) Autonomous - which means there is no central role assigned to a specific entity to control other entities such as master-slave roles.

3.1.2 Network Environments

The network is a structure of data processing nodes that are interconnected. There are two common types of network environments: the Netscape environment and the Mobile network environment.

The Netscape environment is a set of immobility nodes that are connected together with communication links. For example, the Internet, any computing grid, any computer network, etc., [82] [81]. The Netscape environment can be formed by different types of devices such as a server machine, computer desktop, laptop, mobile devices etc., as shown in Figure 3.1. These devices are the computing nodes in the environment that have processing power and resources. Each node may also contain

Figure 3.1: An Example of Different Devices form the Netscape Environment

Figure 3.2: The Mobile Network Environment Deployed in Planar with Obstacles

a software agent that has the capability of moving between nodes within the network. The Netscape environment can be fixed or dynamic. The fixed environment is where the connection of network structure is fixed throughout the lifetime of the task. The dynamic environment is where the connection of the network structure can be changed dynamically during the lifetime of the task. In the Netscape environment, the entity performs a computation that can be either a node or a software agent. The Netscape environment is the most common network environment that has been used and studied, not only within the academic communities but also in industry for

applications such as the Internet, corporate networks, and data grid etc.

The mobile network environment contains a set of mobile nodes that are connected together via wireless connections on a planar landscape [65]. The mobile nodes have the capability to move on the landscape, seeing and/or communicating with other entities, Figure 3.2. The mobile network environment usually contains some obstacle objects that create difficulties for the nodes to perform tasks. For example, in Figure 3.2, the mobile nodes (represented by circles) were deployed in a building where the walls (represented by shade pattern) act as obstacles that limit the mobility, visibility, and communication of the nodes in the mobile network environment. Mobile networks generally have a large number of nodes, and the nodes have more specific tasks assigned based on the nature of the application deployed. Examples are forest fire detectors, mine sweepers, weather forecasters, etc., [54] [42]. In general, the nodes are deployed to cover a large area in a physically challenging terrain that puts the nodes at risk of not being able to function. Therefore, the nodes in the mobile network should be pervasive. Because of the large network size required, the nodes should be inexpensive in the sense of having limited computing powers and resources (processor power, memory size, power supply, etc.) [70] [83] [84]. In general, the nodes in the Netscape environment have more computing power and resources than the nodes in the mobile network environment. Another difference between the mobile network environment and the Netscape environment is that the structure of the mobile network environment is a dynamic structure, because the nodes can change location within the environment, and the limited range of the wireless communication that affects the connectivities structure [53] [52]. This thesis does not cover the sensor network environment and we leave it for future studies.

3.1.3 Communications

A communication is an activity of conveying information. In the reactive distributed computing environment, the entity has no direct access to other entities information. Therefore, an entity has to communicate with other entities in order to obtain information required for computation and collaboration in order to achieve a goal. The communications have important properties that one should take into account: communication time delays, communication reliability, and orientation of the entity in regards to communication.

Time Delays: A point-to-point communication involves many activities such as preparation, transmission, and reception, etc. The times that are required for these activities are unpredictable. For example, in the real world of data transmissions via a cable, the transmission speed depends on the medium of the transmission. If the medium is a copper cable, the speed of the transmission is slower than a fiber optic cable [35]. In the distributed computing environment, it is required that the communication delay be bounded and finite, which means that the information will eventually arrive at its destination in a finite time, otherwise it has failed. It is important to note that the delay of the information may affect the correctness and performance of an algorithm. Some algorithms can be greatly affected by communication delay. For example, in a spanning tree construction algorithm, whether the algorithm will be able to construct a balanced tree or not, is heavily based on the order that the information arrives at the entities from different senders.

Communication Reliability: Communication processes in any network are somewhat unreliable which means that information may fail to arrive at the destination. There are many causes that affect the communication process such as the quality of the mediums used in transmission, environmental problems, and problems with the

software that manages the communication, etc., [74] [4] [76]. In distributed computing environments, communication reliability is the most important property because every algorithm requires a certain amount of communication reliability in order to perform a task. The communication is the major event that triggers the entity to perform actions as well as the only means to access information from other entities. Therefore, in a distributed computing environment, the communication within the network cannot be totally disconnected.

Entity Orientation: The orientation is the placement of an object in a rotational coordinate system with respect to a fixed point and a reference position. The orientation of the entity is another important property in distributed computing environments. There are two levels of orientation, the first is global orientation and the second is local orientation. The global orientation is defined by and used by every entity in the network. For example, in the Sensor network environment, if every entity uses a magnetic compass, the direction of north, south, east, and west are the same for every entity. In comparison, the local orientation is an orientation that is defined and used internally by the entity. For example, in the Sensor network environment, if the entity defines the north orientation based on the direction that it is facing when it is initiated, then the east is 90 degrees to the right and so forth. Therefore, the north of one entity might be the east of another entity if they were facing in different directions when they were initiated. The orientation commonly helps in reducing the complexity of the algorithm in the distributed computing environments, especially when the entities have a global orientation [25] [26] [24]. For example, in a traversing message algorithm in a ring topology with bi-directional links in the Netscape environment, the algorithm wants a message to visit every entity in the ring and return to the entity that initiates the message. In the global orientation of a left-right port, where an entity left port connects to another right port, the algorithm is very simple

and optimal, because the entity only sends the message to the right port and the message will eventually return to the initiated entity from the left port. In comparison to the local orientation, where an entity left port might connect to another entity left port, the above algorithm that was used with the global orientation cannot be used in this case.

3.1.4 Knowledge

Knowledge in a computing system is information acquired through computing or pre-configuration of the entity system. There are four levels of knowledge in the distributed computing environments: i) local knowledge, where the knowledge is only known to the entity in the environment at a given time, ii) implicit knowledge, where the knowledge is known by more than one entity in the environment at a given time, iii) explicit knowledge, where the knowledge is known to every entity in the environment at a given time, and iv) common knowledge, where at a given time the knowledge is known to every other entity in the environment, and every entity knows that every other entity in the environment has the knowledge, and every entity knows that every entity knows that every entity has the knowledge, and so on. There are three types of knowledge in relation to the entity [69] : i) metric information, which is the knowledge of numeric information regarding the network, for example, the number of entities, the number of links, the diameter of the network, etc., ii) information about topological properties e.g. the topology type (Ring, Tree, Hypercube, etc.), the communication type (bi-directional link or uni-directional link), and the transmission order (FIFO, LIFO, and random), etc., and iii) topological map information, which is knowledge of a topology and its connection structure, for example, a map of a topology with labels, a map of entity neighbours up to a specific distance with labels [69] [9] [44].

3.1.5 Cost and Complexity

In distributed computing algorithms there are three types of complexities: time complexity, communication complexity, and distance complexity. The time complexity is the time that an algorithm requires to complete the task, which is a function of the size of the environment. The communication complexity is the amount of communication activity that the algorithm requires to complete the task, which is a function of the size of the environment. The distance complexity is the amount of distance that the algorithm requires to complete the task, which is also a function of the size of the environment. The communication complexity is one of the most important issues in cost analysis, since the amount of communication activity defines the actual time and expense of resources required by the algorithm. Therefore, the thesis is mainly focused on the communication complexity because the time complexity is commonly known and used in any algorithm analysis in Computer Science studies, and the distance complexity is used only in a Sensor network environment (not covered in this thesis). For a simpler discussion and analysis of the communication complexity, we abstract the communication activities by merging detailed activities, such as preparing information, transmitting information, and receiving information into a single activity, called sent/transmitted information. Hence, the communication complexity can be determined by counting the number of messages that are sent by the entity or the number of transmissions that an agent transmitted during the execution of the algorithm [69] [36] [8].

Figure 3.3: The Distributed Communication Models in the Netscape Environment

3.2 The Communication Models in the Netscape Environment

In the Netscape environment there are three common communication models: the Message Passing model, the Mobile Agent with Token model and the Mobile Agent with Whiteboard model as shown in Figure 3.3. The details of the models are discussed briefly in the following subsections.

3.2.1 The Message Passing Model

The Message Passing model is the most common model that has been used in reactive distributed computing environments [1] [45]. A *node* is a unit that has processing power and resources such as a computer, a telephone switch, or a communication tower, etc., and is able to send/receive messages. In this model, a node is a main computing entity that uses messages as a means of communication between entities so that they can collaborate and perform a task in order to achieve a goal. A *message* is a finite sequence of bits that contains information that is understandable by an entity in the network. The entity can transmit messages to any other connected entity

via a communication link via wired or wireless connections. A combination of the message and the current status of the entity drives actions; $STATUS \times EVENT \rightarrow ACTION$, as discussed earlier in this chapter. There are many common and widely use protocols based on this model, such as information broadcasting, leader election, routing, and shortest path, etc.

3.2.2 The Mobile Agent with Whiteboard Model

The Mobile Agent with Whiteboard model uses agents as a main computing entity, and nodes which act as hosts that may contain one or more agents. Agents can move from node to node within a connected network. An *agent* is a piece of software that acts as a computing entity and has its own resources, such as state, memory, internal alarm clock, and protocols. The agent can perform action if-and-only-if it resides within a node. An agent may use the processing power and resources of a node when it resides within the node. Also, it has the ability to access the information stored in the node that it currently resides in. Every agent needs to have a host node, called home base, where the agent resides at the beginning of a computation. A *node* is similar to the node that was described in the Message Passing model (section 3.2.1), with the exception that a node in this model is passive and does not communicate with other nodes in order to achieve a goal. A node in this model contains a Whiteboard for the agent to use as a means to communicate with others agents. A *Whiteboard* is a bounded memory holder that each node has. It is visible and accessible to any agent that resides within the node. However, only one agent at a time may access the Whiteboard (mutually exclusive access), and the access is granted by the node in a fair manner [18] [13]. An agent can write to the Whiteboard in order to leave messages for other agents, and an agent can read from the Whiteboard in order to retrieve messages left by other agents. An agent is also able to access the internal

memory of the nodes for computing purposes. The obtaining of information from a Whiteboard is considered to be an event under the reactive distributed computing model that drives the action of the entity; $STATUS \times EVENT \rightarrow ACTION$. This model is very useful for large networks, which are physically remote, and difficult to access by humans. This model allows the user to access certain location(s) and deploy the agent(s) to move throughout the network and performs task(s) at different locations on behalf of the user. There are common protocols that use this model such as map construction, dangerous network search, and contamination problems etc.

3.2.3 The Mobile Agent with Token Model

The Mobile Agent with Token model is a similar model to the Mobile Agent with the Whiteboard model where a node is not a computing entity. Instead a node acts as a host that may contain one or more agents that can move from node to node within a connected network. An *agent* is a main computing entity as described in the Mobile Agent with the Whiteboard model (section 3.2.2) except that the agent in this model can carry a certain number of Tokens and use them as a means to communicate with other agents instead of using the Whiteboard. A *Token* is an indistinguishable bit of memory that the agent can carry and is able to drop/pickup (set/reset) inside a node (memory) where it resides. A *node* in this model is also the same as the node in the Mobile Agent with the Whiteboard model except that it provides locations (memory) where the Token can be dropped off and/or picked up, e.g. centre of the node and incoming and/or outgoing ports of the node. The visibility of the Token can be detected by the agent if-and-only-if the agent resides within the node that has the token. The node controls the access to the token by the agents in a fair and mutually exclusive manner. Accessing the Token(s) and the node information is considered an event under the reactive distributed computing model that drives the

action of the entity; $STATUS \times EVENT \rightarrow ACTION$. This model is very useful for large networks which are physically remote, and difficult to access by humans. This model allows the user to access certain location(s) and deploy the agent(s) to move throughout the network and perform task(s) at different locations on behalf of the user. Moreover, this model requires less computing resources than the Mobile Agent with Whiteboard model because the memory required for the Token is typically much less than the Whiteboard [23] [14]. Examples of common protocols that use this model are resource access control, dangerous search, and network search, etc.

Chapter 4

Simulation System Requirements

This chapter discusses the requirements for a simulation system that provides a full lifecycle of reactive distributed algorithm development, from writing a programming code to generating a report after the simulation has ended. The objective of this simulation is to help the user to develop and test reactive distributed algorithms more efficiently. The simulation provides various functions and features that allow the user to be more productive in developing reactive distributed algorithms, and help the user to quickly and thoroughly understand the behaviour and characteristics of the algorithms that the user is developing. This chapter is divided into four sections. The first section briefly discusses the overall requirements that the simulation system should have. The remaining sections offer more detailed discussion of each subsystem and its functional requirements.

In the following chapters, we will refer to a "reactive distributed simulation system" simply as "simulation system".

4.1 System Requirements Overview

A discrete event simulation model is a suitable model for simulating reactive distributed algorithms because the events of a reactive distributed algorithm are discrete, the number of states is finite, and at any given point in time, the state of the system can be observed. A discrete event simulation system can be implemented using two different approaches: i) implemented as a single processor simulation system, and ii) implemented as parallel/distributed processor simulation system. A reactive distributed simulation system in this thesis follows a single processor simulation system approach. Further reading about various types of discrete event simulation models can be found in [20] [30] [31] [85] [46].

From an abstract point of view, a simulation system can be described as a system that allows the user to input the information required by the simulation system in order to simulate and produces outputs that are useful for the user to analyze and study the environment that had been simulated. For a reactive distributed simulation system, there are four inputs that are necessary: distributed algorithm protocol, network topology, network environment parameters, and adversary instructions. There are three necessary outputs from the simulation system: updated information of network topology, data logs, and simulation reports, as shown in Figure 4.1.

4.1.1 The Input Requirements

The inputs required by a simulation system are: distributed algorithm protocol, network topology, network environment parameters, and adversary instructions. We discuss now in further details of each input requirement.

Figure 4.1: Overall requirements for a Reactive Distributed Simulation System

Protocol

The protocol is programming code that describes a reactive distributed algorithm. The simulation system is a simulation machine (software) that simulates reactive distributed algorithms. Therefore, a protocol is the most obvious input requirement that the simulation system must have for the simulation. The protocol should be written in a common programming language that is understandable by the simulation system, and can be applied to real world applications without major modification.

Network topology

The network topology describes a structure of the network connections in a distributed computing environment [56]. A distributed algorithm protocol requires a network topology in order to perform an algorithm. To be more precise, a reactive distributed protocol might have been written in such a way that it specifically works on certain network structures, and may not work with other network structures. Therefore, the ability for a simulation system to accept a variety of network topologies is important

because a protocol can be simulated with different network structures. The presentation of the network topology is also very important because it is easier for the user to understand the structure of network when it is in the form of visual graphic presentation, rather than using text description. It is also very time consuming for the user to understand the network structure by reading text descriptions.

Network environment parameters

The network environment parameters define network environment behaviour and characteristics [10] [86]. The network environment has several properties including connection reliability, communication delay, availability of entity, etc. These properties have a definitive impact on the simulation of a distributed algorithm because the behaviour and characteristics of the algorithm also depend on the environment configurations.

Adversary instructions

The adversary instructions are used for verifying correctness and efficiency of reactive distributed algorithms. The user can use the adversary instructions to create expected environments and scenarios of the network while the simulation is executing the algorithm. The ability to create or duplicate environments and scenarios allows the user to be able to verify the algorithm directly instead of just hoping these environments and scenarios occur during the simulation. The outcome of the simulation would provide the user a better understanding of the correctness and efficiency of the algorithm.

4.1.2 The Output Requirements

A simulation system must produce outputs in order to provide feedback and responses to the user. In particular, the simulation system is required to produce necessary outputs including updated information of network topology, data logs, and simulation reports. A discussion of these output requirements and their importance follows.

Updated information of network topology

As discussed earlier, the network topology is required by a distributed algorithm protocol for the simulation. During the simulation, the outcomes of protocol execution may change the network topology information and status. In order to preserve the integrity of the simulation system, the change of network topology information and status needs to be updated and visible to the simulation system, because this information is used by different components of the simulation system including executing protocol, display views, and logger. Moreover, the information describes the behaviour and characteristics of the algorithm, which is very useful to the user.

Data log

The simulation system should provide a logging mechanism to produce a data log as with any other software systems. The data log contains important information that relates to the simulation system, and is stored in persistent storage that can be retrieved at a later time. The data log is very important and useful for the developer (who is developing the simulation system) and also for the user (who is using the simulation system to simulate distributed algorithms). There are two types of data log: system log and simulation log. The system log is a record of the simulation system internal warning and error messages that relate to the simulation system's correctness and performance, which are important for the developer. The simulation log is a

record of simulation processes, protocol executions, and the simulation outcomes. The system log is used by the developer to fix and improve the simulation system. The simulation log is used by the user to learn about the simulated algorithm. The simulation log is also used by the simulation system to produce simulation reports for the user.

Simulation report

A report of the simulation results is another important output because it presents the outcomes of a simulated algorithm to the user. The report contains processed data that it derived from the raw data collected by the simulation system during the simulation (i.e. logger). The report may contain statistical information that relates to the network environment such as the number of entities in the network, the number of links in the network, the number of agents in the network, etc. The statistical information could be related to characteristics of the algorithm such as the number of messages sent, the number of messages lost, the state transitions of an entity during the simulation, etc. The report provides very useful information for the user, who can use it to learn and analyze the simulated algorithm offline.

4.1.3 System Requirements Partition

The simulation system consists of many requirements that can be categorized based on the nature and characteristic of the requirement. The partition of the simulation system requirements helps the designer and developer of the system to have a better understanding of the system. The simulation system is divided into three groups, with each group corresponding to a subsystem of the simulation system (as shown in Figure 4.2): the primary inputs, the runtime execution, and the presentations. The primary inputs group contains the functional requirements that relate to the receiving

Figure 4.2: The Simulation System and its Subsystems

mandatory inputs from the user. The runtime execution group contains the function requirements that relate to the execution of the simulation. The presentation group contains the functional requirements that relates to the presentation of the simulation progress and results to the user. The following sections discuss more thoroughly these groups as subsystems of the simulation system.

4.2 The Primary Inputs Subsystem

This section discusses the functional requirements of the primary inputs subsystem of the simulation system. The primary input subsystem is responsible for receiving mandatory inputs from the user. There are two primary inputs that are imperative to run the simulation system: a protocol and a network topology. A protocol is programming code that describes a distributed algorithm. A network topology is a representation of the network structure and its environment that the protocol will be executed on during the simulation. Therefore, the simulation system should provide a mechanism to help the user to produce these two inputs in an efficient way and

feed them to the simulation system. This subsystem consists of two components: a protocol editor component that allows the user to write a protocol that describes distributed algorithm, and a topology editor component that allows the user to create network topologies that describe network structures and its environment, which will be used by the protocol during the simulation. This section consists of two subsections. The first subsection discusses the protocol editor component, and the second subsection discusses the topology editor component.

4.2.1 Protocol Editor Component

A protocol editor component is an editor tool that allows the user to write programming codes for distributed algorithms in a common programming language. There are many programming languages that can be used to describe distributed algorithms. The protocol editor component should be able to support such programming languages and also provide necessary components required by these languages; for example, the protocol editor component provides a canvas for writing the programming codes, a compiler for validating the programming syntax, necessary API libraries required by the programming language, development tools to facilitate the user while writing a protocol, etc. Therefore, the following five requirements (as shown in Figure 4.3) should be included in the protocol editor component: i) to support common programming languages, ii) to provide an integration development environment, iii) to provide necessary internal libraries required by the programming language, iv) to allow the user to import external libraries as needed, and v) to be able to access network environment information. The detail of these requirements are discussed in the following paragraphs.

Figure 4.3: The Protocol Editor Component Requirements

To support common programming languages

The protocol editor component should provide a text editor for the user to write and develop the reactive distributed algorithms in common programming languages with which the user is already familiar (since any user who is developing reactive distributed algorithms already has a fairly complex understanding of how to write some programming languages). Some programming languages that are commonly used in programming communities include Java [21], C/C++ [64], Python [89], etc. Each programming language has its own syntax, libraries, and programming structures that determine how the language is written and developed. The advantages and disadvantages among different programming languages are beyond the scope of this thesis.

To provide an integration development environment

The protocol editor should provide an integration development environment (IDE) for the user. An IDE is a standard programming editor tool used in software development

communities to help develop protocols more efficiently. The user could write the protocol in a simple text editor such as Notepad, but it would be very difficult if the protocol was large and complex. The user has limited capacity to memorize the programming syntax, structure, library functions, and the toughest challenge of all: tracking and organizing the logical flow of the protocol. Therefore, an IDE provides mechanisms to help the user to cope with these difficulties while developing the protocols. For example, an IDE helps the user to identify programming syntax and structure errors, to identify runtime errors, to assist with a list of API that are valid for usage.

To provide necessary internal API libraries

The protocol editor component should provide internal API libraries that support programming in distributed environments, since the development of reactive distributed protocol is different than development of any other single process protocol and because the distributed environment has many constraints in programming such as memory access, data sharing, and asynchronous execution. Therefore, the editor component should provide an APIs (Abstract Programming Interface) library that supports the reactive distributed computing environment for the user while developing the algorithms. The APIs library is a set of programming interface functions that support programming in distributed computing environment. The APIs allows the user to utilize implemented functions in the protocol. For example, communication APIs library consists of a set of communication functions that an entity uses for communication purposes, and data access APIs library is a set of data access functions that an entity uses for accessing information. These APIs libraries are required specifically for particular reactive distributed communication models and programming languages. In general, the protocol editor component does not require access to any library while

writing a distributed algorithm protocol until the protocol is compiled and executed. However, since the protocol editor component supports the IDE features, the protocol editor component should be able to access these libraries in order to supply the information to the IDE features such as code assist, dynamic compilation, and debugging.

To allow importing external API libraries

The protocol editor component should allow the user to import or use external API libraries that the user prefer to use for developing the reactive distributed algorithms, since the user may want to use the libraries that differ from the libraries provided by the simulation system.

To allow accessing network environment information

The protocol editor component should provide a mechanism for the distributed algorithm protocols to access the network environment information in order to use it as part of the algorithm; this information could include a list of connecting ports, an identification of a node, the size of the network, etc. Moreover, the information can also be used as "knowledge" of an entity such as the size of the network, the identification of a node, port labelling, etc., which can be used in the protocols.

These requirements offer powerful means to improve the user efficiency in developing the distributed algorithm protocols. It allows the user to apply the distributed algorithm to a protocol quickly without learning a new programming language, and the support of the IDE features takes care of tedious tasks of programming. The user can use provided reactive distributed APIs libraries or import external APIs libraries in the protocols without reimplementing these libraries. The user can also programmatically access information on the network environment that may help in reducing

the complexity of the algorithms, as well as help decouple the dependency between the protocols and network environments.

4.2.2 Topology Editor Component

A topology editor component should provide a graphical editor tool that allows the user to create and modify network topologies graphically. A network topology represents a network connection structure in the Netscape environment that contains node and link elements. A node represents an immobile computer that has a processor and computing power. A link represents a communication link between nodes that can transmit information. The network topology can be presented in two different forms: text or graphical form. Text form is a text description of the network structure and its elements. Graphical form is a graphical presentation of the network structure and its elements. The network topology presented in text form is more difficult for the user to visualize and understand than when it is presented in graphical form. Therefore, the topology editor component should focus on the network topology that is represented in 2D graphical form. The topology editor component should allow the user to create and define network topologies independently from protocols, which means network topologies can be reused with different protocols and in different simulation executions. The network topology should be created in a file format that can be saved in persistent storage (e.g. an I/O file), and it can also be opened, edited, deleted, and transferred to different devices in the same way as any data file. The topology editor component consists of the following three requirements (see in Figure 4.4): i) to provide a canvas to contain visual graphic of network topologies, ii) to be able to save in persistent storage, iii) to allow the user to view and edit network topology information, and iv) to be able to share network topology information with other components of the simulation system. The following paragraphs discuss further

Figure 4.4: The Topology Editor Requirements

details of each requirement of the topology editor component.

To provide a canvas

The editor topology component should provide a canvas for the user to create, view, and modify the network topologies. A canvas is a blank slate on which to display network topologies that have been created or opened. The canvas is also an editing playground, where the user can delete and modify the network topologies graphically in a similar way to many 2D graphic drawing applications. The topology editor component should provide two methods of creating network topologies: creating a network topology from scratch, and creating a network topology from a topology wizard.

To create a network topology from scratch: creating a network topology from

scratch would allow the user to create custom network topologies by using basic network elements to construct: node and link elements. A rectangle symbol represents a node element, and a line represents a link element that connects two nodes together. A link element requires two nodes in order to connect - a source and a destination - but a node can have as many links as required as long as no link of the same type having the same node pair connection as another link. A link element is further dividable into two types: uni-directional link and bi-directional link. Uni-directional link is a one way transmission link from source node to target node, while bi-directional link is a two-way transmission link where both nodes can transmit and receive information. From these two elements (node and link elements), the user can construct any network topology by connecting nodes and links together, and the size of network can be small or large as dictated by the user requirements, computing power, and hardware of a computer that runs the simulation system. A network topology can be saved, and the user can open and modify it at a later time.

To create a network topology from topology wizard: the topology editor component should provide a wizard for creating common use network topologies in Netscape environments that the user can select from, including Ring, Tree, Mesh, Hypercube, Spatial, Torus, and Complete topological structures. In order to create a network topology with the topology wizard, the user must select a topology structure and provide some basic required inputs; for example, creating a Ring topology requires the user to input the number of nodes, creating a Mesh topology requires the number of rows and the number of columns, creating a Hypercube requires number of dimensions, etc. The user also has to specify the type of link that connects the nodes in the topology. The user may modify the topology that is created by the topology wizard

in order to meet the requirements of a protocol. The size of the topology is based on the computing power and hardware of the computer that runs the simulation system.

The topology editor component provides basic graphic editing functions that the user can perform on the elements in the canvas such as select, resize, move, undo, redo, and delete. The graphical presentation of the network topology also allows the user to add text information into the network topology and its elements. The information is saved in a file that contains the network topology.

To store in persistent storage

The topology editor component should allow the user to save a network topology that is created into persistent storage and to retrieve it at a later time. In this way, the user can reuse the network topology in different simulations.

To view and edit network information

The topology editor component should provide a container that contains the network topology information that can be viewed, edited, and saved. A network topology presented on a canvas is not only an image of the network structure, but also contains information describing the network topology and its elements. For example, a link element that connects two nodes has properties information such as port label, transmission reliability, connection state, etc. This information should be displayed to the user when the element has been selected on the canvas. The displayed information is modifiable and can be saved for later use. The topology editor component should provide two levels of information: topology level and element level. The *topology level* provides information relating to a network topology and its environment, including a topology name, number of nodes in the topology, number of links in the topology,

number of agents in the topology, etc. The *element level* provides information relating to an element in the network topology (i.e. node and link elements) including element name and elements properties.

To share information with other components

The network topology component should provide a mechanism for other components in the simulation system to access its information during the simulation; for example, sharing network topology information with an executing protocol - the executing protocol instructs a node to broadcast messages to all connected neighbours. The information of the node outgoing ports must be available for the simulation system in order to generate and send messages to the correct target nodes. Another example is when same aspect of the network topology is changed during the simulation, as results from a protocol execution.

4.3 The Runtime Execution Subsystem

This section discusses the requirements of the runtime execution subsystem of the simulation system that is responsible for executing a distributed algorithm protocol and performing various simulation tasks during the simulation. The first subsection discusses an overview of the runtime execution subsystem and presents the components that are required by the subsystem. The remaining subsections discuss each component and its requirements in further detail.

4.3.1 Runtime Execution Overview

The runtime execution subsystem is responsible for performing a simulation of a reactive distributed algorithm protocol with a given network topology. The subsystem

Figure 4.5: The Runtime Execution Subsystem

lifetime extends from the beginning of the simulation to the end. The subsystem has many responsibilities and tasks to perform which include executing a protocol, organizing simulation tasks, producing output data and updating the system information. The subsystem logically consists of five components (see Figure 4.5): i) the simulator engine, ii) the logger, iii) the playback, iv) the adversary control, and v) the replay. The simulator engine component performs internal simulation processes for the simulation system. The logger component is responsible for recording necessary data into persistent storage. The playback component is an interaction function that allows the user to control the simulation process via a graphical user interface (GUI). The adversary control component is a mechanism that allows the user to manipulate the network environment. The replay component allows the user to review a past (saved) simulation from a recorded file. The details of these components are discussed in the following subsections.

4.3.2 Simulator Engine Component

The simulator engine component is responsible for creating and executing the simulation events in the same way as any discrete event simulator engine [20]. The lifecycle of the simulator engine (see Figure 4.6) starts with the simulator engine reading the user inputs and generating an initial set of simulation events; then the loop of actions begins with the execution of an event - updating simulation information - generating zero or more new events until there are no more events to execute.

In the context of the reactive distributed simulation system, the simulator engine lifecycle starts by:

1. the simulator engine initializes some events from the primary inputs
2. the simulator engine puts the initial set of events into an event queue

From this point on, the simulator engine is working continuously until there are no more events left in the event queue:

1. the simulator engine takes an event from the event queue and executes it.
2. the outcome of the execution of an event may lead to a change of states and information of the network topology
3. the outcome of the execution of an event may generate some outputs
4. the outcome of the execution of an event may create zero or more new events
5. if one or more new events are created, they will be put into the event queue

The outcome of the event execution may trigger other components in the simulation system to perform tasks. For examples, the outcome of the execution trigger a logger to record the output information, display views to update presented information based on the outputs, etc. The simulator engine also keeps listening for external

Figure 4.6: The Simulator Engine Lifecycle

instructions or inputs from the user that include playback and adversary instructions. The simulation is ended normally when there are no more events left in the event queue to execute.

4.3.3 Logger Component

The logger component is responsible for recording necessary data into persistent storage [68]. The recorded data can be retrieved and used at a later time by the simulation system, the developer, or the user. The logger component begins recording the data once the simulation is started, and stops when the simulation has ended. The developer of the simulation system defines the types of data or information that the logger component should record. The types of data that are recorded can be categorized into one of the following: a system log and a simulation log.

System log

The system log should contain information relating to the simulation system and its activities from the start to the shutdown of the simulation system. The information generally relates to the state of the simulation system, in which the states of the system need to be tracked in order to make sure that the simulation system works correctly. The developer of the simulation system is required to define a set of states that produce notification messages, and the messages will be logged by the logger component. The notification messages should contain important information that the developer needs to know such as warnings and errors. The content of the messages should be organized in such a way that the developer can easily understand and provide enough information for the developer to find the source of errors or warnings. The notification messages can be categorized into different levels of severity such as critical, high, medium, low, suggestion, etc., so the developer can manage the priority of each message.

Simulation log

The simulation log should contain information relating to the execution of a distributed algorithm protocol and a network topology during the simulation. The simulation system should produce a simulation log per simulation, and each simulation log should contain information relating to a simulation from the beginning to the end. The objective of the simulation log is to record the progress of protocol executions and their outcomes, as well as activities of the network environment during the simulation. The information in the simulation log must be made available for other components or subsystems to access (i.e. replay component, presentation subsystem). Therefore, the simulation log should contain very detailed information relating to the simulation process such as a record of every event that has been executed, state transitions of

every element in the network topology, and environment activities from the start of the simulation to the end.

4.3.4 Playback Component

The playback component is a graphical user interface (GUI) component that allows the user to control simulation process during the simulation. The playback component should provide the playback functions for the user to control the simulation process. The ability to control the simulation process allows the user to better observe the simulation through the visual presentation and the information shown by the simulation system. The visual presentation is discussed in a later section in this chapter. The playback functions include play/resume, pause, and stop functions, which are discussed in the following paragraphs.

Play/Resume

The play/resume function should command the simulation system to start or continue simulating a distributed algorithm protocol. The user can issue the play/resume command via a GUI button and/or menu.

Pause

The pause function should command the simulation system to discontinue any activity and simulation of the protocol until the resume command is issued. This function should not end the simulation; every state of the simulation should still be valid and ready to continue. The user can issue the pause command via the GUI button and/or menu.

Stop

The stop function should command the simulation system to stop simulating. The stop command should execute two steps, one after another. In the first step, the function stops the simulation system from simulating regardless of the progress and state of the simulation. In the second step, the function resets the state of the simulation system to the original starting state. There are two options to issue the stop command: i) using the GUI button and/or menu, ii) shutting down the simulation system. The first option ends the simulation normally and performs the two steps listed earlier, while the second option kill a processor that is running the simulation system directly. The issuing of the Stop command should not affect a saved topology or protocol files in persistent storage, but a simulation log of the stopped simulation may be corrupted or may not have a complete record.

4.3.5 Adversary Control Component

The adversary control component is a mechanism that allows the user to control a network environment for the simulation of distributed algorithm protocols. The adversary control component should work in such a way that it receives network environment control instructions from the user and executes accordingly. For example, the user may input an adversary script into the adversary control component before the simulation starts, and the instructions will be executed during the simulation of the protocol. This method requires the user to code some algorithms in the adversary script in order to detect certain events and conditions of the simulation before applying the instructions to adjust the environment. This method requires three components: adversary script, environment access API, and event registration. The details of these components are discussed as in the following paragraphs.

Adversary script

The adversary script is a set of adversary instructions (programming codes) written by the user in a common programming language. First, the adversary script tries to detect the events or conditions of the simulation that the user is interested in. Once the events or conditions have been found, the script may apply the environment control instructions according to the request of the user.

Environment access API

The environment access API is a set of functions that allows the user to access the network environment information such as a node's name, a message's label, a node's current state, the number of nodes in the network, the number of agent currently alive in the network, etc. The user should be able to use this API in the adversary script in order to observe and detect events and conditions of the simulation.

Event registration

The event registration is a set of simulation events that the user can register to the system in order to receive a notification when the event occurred. The adversary control component should provide a predefined set of simulation event that the user can register via the adversary script. The following are the events that the adversary control component should allow the user to register:

- arrival control
- initiation control
- time control
- lost control

These events must be mapped to the function calls defined in the adversary script. Therefore, the user can register the chosen events by overriding the function calls that map to them. The details of each event and a corresponding function call are discussed in the following paragraphs.

Arrival control

The adversary control component should allow the adversary script to register a function to be called from the simulation system when a message/agent is arriving at a target node.

Within the arrival control function the user may block/unblock a message/agent before it arrives at the target node. The ability to block/unblock a message/agent would allow the user to control the set of messages/agents to arrive at the target node in a particular time or condition. However, it is the user responsibility to make sure all blocked messages/agents are unblocked. The arrival control function should provide three argument parameters: a message/agent that is arriving, a link that transmits the message/agent, and a destination node of the link. An example of the arrival control function declared in Java defined in adversary script is the following:

```
public void arrivalControl(Message msg, Edge link1, Node node1)
```

When msg is arriving at node1 via link1, the function will be executed and the adversary script may block/unblock the arriving msg at node1 on link1. For example, if msg has a message label equal to "RUN" and node1's ID equals to "n1" and link1's ID equals "e5" then block msg. Otherwise, if msg has message label equal to "CATCH" and node1's ID equals "n1" and link1's ID equals "e5", unblock every message. This example means every message that arrives at node "n1" via link "e5" with message label "RUN" will be blocked until a message with label "CATCH" arrives at node "n1" via link "e5" .

Initiate Control

The adversary control component should allow the adversary script to register a function to be called from the simulation system when a node/agent is initializing.

Within the initiate control function the user may dictate when a node/agent should be initialized. A node/agent time of initializing is vital because the number of entities activated at different times may affect the outcome of the algorithm. The initiate control function should provide one argument parameter: a node/agent that needs to be initialized. An example of the initiate control function in Java defined in adversary script is the following:

```
public int initTimeControl(Agent agent1)
```

When agent1 needs to be initialized, the function will be executed and the adversary script may check the conditions of the simulation environment before deciding to initialize agent1. For example, the adversary script will count the number of agents that have been initialized and are alive, and if there are three or more agents alive, delay the initialization of agent1 further (by return a future simulation time), otherwise do it now (by return a current simulation time).

Time Control

The adversary control component should allow the adversary script to register a function to be called from the simulation system when a message/agent is entering a communication link.

Within the time control function the user may define the transmission time of the message/agent in a link. The control of transmission time is very important since the time of arrival of a message/agent at a target node can affect the outcome of the algorithm. The time control function should provide three argument parameters:

a message/agent that is being transmitted, a link that transmits a message/agent, and a destination node of the link. An example of the time control function in Java defined in adversary script is the following:

```
public int arriveTimeControl(Message msg, Edge link1, Node node1)
```

When msg is entering link1 and heading to node1, the function will be executed, and the adversary script may specify an arrival time of msg at the destination node. For example, if msg has message type equal to "T1" and if link1 has ID equal to "E5", then let msg arrive at node1 now (by return a current simulation time), otherwise arrive at a future time (by return a future simulation time).

Drop Control

The adversary control component should allow the adversary script to register a function to be called from the simulator engine when a message/agent has entered a communication link.

Within the drop function call the user may drop a message/agent that entered the link. The drop of a message/agent may impact correctness and efficiency of the algorithm. The drop function should provide three argument parameters: a message/agent that has entered a link, a link that is transmitting the message/agent, and a destination node of the link. An example of the lost control function in Java defined in adversary script is the following:

```
public boolean dropControl(Agent agent1, Edge link1, Node node1)
```

When agent1 is entering link1 and heading to node1, the function will be executed and the adversary script may drop agent1 before it arrives at node1. For example, if agent1 has status equal to "RUN", and node1 currently has two agents, then drop agent1 by returning true, otherwise return false.

Note that the adversary script may break the algorithm even though the algorithm is actually correct, because the script can change the network environment properties in ways that are unexpected by the algorithm. For example, the misuse of the lost control function when the algorithm requires 100 percent transmission reliability in the network environment, but the adversary script dropped some messages on a link, which means the algorithm may not receive all the required messages - and the algorithm fails. However, the adversary control component does not require to have a mechanism to validate the use of the adversary script in such a way that the adversary control component can detect the algorithm rules or assumptions have been broken. Therefore, it is the user responsibility to make sure that the adversary script does not break the algorithm rules and assumptions. In addition, the script is generally fixed to a certain protocol and network topology because the script needs to explicitly specify the information of both protocol and network topology to detect events and conditions before applying specific instructions. Therefore, the script does not have to work on different combinations of protocols and topologies.

4.3.6 Replay Component

The replay component is a mechanism that allows the user to view or replay past simulation of the algorithms. The replay is an animated display of the progress of the past simulations without alternating results and behaviour of the algorithm. The replay works by reading a simulation record and reproduces results sequentially as if the simulation is simulating a protocol. The simulation system stores a replay record in persistent storage (e.g. an I/O file) during the simulation. The replay record can be retrieved at a later time after the simulation is ended. The replay record can be the same as a simulation log or an independent record produced by the simulation system. The replay component starts working by reading a replay record, updates network

topology information, and reproduces visual animation. Moreover, the replay record can be used repeatedly many times since the replay component does not modify the record content. The further reading about replay can be found in [80].

The benefit of replaying past simulation is that it allows the user to repeatedly review the simulation in order to have better understanding of behaviour and characteristics of the algorithm especially when the behaviour and outcomes of the algorithm are inconsistent. For example, an algorithm that is based on transmission speed and reliability has different outcomes based on the condition of transmission when performing a task. Moreover, inconsistencies of behaviour and outcomes of the algorithm are difficult for the user to debug. Simulating or executing the algorithm repeatedly in different rounds would yield different results and outcomes, which makes it very hard for the user to determine the true cause of error(s). In contrast, the replay can reproduce the exact sequence of outcomes of the simulations and it can be executed repeatedly. Thus, it is a lot easier for the user to find the true cause of the failure by using the replay. Also, the replay can be used as a presentation tool for explaining the algorithm behaviour and characteristics to others.

4.4 The Presentations Subsystem

This section discusses the presentation of the simulation feedback and outputs information to the user during the simulation and at the end of the simulation [79] [51]. The presentations subsystem is responsible for presenting information to the user during the simulation until the simulation has ended. The information can be presented in two different periods: i) during the simulation, and ii) when the simulation has ended. The information presented during the simulation is information that relates to the status and progress of the simulation. The nature of information presented

Figure 4.7: The Types of Collected Information

in this period is short and simple. The information presented after the simulation has ended is information that relates to the results of the simulation. The nature of information presented in this period is large and detailed.

The section is organized by the first subsection discusses different types of information that the simulation system should present to the user. The second subsection introduces "Live Presentation" where the information is presented during the simulation. The final subsection introduces another form of presentation, "Final Presentation", where the information is presented to the user when the simulation has ended.

4.4.1 Types of Information

The simulation system needs to collect every information that is required by the presentation subsystem in order to present useful information to the user in an informative and efficient way. There are two types of information that are useful for the presentation subsystem: network element information and network element activities information (see Figure 4.7).

Network Elements Information

Network topology consists of three major elements that are important for the user: node, agent, and link elements. Each element contains information about state and properties that describes the element (e.g. the node element has name, state, ports, hosted agent, etc.) This information may change during the simulation as a result of protocol executions, and the change in this information describes the progress of an algorithm that is simulating. This information can also be used as a tool to verify correctness of the algorithm. For example, the sequence of state transition of a node from the beginning to the end of the simulation for an election protocol where every node is simultaneous awake and is a candidate. The state sequences of every node in a network should either be "AWAKE - CANDIDATE - FOLLOWER - TERMINATED" or "AWAKE - CANDIDATE - LEADER - TERMINATED", otherwise the algorithm may be incorrect. Therefore, network element information is very useful for the simulation system to collect and present to the user at appropriate times.

Network Elements Activities Information

Network element activity is any activity performed by a network element during the simulation, and it is important that the simulation system should present it to the user. There are two common activities that may be performed by a network element: transmission activity and memory access activity. Transmission activity is an activity of transmitting information (i.e. message and agent) in the network such as sending information, receiving information, loss of information, or delay of information. Memory access activity is an activity of a network element accessing information stored in memory storage (i.e. node memory and agent memory) such as obtaining an element status, accessing Whiteboard, or dropping a token. These activities are initiated by the instructions of a simulating protocol and a network

environment. For example, a protocol instructs a node to send messages, a network environment instructs a link to drop a message, etc. This information describes behaviour and characteristics of a simulating protocol in a network environment. The collection of network element activity information during the simulation can describe the efficiency of the algorithm in performing a task (e.g. the number of messages sent by a node, the number of messages that a link has dropped, the number of times an agent accessed a Whiteboard, etc.) Therefore, network element activity information is very useful for the simulation system to collect and present to the user at appropriate times.

4.4.2 Live Presentation

"Live Presentation" is a presentation of information while the simulation is running. The purpose of "Live Presentation" is to report the progress and state of the simulation to the user. The user is required to know the progress and state of the simulation and protocol. In order to keep up with progress and state of the simulation and protocol which is quickly changed and updated, the information presented in "Live Presentation" has to be very informative and understood quickly. Therefore, the content of information has to be short and simple. For example, presenting an updated state of node element in network topology. The network topology contains many nodes that are independently and synchronously executed, which means the state of each node is continuously updated at different times. Thus, the presentation of this information has to be very informative and simple in order for the user to absorb and understand quickly. Examples of presentation formats for "Live Presentation" are a grid of numbers, colour charts, and visual animation (see Figure 4.8).

Another reason that "Live Presentation" is very important is that the user may

Figure 4.8: The Information Presentations Formats

use the information presented in "Live Presentation" to see an overall view of an algorithms behaviour and characteristics and be able to point out strengths and weaknesses of the algorithm without looking into further detailed information from other sources (e.g. log file, statistical report, etc.)

4.4.3 Final Presentation

"Final Presentation" is a presentation of information when the simulation has ended. The purpose of "Final Presentation" is to report results of simulated algorithm to the user. The user is required to know the outcomes of a simulated algorithm in order to evaluate correctness and efficiency of the algorithm. There is a lot of information that the simulation system has to collect and provide to the user in order to be sufficient for evaluating the algorithm. The collected information also has to cover from the start to the end of the simulation, and it has to be organized in proper structures and forms that can be easily retrieved and presented. "Final Presentation" should select useful information and presents it to the user in an efficiency way. The user can use this presented information to analyze and evaluate the algorithm with ease. For example, a statistical report on the communication cost of an election algorithm with

the Message Passing model. The report should present a total number of messages used by the algorithm, and break this down by reporting on each node in the network topology (e.g. node1 sent 100 messages, node2 sent 80 messages, etc.). The report should also break down the information based on node state (e.g. state CANDIDATE requires 2000 message, state ELECTED requires 95 messages, etc.). Therefore, the presentation of "Final Presentation" has to be informative and well organized in order for the user to read and understand easily using method such as text pointed form, data tables, and statistical graphs. (see Figure 4.8). "Final Presentation" does not have to present all the collected information provided by the simulation system. It only needs to present what is useful for the user "Final Presentation" allows the user to specify what information is needed to be presented.

Chapter 5

Architecture and Implementation

This chapter discusses the methodology, architecture, and implementation of the DisJ simulation system based on the requirements discussed in chapter 4. This chapter presents an architecture of the DisJ simulation system and its philosophy. Also it discusses the methodology of the implementation of the DisJ simulation system and its approach.

This chapter is organized as follows: the first section discusses general design problems of a simulation system and presents the main goal of the DisJ simulation system design and architecture. Section 2 discusses the methodology and approach in developing a simulation system in order to meet the goals presented in the first section. Section 3 discusses the design and architecture of the DisJ Simulation System that follows the methodology and approach presented in the second Section, and Section 4 discusses the Object Oriented implementation of the DisJ simulation system in Java.

Table 5.1: The Organization of the Simulation System

Input System	Presentations System	Runtime Execution System
User Protocol	Simulation Outcomes	Event
Network Topology	Logger	Event Queue
Network Environment Parameters	Reports	Time Generator
Adversary Control		Event Processor
		Event Generator

5.1 A Simulation System Design Overview

The best practices in software engineering and system architecture suggest that a large system should be broken down into a set of smaller subsystems in order to reduce complexity of the system [66]. Therefore, a simulation system should be broken down into several smaller subsystems, each consisting of several components with respect to the tasks for which it is responsible. In general, a simulation system of a similar type to the DisJ simulation system consists of three top level subsystems: "Inputs System" that receives data from the user, "Runtime Execution System" that processes the input data and produces raw output, and "Presentations System" that produces appropriate presentations from the raw output. In fact, the partitioning of the simulation system can be done in different ways resulting in different combinations of subsystems and components depending on the decision of the designer. The following discussion presents an example of two different designs in partitioning the simulation system. To make the discussion simple and easy to follow, the example does not include every component of the simulation system. Table 5.1 shows a design of a simulation system that is partitioned into three subsystems, each containing specific sets of components that relate to the subsystem to which they belong.

Table 5.2: Another Possible Organization of the Simulation System

Input System	Presentations System	Runtime Execution System
User Protocol	Simulation Outcomes	Event
Network Topology	Reports	Event Queue
Network environment parameters		Time Generator
Adversary Control		Event Processor
Event Generator		Logger

Let's consider the "Logger" component that is responsible for recording simulation information and error messages. The "Logger" component could be put under "Runtime Execution System" since all simulation information, such as "Event" and simulation results are known to "Runtime Execution System", as shown in Table 5.1. Also, "Event Generator" that is responsible for generating "Events" for "Runtime Execution System" could be part of "Input System" since it generates "Event" that feeds to "Event Queue" as shown in Table 5.2. These are a few of possible designs of a simulation system. A discussion of all of the possibilities of designing a simulation system would be infeasible for this thesis and is out of scope of as well. Further reading about the design of a simulation system can be found at [20] [30] [31] [85].

The design of the simulation system that was chosen for DisJ is shown in Table 5.1 because DisJ groups the components into a subsystem based on the relationship between them. A direct relationship between components exists only within a subsystem, not with other components in other subsystems. This constraint provides modularity to the subsystem. Therefore, the modification of the simulation system can be done easily. For example, if the simulation system was required to change something about the "Event" component, the change may affect other components such as the "Event Generator" component, which has to know how a new "Event"

should be created; otherwise it cannot generate a correct "Event" based on the new requirement. Other subsystems such as "Input System" and "Presentation System" do not need to make any change based on this requirement. An another example, take the "Logger" component: if the simulation system requires a different method of logging data (e.g. logging into database instead of an I/O file) "Runtime Execution System" and "Input System" should not be concerned about the change of logging method into a persistent storage since "Runtime Execution System" is concerned with producing raw data, and "Input System" concerned with receiving inputs from the user; they have nothing to do with storing data into a persistent storage. Thus, the change of logging method from writing I/O file to querying/accessing a database should not affect "Runtime Execution System" or "Input System".

From the discussion above, we realize that the main goal of the design of the DisJ simulation system is to provide the ability of the system to be more flexible in order to be able to adopt new requirements, such as adding new functionality or modifying existing functionality with the least amount of code changes and modifications, and without affecting the reliability and quality of other components of the simulation system.

5.2 Software Development Methodology and Approach

The previous section introduces the main goal of the design of the DisJ simulation system. This section discusses the methodology and approach used in designing the DisJ simulation system in order to meet the design goal and the system requirements discussed in Chapter 4. The design, architecture and implementation of the simulation system are also influenced by non-functional requirements of the system. The DisJ

simulation system also has the following non-functional requirements:

1. it is a proof of concept system or a prototype system
2. it must be a standalone application that can run on a single computer
3. it must be an independent platform that works on a common OS
4. it does not require multiple users to interact with the system
5. it requires the lowest effort of development possible

This thesis introduces non-functional requirements of the DisJ simulation system at this stage because the non-functional requirements of the system are heavily tied to the implementation of the simulation system, not the simulation system itself. The non-functional requirements are not fixed and can be adjusted by the simulation system developer based on available resources and target users, while the ideal of the simulation system is still preserved. Therefore, the thesis does not provide detailed discussion of non-functional requirements and leaves it to the simulation system developer to decide what is best for the simulation system.

Based on the main design goal and non-functional requirements, the combination of Java technologies [37], Eclipse Plug-in development [43], and Object Oriented Methodology(OO) [7] are the approaches that seem to fit best for the implementation of the DisJ simulation system. A brief discussion of reasons and benefits of this combination is the follows:

- *Object Oriented Methodology:* provides a method of developing software that emphasises modularity, adaptability, and reusability [7] [34], which serve the main goal of the design of the DisJ simulation system.

- *Java Technology*: is one of the most common programming language that supports Object Oriented Programming [21] [37], and it is platform independent (it can be run on various operating systems) so it meets the non-functional system requirements presented previously.
- *Eclipse Plug-in Development*: provides a framework that allows the DisJ simulation system to reuse many existing applications and functionalities with less efforts [73] [43] [29], which meets the non-functional system requirements presented previously. Moreover, Eclipse plug-ins are Java based applications that adopt the Object Oriented methodology and are widely used both in the software industry and in academic communities.

Combining these three approaches not only meet the main design goal and non-functional requirements of the simulation system, but also results in increasing the reliability of the simulation system compared to starting everything from scratch, because:

- The Eclipse components have been developed, tested, and used extensively in the software development community, which mean the reliability and stability of the components are well checked.
- Java provides many features and API libraries that handle low level programming tasks such as strong type checking, memory access and allocation, network management, etc., which means it is not only saves development time but also reduce programming errors that can be made by the simulation system developer. Therefore the developer can fully focus on the development of the simulation system instead of programming tedious tasks and checking errors [38].

- Object Oriented methodology is a well-known software development methodology that has been used widely. The methodology was designed to contribute to system reliability [11] [7]. Currently, there are many applications software with various domains, levels of critical, and sizes that have been developed using this methodology.

5.3 DisJ Simulation System Design and Architecture

This section discusses the design and architecture of the DisJ simulation system. The first subsection discusses the workflows of the simulation system and how they influences the design of the DisJ simulation system. Object Oriented methodology logically divided the simulation system into two groups, the loading group and the runtime group. The second subsection discusses the architecture of the loading group and its components. The third subsection discusses the architecture of the runtime group and its components. The final subsection discusses the infrastructure components that are adopted from the Eclipse platform and reused across multiple components in the loading group and the runtime group.

5.3.1 DisJ Simulation System Workflows

In order to develop a simulation system with Object Oriented methodology, one has to understand the workflow of the system so that one can group and/or divide its components and functionalities based on their responsibilities and relationships [60].

A top level view of the DisJ simulation system workflow starts with the user providing some inputs into the system. The system processes those inputs, then

generates some outputs. This top level view of workflow can be further divided into three sub-workflows: "Input Workflow", "Execution Workflow", and "Output Workflow".

- "*Input Workflow*" is a workflow that describes how inputs are fed into the simulation system as shown in Figure 5.3. There are two types of input: i) the inputs that are provided directly into the simulation system by the user such as libraries, configurations, saved topology, etc., and ii) the inputs that are produced by the simulation system and later fed into the simulation system. Use cases of these two types of input are shown in Figure 5.1 and Figure 5.2 respectively. For instance, the user may use the system to create a new topology and protocol, then feed them back to the system to simulate the algorithm (described in the protocol) on the topology.
- "*Execution Workflow*" describes the workflow of the simulation system during a simulation; in other words, it describes how the inputs are processed. For example: the system loads user inputs → the system constructs an internal data structure → the system connects to a random number generator → the system simulates events → the system updates simulation data, etc., until the simulation is ended. Figure 5.5 shows "Execution Workflow" of the simulation system in more detail.
- "*Output Workflow*" describes the workflow of the simulation system presenting outputs to the user in appropriate ways such as logging information into persistent storage, generating visual animations, and presenting reports. Figure 5.4 shows "Output Workflow" of the simulation system in more detail.

Figure 5.1: Use case for the user input data directly into the system

Figure 5.2: Use case for the user that uses the system to produce data and then input the data back into the system

From observing the workflows of the simulation system, we can categorize the activities of the system into two groups based on the time the activities are performed: the loading time and the running time groups. The activity of the system either happens before the simulation starts running (loading time) or happens when the simulation is simulating (runtime). Therefore, we logically organize the components into two groups, as shown in Figure 5.6, based on the time the activities happen. "Loading Group" is a group of components that mainly perform activities before the simulation starts running, and "Runtime Group" is a group of components that mainly perform activities while the simulation is running.

5.3.2 Loading Group Architecture

"Loading Group" is a group of components that perform activities before the simulation starts running. More precisely, it is responsible for loading data from the user

Figure 5.3: The User Inputs Workflow

Figure 5.4: Outputs Workflow

input and other necessary data into an allocated workspace before the simulation starts. A workspace is a space created for storing necessary data required by the simulation in order to simulate the algorithm. The workspace contains important data that derives from the "Loading Group" such as topology data, user protocol, libraries, and configuration parameters. The following paragraphs discuss four different components under "Loading Group".

The Topology Editor Component

The topology editor component allows the user to create, load, view, and modify a network topology that will be used during the simulation as discussed in Section 4.2.2. The topology editor component stores necessary data of the network topology and its environment in a data structure that will be used by the simulation during runtime. The topology editor component consists of many internal subcomponents as shown in Figure 5.8, which are grouped into two parts as following:

- *Interface part* is responsible for providing an interaction interface for the user

Figure 5.5: Execution Workflow

Figure 5.6: The components grouped by the time of activity

and other components in the simulation system to communicate with the component. The interface part consists of three subcomponents: display canvas, interaction GUI, and internal delegate API. i) The display canvas subcomponent allows the user to draw, modify, and view a network topology in 2D graphics that are in a topology file. ii) The interaction GUI subcomponent allows the user to interact with the topology editor component via various types of widgets such as buttons, text fields, combo boxes, etc. The interaction GUI subcomponent provides many commands for the user to issue to the topology editor component in order to access and modify a topology and its elements that includes create, save, delete, etc. iii) The internal delegate API subcomponent is an interface function library that allows other components in the simulation system to access and modify information in the topology editor component programmatically.

- *Core part* is responsible for controlling and managing the topology editor component activities and data. The core part can directly read/write data to/from persistent storage as required by the topology editor component. The core part consists of two subcomponents: controller and internal data structure subcomponents. i) the controller subcomponent is responsible for organizing and managing internal activities of the topology editor component such as synchronizing/updating data, and controlling execution of different processes within the topology editor component, ii) the internal data structure subcomponent is responsible for storing and managing data required by the topology editor component in order to be able to access data in an efficient way.

The topology editor component uses Graphical Editing Framework (GEF) as a foundation of the component. GEF is an Eclipse plug-in that provides infrastructure

Figure 5.7: Topology Editor Class Relationship with GEF

Figure 5.8: The Topology Editor Internal Components

for graphical editing tools, in which the topology editor component adopts and reuses such as a canvas for drawing 2D graphical shapes and a graphic editing interaction GUI. For example to create a canvas with a palette, the topology editor only has to extend a class called "GraphicalEditorWithFlyoutPalette". The class diagram in Figure 5.7 shows how the DisJ simulation system adopts the GEF canvas with palette. Further information about GEF can be found at [3].

The Protocol Editor Component

The protocol editor component is a "tool" for the user to write a protocol, in this case is Java code, that describes a reactive distributed algorithm as discussed in Section 4.2.1. The editor validates protocol correctness by compiling the code to meet standard Java compilation rules. It also makes sure that the protocol has been linked properly to the DisJ internal libraries based on the configuration setup by the user before writing the Java code. The protocol editor component consists of various subcomponents that are shown in Figure 5.9, which are grouped into two parts as follows:

- *Interface part* is responsible for providing an interaction interface that allows the user and other components in the simulation system to communicate with the component. The interface part consists of three subcomponents: display canvas, interaction GUI, and internal delegate API. i) The display canvas subcomponent allows the user to write, edit, and view Java codes that are in a Java file. ii) The interaction GUI subcomponent allows the user to interact with the Java editor component via various types of widget such as menu, button, text field, combo box, etc. The interaction GUI subcomponent provides many commands for the user to issue to the Java editor component in order to access and modify Java codes that includes save, delete, compile, debug, etc. iii) The internal delegate API subcomponent is interface function library that allows other components in the simulation system to access information in the Java editor component programmatically.
- *Core part* is responsible for controlling and managing the Java editor component activities and data. The core part can directly read/write data to/from

persistent storage as required by the Java editor component. The core part consists of five subcomponents: controller, Java compiler, Java runtime libraries, internal data structure, and user libraries adapter. i) The controller subcomponent is responsible for organizing and managing activities within the Java editor component such as synchronizing and preserving integrity of data used in the Java editor component, and controlling executions of different processes within the Java editor component. ii) The Java compiler is responsible for validating Java code written by the user in a Java file, and generating a Java class file for runtime execution. iii) Java runtime libraries are responsible for executing Java class file, runtime memory management, linking data, etc. iv) The internal data structure is responsible for storing and managing data used by the Java editor component. Its task include caching Java code loaded from a Java file, tracking references of Java variables, caching current used API libraries, etc., and v) the user libraries adapter is responsible for connecting and delegating the Java editor component to external libraries provided by the user that are used in the user Java codes during compilation and runtime execution.

The Java editor component uses Java Development Tool (JDT) as a foundation. JDT is an Eclipse plug-in that provides a complete environment for Java development, which the Java editor component inherits and uses in the simulation system without any further implementation or modification. The details of inheriting JDT by the Java editor component are discussed later in this chapter, and further information about JDT can be found at the JDT project website [50].

Figure 5.9: The Protocol Editor Internal Components with Java Implementation

The DisJ Internal Libraries Component

The DisJ system provides libraries to support reactive distributed computing in the Netscape environment with three different communication models: Message Passing, Agent with Whiteboard, and Agent with Token. These model libraries were required and discussed in Section 3.2. The libraries can be accessed by a user protocol and the simulator engine. The protocol can access the libraries by inheriting a particular Java class, which also provides necessary APIs for usage based on a communication model that the protocol implements. The class inheritance structure of these models is shown in Figure 5.10. The libraries provides three different abstract classes for the user to implement in three different communication model: *MessagePassingEntity*, *AgentWithWhiteboard*, and *AgentWithToken*. These three classes implement an interface class called *IDistributedModel*, which contains basic operations that are required by entity in every communication model such as *init*, *getState*, *setState* (*become*), and *alarmRing*. The following are discussions of each class that the user protocol can extend.

- *MessagePassingEntity* class is an abstract class that allows the user to extend and write a reactive distributed algorithm for the Message Passing model in

Java language. This class provides Message Passing operations which the user can use for node entity communication called the `sendTo` operation. The `sendTo` operation provides different types of function arguments that the user may use such as `sendToAll`, `sendToOthers`, and `sendTo` with a given destinations. However, this class requires the user to implement `init` and `receive` functions, which are executed when a node entity is initiated and received messages respectively.

- *AgentWithWhiteboard class* is an abstract class that allows the user to extend and write a reactive distributed algorithm for an agent with the Whiteboard model in Java language. This class provides Whiteboard operations for agent entities to communicate with each other via Whiteboard that locates in a node that an agent currently resides. The operations are include `read`, `append`, `remove`, and `move` operations. i) The `read` operation allows an agent to access and read information contained in Whiteboard, ii) the `append` operation allows an agent to write information to the end of Whiteboard, iii) the `remove` operation allows an agent to delete information from Whiteboard, and iv) the `move` operation allows an agent to travel from one node to another via a connected link. However, this class requires the user to implement `init` and `arrive` functions, which are executed when an agent entity is initiated and arrives at a node respectively.
- *AgentWithToken class* is an abstract class that allows the user to extend and write a reactive distributed algorithm for an agent with the token model in Java language. This class provides token operations for agent entities to communicate with each other via token at a node that an agent currently resides. The operations are include `drop`, `pickup`, `countToken`, `countHostToken`, and `move` operations. i) The `drop` operation allows an agent to drop a token at a node

Figure 5.10: Distributed Communication Model Libraries that DisJ Supports

where it currently resides, ii) the pickup operation allows an agent to pickup a token, if a token exists, at a node where it currently reside, iii) the countToken operation allows an agent to count the number of tokens that it is currently carrying, iv) the countHostToken operation allows an agent to count the number of tokens that exist at a node that the agent currently resides, and v) the move operation allows an agent to travel from one node to another via a connected link. However, this class requires the user to implement init and arrive functions, which are executed when an agent entity is initiated and arrives at a node respectively.

The detail of DisJ internal libraries and usage are available in Appendix E.

The Environment Configuration Component

The DisJ simulation system allows the user to input environment configuration parameters in order to define an environment of a network topology before the simulation

begins. The configuration parameters allow the user to define a simulation network environment required for simulating an algorithm such as the reliability of transmission, the number of agents, the number of token, etc. These configuration parameters have a high impact on the implementation of a reactive distributed algorithm as well as behaviour and characteristics of a network that the algorithm is executed on as discussed in Section 3.1 . In the current implementation, the configuration parameters can only be provided before the start of the simulation and cannot be modified once the simulation has started; the simulation system loads the parameters into an internal data structure only once, before the simulation begins, and the structure cannot be modified. The design of the simulation system allows the user to input the parameters into the simulation system via GUI, which is more intuitive and user friendly than providing the configuration parameters via a configuration file. The simulation system divides the input parameters into two different levels: network topology and network element.

- *Network topology level* allows the user to configure the environment of a network topology as a whole by applying the configuration parameters to every related element in a network topology such as the total number of tokens an agent can carry, a link transmission order, a link transmission delay, etc. This design helps the user to save time from configuring the environment parameters of each element in the network topology manually.
- *Network element level* allows the user to configure the environment of a particular element in a network topology (i.e. node and link elements) to be different than others. For example, configuring a node to be an initiator, configuring a link transmission reliability, configuring a link transmission delay, etc. This

Figure 5.11: The Flow of Runtime Information Process

design helps the user to be able to set particular elements to have different environment configurations than others in order to meet a variety of requirements of the user algorithms.

5.3.3 Runtime Group Architecture

”Runtime Group” is a group of components that are mainly responsible for performing actions when the simulation is running. Based on the execution workflow of the DisJ simulation system that is shown in Figure 5.5, there are six components that are grouped under runtime group: Simulator Engine, Logger, Visual Display, Adversary Control, Random Number Generator, and Replay Engine (see Figure 5.11).

The Simulator Engine Component

The Simulator Engine component is an engine that drives and manages execution of the simulation as discussed in Section 4.3.2. The component creates a virtual

Figure 5.12: Simulator Engine Internal Components

processor, a processor like any other discrete event simulation processor, to simulate the user input (i.e. user protocol and network topology). The virtual processor allows the Simulator Engine to support multiple simulations at the same time. However, the current implementation supports only one simulation at a time. The Simulator Engine component contains five subcomponents as shown in Figure 5.12. Figure 5.13 shows inter-subcomponent relationships within the component. The following paragraphs discuss each subcomponent in more detail.

- *Event:* Event is a collection of data that is executed by Event Processor. Event contains a simulation Timestamp (which indicates a time to execute the Event) and necessary information that is required for the execution of the simulation.
- *Event Queue:* Event Queue is an ordered queue of Events. The queue is ordered by simulation Timestamp from the smallest to the largest Timestamp. The queue provides a sequence of Events based on a simulation Timestamp. When new Event arrives, the queue inserts Event into an appropriate position based on Timestamp assigned to Event. However, Timestamp of new Event will

Figure 5.13: Simulator Engine Internal Components Relationship

never be smaller or equal to Timestamp indicated in Simulation Clock, which means that new Event always created for future execution.

- Time Generator:* Time Generator is a generator that generates Timestamps. Timestamp is a label that represents a simulation time in the form of simulation time units (STU) of a running simulation. The DisJ simulation system defines the STU to be a positive integer number. Time Generator assigns new Timestamp to newly created Event based on a time in Simulation Clock and a request from Event Generator. New generated Timestamp always has higher value than Timestamp indicated on Simulation Clock.
- Simulation Clock:* Simulation Clock is a local clock of an Event Processor of Simulator Engine that is simulating. Simulation Clock indicates the current simulation time of a running simulation. The time of Simulation Clock is Timestamp value, and the value always increases based on currently executing Event. Simulation Clock moves a simulation time value such that it is the same time as Timestamp assigned to current executing Event.

- *Event Generator*: Event Generator creates new Events for the simulation based on the result of executed Event (that was performed by Event Processor). Once new Events have been generated, they will be put into Event Queue for further processing.
- *Event Processor*: Event Processor executes Event that is taken from the top of Event Queue. The execution of Event may lead to the change of simulation environment and status. Furthermore, the execution of Event may or may not cause the creation of one or more new Events. The simulation ends when Event Processor attempts to obtain new Event from empty Event Queue.

The Logger Component

The logger component logs necessary information of the DisJ simulation system. The requirements and details of the logger have been discussed in Section 4.3.3. There are two types of information that are logged by the DisJ simulation system: the System Log and the Simulation Log.

- *System Log*: The system log records the DisJ simulation system warning and error messages. The log is intended for a developer who is developing or extending the DisJ simulation system. The log provides information to the developer in the form of warning and error messages, in which the content of messages are directly related to software bugs that occur in the DisJ simulation system. The content information includes warning/error messages, severity level, Java Exceptions, and Stack Traces from the origin where an Exception has been thrown. This information would definitely help the developer to easily manage, trace, and fix software bugs in the system efficiently.
- *Simulation Log*: The simulation log records any change of network element

status and values during the simulation, and stores it into persistent storage (e.g. an I/O file). The recorded information is used for "Live Presentation" during the simulation execution, "Final Presentation" after the simulation is ended, and "Replay" function when the user needs to replay the simulation. The recorded information is saved as a text format with predefined structure so that the DisJ simulation system can read and extract the information as needed. The structure of a simulation log is defined as follows: each line in the file is a record representing information of a single Event execution, with the exception in the first two lines; the first line represents a list of available states of nodes and agents defined in a protocol, and the second line represents the name of a distributed model and a protocol file. The Event values of each line are separated by a comma. The first value is execution Timestamp of Event. The second value is a type of Event action that is predefined in the simulation system. The remaining values are parameters corresponding to the type of Event action. The record is saved and stored in a file system with file extension ".rec". The example of logging information is shown in Figure 5.14.

The logger has been designed to adopt different ways of recording data into different types of persistent storage and different structure of recored data by providing an Interface class for implementing a particular type of logging system. For example, logging into an I/O file, as shown in Figure 5.15. The LogIO abstract class has implemented the ILogable interface class that has functions such as `initLog()` that initiates an I/O connection, `cleanup()` that disconnects an I/O connection, and `getCurrentTime()` that obtain a simulation time. To record actual information, a concrete logger class must extend the LogIO abstract class. The concrete logger class contains an implementation specific to how elements or objects information is structured in the

```
STATE_FIELD,C,STATE_NODE_UNKNOWN,1:STATE_NODE_CLEAN,  
C,MODEL_AGENT_BOARD,test.BMC  
0,AGENT_LIST,1,0,0,0,0  
0,AGENT_INIT,0,0,0  
0,AGENT_STATE,C,4  
0,NODE_STATE,C,1  
0,AGENT_WRITE_TO_BOARD,C,0,active:left  
0,AGENT_LEAVE,C,0,0,0  
0,AGENT_INIT,0,0,0  
0,AGENT_STATE,1,4  
0,AGENT_WRITE_TO_BOARD,1,0,active:right  
0,AGENT_LEAVE,1,0,0,0  
1,AGENT_ARRIVE,C,C,0,0,0  
1,AGENT_WRITE_TO_BOARD,C,C,4,safe:right  
1,AGENT_LEAVE,C,C,4,0,0  
1,NODE_STATE,C,1  
1,AGENT_ARRIVE,0,0,0,0,0  
1,AGENT_WRITE_TO_BOARD,1,0,0,safe:left  
1,AGENT_LEAVE,1,0,0,0,0  
1,NODE_STATE,C,1  
2,AGENT_ARRIVE,C,0,0,0,0  
2,AGENT_DELETE_FROM_BOARD,C,0,0,active:left  
2,AGENT_WRITE_TO_BOARD,C,0,0,safe:left  
2,AGENT_LEAVE,C,0,0,0,0  
2,AGENT_ARRIVE,1,0,0,0,0  
2,AGENT_DELETE_FROM_BOARD,1,0,0,active:right
```

Figure 5.14: DisJ Simulation Log Content

Figure 5.15: DisJ Log Class Relationship

Figure 5.16: DisJ Log and Network Element Classes Relationship

record. In this way, the designer of a concrete logger class only has to decide what information is to be logged and how the information should be structured in a log. For an abstract class, in this case LogIO, only responsible for accessing and writing the information to a persistent storage.

The benefit of this design is to minimize code modification in the system by decoupling the logging processes that are concerned with the type of persistent storage from the structuring record contents. For example, if there is a change in the structure of logging content, the developer only needs to modify the implementation of a specific concrete logger class instead of going through the whole system and modifying every class that is associated with the change. Figure 5.16 shows class relationships between logger and network elements that are required to be logged.

The Visual Display Component

The visual display component provides a visual presentation to the user that includes simulation playback (see Section 4.3.4), replay playback (see Section 4.3.6), and simulation report (see Section 4.4) as shown in Figure 5.17.

Figure 5.17: Playback Animation and Report are Visual Display of the DisJ Simulation System

The simulation playback in term of DisJ architecture is a subcomponent that presents visual feedback of the progress and status of a running simulation to the user. The Simulator Engine component controls and processes the simulation and provides outcomes to the visual display. However, the playback also acts as an intermediate unit that relays simulation playback control commands from the user to the Simulator Engine component. These commands are: stop, resume, pause, and run. The commands are performed by the Simulator Engine component (not by the playback). Based on the MVC architecture, the playback is a View while the Simulator Engine component is a Controller. Therefore, the simulation playback in DisJ architecture is just a part of the visual display component.

The replay playback has been architected the same way for the same reasons, as the simulation playback. The replay playback presents a visual update of information and status of a network topology to the user, provided by the Replay Engine component. However, the playback also acts as an intermediate unit that relaying replay playback control commands from the user to the Replay Engine component. These commands are: stop, resume, pause, and run. The commands are performed by the Replay Engine component (not by the replay playback). Based on the MVC architecture the replay playback is a View that displays information received from a Controller, which is the Replay Engine component.

The simulation report in terms of the DisJ architecture is a subcomponent that

presents results/outcomes of a simulation to the user visually. The report is managing the layout and format such as text, table, and graph of the information that will be presented to the user. The replay receives information from the simulation system while the simulation is running. According to the MVC architecture, the report is a View that displays information, and the simulation system is a Controller that produce the information data.

The visual display component provides a view to display information that can be divided into two different types of view: an editable view and a non-editable view. These views are discussed below.

An Editable view is a visual presentation that allows a user and other components in the simulation system to interact with the display information by adding information or modifying information to the system via the view itself. An example of an editable view in the DisJ simulation system is a topology editor canvas, which has been discussed in the previous chapter (see 4.2.2). The editor not only allows the user to create topologies of networks for the simulation before the simulation starts, but also presents a visual animation of network status changes while the simulation is running. The updating of the visual animation is based on the results of protocol execution and the user inputs. The design of editable views is based on the Model-View-Controller (MVC) architecture [17] that organizes the classes into three different groups: a model group that represents an information holder, a view group that represents a blueprint of information presentation format and layout, and a controller group that represents a management of information flow. The design of the editable view in the MVC architecture provides information integrity through out different views in the DisJ simulation system even if more than one source is modifying data information concurrently (e.g. the simulator engine and the user). The components that use the editable view are the protocol editor, the topology editor, the network

Figure 5.18: The MVC Architecture for DisJ Views

environment configuration, and the Replay Engine.

A *Non-editable view* provides a read-only information view where a user can only read and re-arrange the presentation format and layout but cannot modify the data itself. An example of a non-editable view in the DisJ simulation system is a Statistics view. The Statistic view provides a statistical report of the algorithm (e.g. the number of messages used, the current number of agents, and relationship of states vs. the number of nodes in a network, etc.). Non-editable view has been designed in a similar manner to the editable view by using the MVC architecture as shown in Figure 5.18. The MVC architecture also allows the contents of a model to be presented in different forms without altering the original contents. For example, a nodeModel object that contains node information can be used to display in the Node view and in the Properties view.

The Adversary Control Component

The adversary control component provides a means for a user to validate the correctness of a simulating protocol by allowing the injection of adversary instructions into

the simulation system. As discussed in Section 4.3.5, an adversary script is a set of instructions (programming codes) that can control and instruct the network environment while the simulation is running. The DisJ simulation system provides interfaces for the adversary script to access and control the simulation network environment in order to respond to the executing protocol. However, it is the user responsibility to make sure that an adversary script does not break rules, definitions, and assumptions of the protocol that is being simulated.

The architecture of the adversary control can be divided into two parts: the first is a library that allows the user to access internal functions of the system via APIs provided in a programming script, and the second part provides a mechanism that allows the simulation system to know when it should execute the adversary script (when the simulation and the executing protocol reach certain conditions).

The first part allows the user to extend an abstract class that provides the APIs to access the simulation environment information and to issue commands to the simulation system by overriding the functions provided as discussed in the previous chapter. The second part allows Simulator Engine to detect the existence of an adversary script by using the polymorphism technique in object-oriented to switch instructions between the default implementation and adversary script implementation. Figure 5.19 shows Simulator Engine executes an instruction either from the default instruction or adversary script. Simulator Engine needs to detect the existing of adversary script instructions; if they exist, the instructions from the adversary script will be executed, otherwise the instructions from the default implementation will be executed.

Figure 5.19: Relationship between Adversary Script Instructions, Default Instructions, and Simulator Engine

The Random Number Generator Component

The random number generator component generates random numbers required by the simulation system to simulate and control the network environment based on the probability model configured by the user before the simulation starts. For example, when a message has been sent from a node via an unreliable link, the simulation system needs to know whether the message is to be dropped on the unreliable link. Let's say the link probability rate of reliability is 30 percent (which is provided by the user). The generator generates a random number with a configured probability model and check whether a generated number (between 1-100 inclusive) is lower or equal to the probability rate provided (in this example is 30). If it is true, then the message will be sent to a destination, otherwise the message will be dropped. The component also allows the user to integrate or import different random number distributions into the component as shown in Figure 5.20. Currently, the DisJ simulation system provides only two common random number distribution models: the Uniform and Poisson distributions. By default, the DisJ simulation system uses the Uniform distribution model to control the simulation and its environment.

The component provides an interface class that enables a user to implement and

Figure 5.20: Random Number Generator and Adaptation of External Distribution

import custom random number distributions. Once the interface class has been implemented, the user can connect the external random number distribution into the DisJ simulation system by using a GUI option provided in the DisJ simulation system workbench (window) to load the implemented interface class before starting the simulation. If custom random number distribution is required and configured, then Simulator Engine can use a polymorphism technique to execute the user implemented codes to generate random numbers. Otherwise one of the default random number generators will be used.

The design of the component works even if the external random number generator library was not written in the Java language. It works as long as the user implements the interface class and uses it as a delegator between the library and the interface of the adversary script. For example, if the external random number generator library was written in C/C++, the user may use Java Native Interface (JNI) [55] inside the implemented interface class to redirect function calls to the library.

The Replay Engine Component

The replay engine component provides a mechanism that allows the user to view past simulations that have been saved by the simulation system as discussed in Section 4.3.6. A past simulation is generally a sequence of state changes of the network

topology and its elements that were affected by event executions. The workflow of the replaying process described in Figure 5.21 is that the replay engine reconstructs the results of Event executions by reading the information recorded in a simulation log. The replay engine provides the results to Simulator Engine. Simulator Engine updates states of the network topology and its elements based on the results received from the replay engine component, then Simulator Engine notifies the visual display component to update views and animation. The following is a detailed discussion of the replay engine component.

1. The replay engine component reads a simulation log (given by the user) that contains the state changes of the network topology and its elements with all necessary information related to the simulation. The content of the simulation log is organized in a format that the replay engine component understands and is able to parse and reconstruct the results of Event executions.
2. The replay engine component feeds the Event results to Simulator Engine (in sequence based on the time of Event execution occurred) via the provided interfaces from Simulator Engine.
3. After Simulator Engine has received the results, it manages the results and updates the simulation states as if the results were received from Event Processor during a simulation.
4. Then, Simulator Engine notifies all related components of the simulation system that are affected by the results and updates.
5. Consequently, the visual display component receives the notification to update the visual display and all views of the simulation system that are affected.

Figure 5.21: The Replay Main Workflow

The process of replaying the past simulation continues until the end of the simulation log is reached, which means there is no result of Event execution left or the past simulation has ended. The process of live simulation and replay are the same from the step 3 to step 5. The replay engine component also provides a playback function for the user to control the progress of the replay in the same manner as the simulation playback including start, stop, resume, and run as discussed in the previous chapter (see section 4.3.6). The commands are received via GUI provided by the visual display component and the replay engine acts accordingly.

5.3.4 Infrastructures

The methodology and approach that the DisJ simulation system adopts has encouraged us to reuse the existing application and codes (see Section 5.2). Therefore, we follow the Eclipse plug-in development [29] approach to implement the DisJ simulation system infrastructure. The infrastructure is a set of utility applications that are used by various components in the DisJ simulation system that includes GUI and coding frameworks. The details of the infrastructure that the DisJ simulation system reuses from the Eclipse platform are discussed later in this subsection. In order to reuse existing applications provided by the Eclipse platform, the DisJ simulation system has to be developed as an Eclipse Plug-in. The Eclipse platform is an extendable platform that provides building blocks and frameworks to facilitate developing new seamlessly-integrated applications (see Figure 5.22). These frameworks and building blocks for developing plug-in application are expressed in terms of well-defined API interfaces, classes, and methods that become mechanisms to use and rules to follow by the plug-in developer. An Eclipse plug-in is the smallest application unit of the Eclipse platform that can be developed. It consists of the Java codes, the Java Archive libraries, some read-only files, and other resources such as images, web templates, message catalogs, and native code libraries, etc., like any normal Java applications. A plug-in is similar to a small Java application that performs a specific task based on the requirements and design. However, the main difference between regular Java applications and a plug-in is that a plug-in can reuse any other plug-ins that have been developed before, and it can be integrated into a new plug-in without having to rewrite codes of the existing plug-in. The intention of the Eclipse platform is to reuse existing applications and frameworks in the future, and the DisJ simulation system embraces and adopts this idea. Further reading about the Eclipse platform

Figure 5.22: The DisJ Plug-in with the Eclipse Platform [73]

Eclipse Components	The DisJ Simulation System Components										
	<i>Replay Engine</i>	<i>Protocol Editor</i>	<i>Topology Editor</i>	<i>DisJ Lib</i>	<i>Env Config</i>	<i>Simulator Engine</i>	<i>Logger</i>	<i>Adversary Control</i>	<i>Random Number Gen</i>	<i>Visual Display</i>	
										<i>Playback</i>	<i>Report</i>
<i>Outline View</i>											
<i>Flow</i>											
<i>Workbench</i>											

Figure 5.23: Mapping the infrastructures that the DisJ Simulation System Components reused from the Eclipse Platform

and Eclipse plug-in can be found at [12] [43].

There are many existing plug-ins that the DisJ system has reused as infrastructures of the system. For examples i) the Eclipse Core plug-in, which provides many foundation components that are required by the DisJ system such as runtime management, file system, networking management, data binding, error logging, resource management, etc., ii) the Eclipse JDT plug-in, which provides Java development tool such as a Java code editor, a Java code assist, a Java compiler, etc., iii) the Eclipse UI plug-in, which provides a graphical interface that include workbench, view, toolbars, menus, canvas, etc., and iv) the GEF plug-in, which provides graphical editor framework with 2D graphical drawing tools that are used by the topology editor to draw, render, display, edit, etc. These plug-ins provide several components that the DisJ simulation system can reuse as infrastructures for the simulation system as shown in Figure 5.23. A discussion of the Eclipse components that are used by the DisJ simulation system follows.

Eclipse Core

The Eclipse core component provides basic infrastructure for the Eclipse platform as well as every Eclipse plug-in including resource management, generic file system API, concurrent control within the Eclipse platform, Eclipse plug-in coordination, and others. Every Eclipse plug-in requires the Eclipse core component, even if the plug-in developer does not directly interact or use the component, because the core component handles the low level of plug-in development which includes coordination communication between plug-ins, managing resource sharing between plug-ins, etc.

Eclipse Error Log

The Eclipse error log component is the part of the Eclipse Core plug-in that is responsible for logging system error within the Eclipse application and displaying error messages to the user via an Eclipse error log view. The Eclipse error log message contains information such as the Exception messages, the Java Stack Trace, the time of Exception occurs, the plug-in name, the Java Runtime Library version, and the Eclipse build number. Figure 5.24 is an Eclipse error log view that shows a DisJ system error caught by the Eclipse error log while running the DisJ simulation system. Further information on the Eclipse error logging system can be found in [43] [29]. The Eclipse error log is used by every component in the DisJ simulation system in order to log system errors and present the error messages to a developer who is developing the DisJ simulation system.

Java Development Tool

The Eclipse platform provides a Java Development Tool (JDT) plug-in to support a Java development environment. JDT provides an editor canvas, called the editor part,

Figure 5.24: The System Log View when system error occurs in the DisJ simulation system

for the user to write and edit Java codes on a blank slate. The JDT plug-in supports several code development functionalities such as syntax validation, code formatting, code assist function, Java compilation, console view, runtime library linking, etc. The DisJ simulation system can reuse and integrate the JDT plug-in with the Protocol Editor component and the DisJ internal library component. These two components require the support of the Java development environment and linking mechanism between a user protocol and the DisJ internal library during development; the JDT plug-in supports that requirements. Further information about JDT plug-in can be found at [73].

Graphical Editor Framework

The Graphical Editing Frameworks (GEF) plug-in [3] provides a framework for the 2D graphical editor. GEF provides a Draw2d component for rendering and displaying graphic on a canvas, and the GEF-MVC component for interactive Model - View -

Figure 5.25: Protocol Editor with JDT plug-in

Figure 5.26: Topology Editor with a Palette on the Righthand Side

Controller framework that combine SWT (Simple Widget Toolkits) and Draw2d to handle data model and user interactions on an editor. The topology editor component allows the user to create and edit a network topology in 2D graphics. Therefore, the topology editor component has adopted GEF to develop a 2D graphics editor with ease. For example, GEF provides a class for the topology editor component to implement in order to get a canvas with the palette as shown in Figure 5.26 (without creating a palette manually). GEF also provides a high level API for the topology editor component to create 2D geometric shapes by extending a specific geometric class, e.g. "RoundRectangle" class, in which the super class implementation takes care of drawing and rendering a round rectangle graphic shape on a canvas with a

given size and colour. Further information on the GEF plug-in can be found at [3].

Eclipse UI Workbench

The Eclipse UI workbench (or window) is the main container of the Eclipse platform that contains every GUI component used in the Eclipse platform (Eclipse view and Eclipse part) as shown in Figure 5.27. The responsibility of the workbench is to manage the layout of the GUI component, and coordinate the selection (or focus) of GUI component that the user is currently interacting with. Moreover, the Eclipse UI workbench provides a framework for the common operations of every GUI component such as Zooming in/out on a GUI component, dragging and dropping a GUI component in and out the workbench, etc. The Eclipse UI workbench is a foundation for every GUI component of the DisJ simulation system that interacts with the user as shown in Figure 5.23. Further information on the Eclipse workbench can be found at [87].

Eclipse View

The Eclipse view is an Eclipse GUI component that forms an Eclipse workbench. The majority of GUI components in the Eclipse workbench are the Eclipse views. The view is a Simple Widget Toolkit (SWT) [61] called "Composite" class that contains a title bar. The title bar contains the view name, an area for toolbar buttons, and drop-down menu. The drop-down menu consists of actions for moving, resizing, and closing the view. The DisJ simulation system uses the Eclipse view to display current simulation information and a simulation report to the user as shown in Figure 5.28. Further information on the Eclipse view can be found at [77].

Figure 5.27: The Eclipse Workbench that Contains Some of Default GUI Components

The screenshot shows the DisJ View with the following data:

Node ID	IsAlive	Node State	# Token	Whiteboard
n1	True	STATE_NODE_TO_BH	0	safe:left
n0	False	STATE_NODE_UNKNOWN	0	safe:right
n5	True	STATE_NODE_CLEAN	0	
n4	True	STATE_NODE_CLEAN	0	
n3	True	STATE_NODE_CLEAN	0	
n2	True	STATE_NODE_CLEAN	0	
n6	True	STATE_NODE_TO_BH	0	

Figure 5.28: The DisJ View that extends Eclipse View

Figure 5.29: The Outline of the Topology Editor in the Eclipse Outline View

Eclipse Outline View

The Eclipse Outline view is a special Eclipse view that displays an outline of selected GUI component in the Eclipse workbench. The Eclipse Outline view can display an outline in different ways based on the implementation of a plug-in developer. The Eclipse Outline view provides a default implementation for displaying an outline in the form of a tree structured of elements contained in the selected GUI component. The DisJ simulation system uses the Eclipse Outline view for the protocol editor and the topology editor; the protocol editor uses the default implementation (see Figure 5.30) and the topology editor uses an override implementation version to display a mini network topology that mirrors a network topology on a editor part (see Figure 5.29). Further information on the Eclipse workbench can be found at [29]

Figure 5.30: The Outline of the Protocol Editor in the Eclipse Outline View

Eclipse Properties View

The Eclipse Properties view is another special Eclipse view that displays properties of a selected items in the Eclipse workbench. The items may reside in a GUI component within the Eclipse workbench. The information displayed in the properties view corresponds to the selection of a GUI component or the item in it. However, the GUI component or item must register to the Properties view during the implementation of the GUI component. The Properties view can be set to be editable or non-editable during the registration. The DisJ simulation system uses the Properties view to display information related to a selected component and its elements, including the playback subcomponent for displaying current simulation status, the protocol editor component for displaying information of a Java file and its elements, the topology editor component for displaying current information of network topology and its elements, and the environment configuration component for displaying a current network configuration parameters. Figure 5.31 shows a snapshot the Properties view

Property	Value
G00 Graph Name	insard_cs.gph
G01 Total Nodes	7
G02 Total Links	7
G03 Total Messages Received	0
G04 Total Messages Sent	0
G05 Global Message Flow Type	FIFO
G06 Global Delay Type	Synchronous
G07 Global Delay Seed	1
G08 Protocol	1
G10 Number of Agents at Start	2
G11 Current Number of Agents	0

Figure 5.31: The Properties View is Displaying a Topology Information

that displays a topology information. Further information on the Properties view can be found at [47].

5.4 DisJ Implementation

This section discusses the implementation of the DisJ simulation system using the Java programming language with the Object Oriented methodology. The first subsection shows the structure of the programming codes, and shows how the system can be built and deployed. The second subsection shows how the DisJ system reuses existing plug-ins from the Eclipse platform. The third subsection shows implementation codes with respect to the design patterns in Object Oriented Design (OOD). The full implementation code of the DisJ simulation system is located in Appendix D of this thesis.

5.4.1 Coding Structure and Deployment

The DisJ simulation system has been developed under the Eclipse PDE (Plugin Development Environment), using the Java programming language version 5 in developing the system. The organization of the code strictly follows the guidelines and best practices of the Java community [67]. For the deployment of the system, the DisJ simulation system follows the Eclipse Plug-in development guidelines defined by the

Eclipse community [29].

Codes Structure and Organization

The structure of the DisJ simulation system programming code is grouped into five different groups called "Core", "Runtime", "Statistics", "Random Number Model", and "UI" (User Interface). Each group consists of one or more Java packages corresponding to roles and responsibilities of the components of the DisJ simulation system. The following content describes the Java package list for each group that the DisJ simulation system has organized during the implementation.

- Core group
 - distributed.plugin.core
- Runtime group
 - distributed.plugin.runtime
 - distributed.plugin.runtime.adversary
 - distributed.plugin.runtime.engine
- Statistic group
 - distributed.plugin.stat
- Random Number Model group
 - distributed.plugin.random
- User Interface (UI) group
 - distributed.plugin.ui

- `distributed.plugin.ui.actions`
- `distributed.plugin.ui.commands`
- `distributed.plugin.ui.dialogs`
- `distributed.plugin.ui.editor`
- `distributed.plugin.ui.figures`
- `distributed.plugin.ui.models`
- `distributed.plugin.ui.models.topologies`
- `distributed.plugin.ui.parts`
- `distributed.plugin.ui.parts.policies`
- `distributed.plugin.ui.valiators`
- `distributed.plugin.ui.views`

The Java packages can be structured in different ways. There are a few differences between them, and there are tradeoffs depending on which structure is used. Therefore, this thesis does not state that the structure and packaging of the DisJ simulation system is the best way, as it is one of many ways of doing it. For example, the Core group in our approach consists of one Java package that contains all Java classes that relate to key objects which are used and related to every component in the DisJ simulation system (e.g. Node class, Agent class, DisJException class, IConstants Interface, ITransmissible Interface, ILogable Interface, LogIO class and Logger class). However, the package may not include all three log related classes. It may have its own package because there are many ways of logging besides logging into an I/O file (such as logging into a database). The reason that these three log related classes currently reside in the Core group is because it is used throughout the system and our current implementation only requires one way of logging. However, if the

Figure 5.32: GUI for DisJ Build Configuration and Deployment

system needs to support other types of logging such as logging into a database, in which more classes and higher complexity are needed, then the Core group might only have an ILogabel Interface. All other log related classes would be located in different package.

System Deployment

The DisJ simulation system is deployed in the same way as any other Eclipse plug-in. The Eclipse PDE provides a Wizard exporter to produce a Java JAR file that can be directly deployed under the Eclipse_Home\plugins\ directory. Therefore, the user only needs to copy the deployed JAR file into the directory and restart the Eclipse platform and the DisJ simulation system will be available. Figure 5.32 and Figure 5.33 are screenshots that show the configuration of GUI pages provided by the Eclipse PDE for deployment of the Eclipse plug-in. Further information about how to export Eclipse plug-ins using the Eclipse PDE can be found at [29]

Figure 5.33: The Eclipse IDE Provides Plug-in Export GUI Wizard

5.4.2 Reusing Existing Applications and Codes

The DisJ simulation system reuses many existing codes and applications from the Eclipse Foundation projects in order to speed up the development time, saving resources, and enhancing the reliability of the system. Indeed, these Eclipse projects contribute the majority of codes used in the DisJ simulation system. The following paragraphs discuss how to reuse existing plug-ins from the Eclipse foundation projects in the DisJ simulation system.

The look and feel of the DisJ simulation system workbench is very similar to the Eclipse workbench because the DisJ simulation system workbench has extended the Eclipse workbench. The functionalities in the DisJ simulation system workbench include all functionalities available in the Eclipse workbench plus some new functionalities provided by the DisJ simulation system, such as simulation playback functions, configuration functions, etc. The main concept of the design of the Eclipse platform is to provide an extension point for any plug-in application to connect to the platform. The DisJ simulation system uses this extension point of existing plug-ins in the

Figure 5.34: GUI Configuration Plug-in Dependency for the DisJ Simulation System

platform to connect by specifying XML scripts in "*plugin.xml*" of the DisJ simulation system as shown in Figure 5.35. Each XML extension block in Figure 5.35 refers to a specific existing plug-in that the DisJ simulation system prefers to use. The extension point contains information such as a target plug-in ID, a Java entry class name, a contributor Java class name, etc. These are required by the Eclipse plug-in specifications in order for the Eclipse platform to link and perform internal merging of these plug-ins together into a single application. Further details on the Eclipse platform are available in [29].

Figure 5.34 shows a GUI that the Eclipse platform provides for a plug-in developer to specify the dependency plug-ins from existing plug-ins in the platform that the DisJ simulation system requires. The difference between the dependency plug-in and the extension plug-in is that the dependency plug-in is used as part of the DisJ simulation

```

<?xml version="1.0" encoding="UTF-8"?>
<eclipse version="1.0"?>
<plugin>
  <extension
    point="org.eclipse.ui.editors">
    <editor
      filenames=".gph"
      icon="icons/editor.gif"
      class="distributed.plugin.ui.editor.GraphEditor"
      default="true"
      name="DisJ Editor"
      contributorClass="distributed.plugin.ui.actions.GraphEditorActionBarContributor"
      id="distributed.plugin.ui.editor.GraphEditor"/>
    </extension>
    <extension
      point="org.eclipse.ui.views">
      <category
        id="org.carleton.scs.disj"
        name="DisJ">
      </category>
      <view
        category="org.carleton.scs.disj"
        class="distributed.plugin.ui.view.DisJViewer"
        icon="icons/editor.gif"
        id="distributed.plugin.ui.view.DisJViewer"
        name="DisJ View">
      </view>
    </extension>
    <extension
      point="org.eclipse.ui.perspectiveExtensions">
      <perspectiveExtension
        targetID="org.eclipse.jdt.ui.JavaPerspective">
        <view
          id="distributed.plugin.ui.view.DisJViewer"
          ratio="0.5"
          relationship="right"
          relative="org.eclipse.ui.views.TaskList">
        </view>
      </perspectiveExtension>
    </extension>
    <extension
      point="org.eclipse.help.contexts">
      <contexts
        file="contexts.xml">
      </contexts>
    </extension>
  </plugin>

```

Figure 5.35: XML Scripts Describes the Extension Plug-ins in Eclipse

Figure 5.36: The Factory Method Pattern used in the DisJ System

system without modifying the original functionality, while the extension plug-in is a plug-in that the DisJ simulation system uses while modifying and expanding its functionalities. For example, the Core plug-in is a dependency plug-in for the DisJ simulation system, while the UI-View plug-in is an extension plug-in that the DisJ simulation system extends in order to display the simulation information and reports.

5.4.3 Using Software Design Patterns

The DisJ simulation system adopts OO methodology, which is applied to every component in the system. The design pattern helps the DisJ simulation system to preserve OO properties and makes the system manageable, maintainable, and extendable for future works as required by the design goal. The following discussion shows three examples of common design patterns from each category of software patterns: the creational patterns, the structural patterns, and the behavioural patterns, which have been used in different components of the DisJ simulation system [32].

Figure 5.37: The Faade Pattern used in DisJ

Creational Patterns

The "Factory Method" pattern has been categorized under the creational patterns category. Creational patterns help make the system to be independent from objects creation. The intent of the "Factory Method" pattern is to define an interface for creating an object and to let the subclass define an implementation of an object [32]. Therefore, the DisJ system uses the "Factory Method" pattern (see Figure 5.36) to create a topology from a ready-made topology command in the topology editor. There are different types of topologies but the nature of each topology is the same, e.g. every topology consists of nodes and links that connect together and form a network. Therefore, how nodes and links are formed will be defined in the subclasses that belong to a specific topology, while a superclass that implements the interface for topology creation is common to every subclass.

Figure 5.38: The Template (Algorithm) Pattern used in DisJ

Structural Patterns

The "Facade" pattern has been categorized under the structural patterns category. Structural patterns help compose classes and objects into a larger structure and allow independent systems to work together as a single system. The intent of the Facade pattern is to provide a unified interface between the set of user defined interfaces and the system [32]. Therefore, the "Facade" pattern (see Figure 5.37) has been used in the Adversary Control component to connect an independent adversary script to the DisJ simulation system. The DisJ simulation system uses the pattern in such a way that the interface is used by Simulator Engine to access the user adversary script. Since an algorithm of the adversary script provided by the user can be very complicated and unknown to the simulation system, the simulation system provides a limited number of adversary functions that seem sufficient for the user to use (as discussed in the previous chapter).

Behavioral Patterns

The "Template" pattern has been categorized under the behavioural patterns category. Behavioural patterns help to categorize complex control and communication flows between classes and objects in the system and let the implementation focus on the algorithm rather than the flow of the system. The intent of the "Template" pattern is to define the skeleton of an algorithm in an operation, and to defer the details of each step of operation to subclasses [32]. The "Template" pattern (see Figure 5.38) has been used in the DisJ simulation system because the simulation system supports three different distributed computing models in which each model has its own processor that process a protocol in its own way. The model processors used in the simulation system fall into this category, since every processor has the same simulation flow and lifecycle; the differences are only in the details of the operation algorithms that have to be based on the specific model that the processor corresponds to. Therefore, a superclass provides a template set of operations that represents a lifecycle of the simulation process, while a subclass has to implement the interfaces corresponding to each operation based on the model definitions and requirements.

Chapter 6

Case Studies

This chapter discusses some case studies on the usage of the DisJ simulation system. The case studies contain three different distributed algorithms with three different models and one adversary script: a Ring Election algorithm with the Message Passing model, a Black Hold Search algorithm with the Mobile Agent with Whiteboard model, a Black Hold Search algorithm with the Mobile Agent with Token model. These case studies are discussed in three separate sections that are devoted to each experiment respectively.

The objectives of the case studies presented in this chapter are to demonstrate the usage of the DisJ simulation system and the applying of algorithm pseudo codes to writing a protocol in the Java language. Moreover, the size of network topologies presented in the case studies is small for simpler discussions. The maximum size of the network topology that DisJ supports is totally based on the processing power of the machine running the simulation.

6.1 The Ring Election algorithm for the Message Passing Model

6.1.1 Problems

The entities in distributed environments are autonomous, anonymous and homogenous which create symmetry in the environment. However, many applications in distributed environment often require a single entity to temporarily coordinate and control task(s) in order to simplify the complex problem and because of by the nature of the problem [69]. Therefore, the election process has been introduced to break this symmetry in the environment. The election process is a process of finding a unique entity from the entities in the network environment. Examples of applications that use election algorithm are mutual exclusion, synchronization, and concurrency control, etc.

6.1.2 The Assumptions of the Network Environment

The election algorithm demonstrates in this section is designed with the following environment restrictions.

- The network topology is a ring topology
- The communication links are bi-directional links
- The communication links are total reliable
- The transmission order of the links is random (no FIFO nor LIFO)
- There exists a global unique ID assigned to each and every entity in the network

6.1.3 The Pseudo Code

The ring election algorithm in a ring network with Message Passing model that is used in this experimental is called As Far protocol that the original protocol has been presented in [15]

PROTOCOL: As Far

States: $S = \{\text{ASLEEP}, \text{AWAKE}, \text{FOLLOWER}, \text{LEADER}\};$

SINIT = {ASLEEP};

STERM = {FOLLOWER, LEADER};

AT STATE: ASLEEP

Spontaneously

begin

INITIALIZE;

become AWAKE;

end

Receiving("Election", value)

begin

INITIALIZE;

if value < min then

send("Election", value) to other;

min:= value;

endif

become AWAKE;

end

AT STATE: AWAKE

Receiving("Election", value)

begin

if value < min then s

end("Election", value) to other;

min:= value;

else

if value min then NOTIFY endif;

endif

end

Receiving(Notify)

send(Notify) to other;

become FOLLOWER;

end

AT Procedure: INITIALIZE

begin

send("Election", id(x)) to right; min:= id(x);

end

AT Procedure: NOTIFY

begin

send(Notify) to right;

become LEADER;

end

6.1.4 The Implementation of the As Far Protocol in Java with DisJ

The following Java codes show a Java Class definition of the As Far protocol for the Message Passing model. The `AsFarRingElectMsgPass` is an implemented Class that extends the Entity Abstract Class in order to access the Message Passing communication APIs and to notify the simulation system to create a Message Passing processor to simulate the protocol. The implemented Class represents a main computing entity. In this case it is a node that performs computations of the algorithm in the Message Passing model.

```
public class AsFarRingElectMsgPass extends Entity
```

The implemented Class requires the states of the entity object in which it will be used for the computations and simulation presentations. The user must define the states of the entity objects in respect to the rules. The rules include the variable visibility must be public, the variable type must be an integer, and the variable name of the state should begin with the word state that is non-case sensitive. The following Java codes are examples of defining variables for the entity object.

```
public static final int STATE_UNKNOWN = 0;
public static final int STATE_CANDIDATE = 1;
public static final int STATE_PASSIVE = 2;
public static final int STATE_LEADER = 3;
```

The Entity Abstract Class provides an abstract function for the user to implement called `init()`, which is used for initiating entity objects that are initiators of a

protocol. The `init()` function is used only if the entity has been defined as the initiator in a topology. Otherwise, it is never used. The following Java codes are the implementation of `init()` function of the As Far protocol.

```
public void init() {  
    this.myId = this.getId(this.getName());  
    this.leaderId = 99;  
  
    this.become(STATE_CANDIDATE);  
    this.sendTo(MSG_ELECTION, DIRECTION, this.myId);  
    System.out.println(this.myId + " sent " + MSG_ELECTION + " to "  
        + DIRECTION + " with " + this.myId);  
}
```

The Entity Abstract Class also provides another abstract function, called `receive()` for the user to implement. The function is called when the entity receives messages. According to the As Far protocol described in the pseudo code in section 6.1.3, the entity receives messages only when it is in one of three states: a candidate, a passive, and a leader. The following Java codes show how the function `receive()` handles a message when it is arrived at a node (entity).

```
public void receive(String portLabel, IMessage msg) {  
    String msgLabel = msg.getLabel();  
    Object content = msg.getContent();  
  
    if(this.getState() == STATE_CANDIDATE){  
        this.iAmCandidate(msgLabel, content);  
    }
```

```
} else if (this.getState() == STATE_PASSIVE){  
    this.iAmPassive(msgLabel, content);  
  
} else if (this.getState() == STATE_LEADER){  
    if(msgLabel.equals(MSG_LEADER)){  
        // arrival of my throne notification  
        // done and terminated  
        System.out.println(this.myId + ": I am a leader!");  
    }  
    // ignore the rests  
} else {  
    // should not happen!  
}  
}
```

The user can access the entity object and the topology information from provided API functions. For example, the functions that were used in the above `receive()` function are the function `this.getState()`, which provides a current state of the entity object that calls the function, and the function `msg.getContent()`, which provides the content of a message that the entity received. Another abstract function that the Entity Abstract Class provides is the `alarmRing()` function. The function is called when the internal alarm clock of the entity is rung. In the As Far protocol the function `alarmRing()` is not required, therefore, the user may leave it as an empty function as shown in the following Java codes.

```
public void alarmRing() {  
    // ignore it
```

Figure 6.1: A snapshot of DisJ during the simulation of the As Far protocol.

}

The full detail of the implementation of the As Far protocol is available in Appendix A, and the full detail of the usage of the DisJ simulation system is available in the user manual of DisJ in Appendix E. Figure 6.1 is a snapshot of the DisJ simulation system during the simulation of the As Far protocol.

6.1.5 The Simulation Results

The following are the simulation results example that DisJ provide in text format in the Eclipse console.

```
*****Simulation for test.gph is successfully over*****
```

```
***** STATISTIC REPORT *****
```

```
Total Message has been sent: 16
```

```
Total Message has been received: 16
```

```
Total Message has entered link: 16
```

```
Total Message has leaved link: 16
```

Total Execution time: 335 STUs

Average Message traveling time(accumulated): 53 STUs/link

State STATE_LEADER has 1 node(s)

State STATE_FOLLOWER has 4 node(s)

Message election has been sent 11 times

Message leader has been sent 5 times

The output shown in this case study is just an example to demonstrate the ideas that the DisJ simulation system can capture the simulation activity and information from the network environment and the protocol execution, and presents it to the user. Therefore, we leave further detail of implementation of presentation for future works

6.1.6 Worst Case Scenario of the As Far Protocol

The As Far election protocol in a ring topology uses an elimination technique to elect a leader (a smallest ID) in the network by sending one-way messages with entity IDs that are globally unique in the environment. The message will be eliminated by an entity that receives the message if the entity ID is smaller than an ID in the message. Therefore, the number of messages transmitting depends on the order of entity IDs connected in a ring topology. Another condition that affects the number of messages transmitting is the order of transmission in the topology links because the topology links do not support FIFO transmission. The worst case scenario is that the ring topology contains a sequential order of entity IDs and the messages are being transmitted in the direction of a smaller entity ID to a bigger entity ID, except on the join link between the smallest entity ID and the biggest entity ID (see Figure

Figure 6.2: The Ring Topology Connected in the Worst Case Scenario

6.2). The analysis of the number of messages transmitted for election in the worst case scenario is a $\sum n$, where n represents the number of entities in the ring topology, and the number of messages used for notification is equal to the number of entities in the ring topology. Therefore, the total number of messages used is $\sum n + n$.

In order for the user to create the worst case scenario in the simulation environment, the user has to first create a ring topology of entities in sequential order based on the entity ID. Second, the user has to write an adversary script to control the messages transmitting in FIFO order in each and every link in the topology.

6.1.7 Writing an Adversary Script for the As Far Protocol

The user wants to create the worst case scenario for the As Far protocol in a ring topology as discussed in previous subsection with the DisJ simulation system. The first condition, the user can manually create the ring topology with entities IDs connected in sequential order or writing a protocol to generate unique IDs and assigned

them in sequential order to every entity in the ring before the As Far protocol starts. In this case study, we choose the first option in order to make the discussion simpler. The second condition, the user requires to write an adversary script to control the transmission of messages in the simulation environment to be FIFO in every link. The following is an adversary script written in Java that makes sure the transmission is FIFO during the simulation.

```
public class FifoDelay extends MsgPassingControl {

    /*
     * Let the message arrives at destination in FIFO order and
     * the message takes only 1 time unit of transmission
     */

    public int setArrivalTime(final IMessage msg, String edgeId, String desNodeId) {
        int delay = 0;
        int curTime = this.getCurrentTime();
        delay = curTime + 1;
        return delay;
    }

    @Override
    public BlockFlag arrivalControl(IMessage msg, String nodeId, String portId) {
        // do nothing
        return BlockFlag.DEFAULT;
    }

}
```

Figure 6.3: A snapshot of the User Adding the Adversary Script into DisJ

Figure 6.4: A snapshot of DisJ during the simulation of the As Far protocol with the FifoDelay Adversary Script

The following Figure 6.3 and Figure 6.4 are snapshots of the DisJ simulation system during the simulation of the As Far protocol in worst case scenario and with the adversary script.

6.1.8 The Results Comparison

The following are the simulation results that DisJ provide in text format in the Eclipse console.

```
*****Simulation for ring.gph is successfully over*****
***** STATISTIC REPORT *****
Total Message has been sent: 20
Total Message has been received: 20
Total Message has entered link: 20
Total Message has leaved link: 20
Total Execution time: 11 STUs
Average Message traveling time(accumulated): 4 STUs/link

State STATE_LEADER has 1 node(s)
State STATE_FOLLOWER has 4 node(s)

Message election has been sent 15 times
Message leader has been sent 5 times
```

By observing the output results in this case study (worst case scenario) and the results from subsection 6.1.5 (average case scenario), we can see that the number of messages transmitted in the worst case scenario is higher than the number of messages transmitted in the average case scenario. The number of messages transmitted in the worst case scenario produced by the simulation is exactly equal to the number claimed by the As Far algorithm analysis in the worst case scenario. The number of messages transmitted in the worst case scenario is equal to $\sum n + n$, where n is total number of nodes in the ring topology. In this example, n is equal to 5, so the number of

messages used for election process is $\sum 5 = 5 + 4 + 3 + 2 + 1 = 15$, and the number of messages used for notification is n , which is 5. Therefore, the total number is 20 messages.

6.2 Black Hole Search algorithm for the Agent with Whiteboard Model

6.2.1 Problems

In a large network environment such as the Internet, server farms, data grids, etc., commonly use mobile agents to search and locate some required item in the environment, then report back its location. In general, this searching process assumes that the agent has no danger during the process. However, in reality there exist stationaries (nodes) that destroy any visiting agent and leave no trace of the destruction. Therefore, the searching of a danger stationary, or black hole, in the network is unavoidable. The searching process requires at least one survived agent to be able to unambiguously report the location of the danger stationary (node).

6.2.2 The Assumptions of the Network Environment

The black hole search algorithm that discussed in this section has the following assumptions and restrictions for the environment as follow.

- The network topology is a ring topology
- The communication links are bi-directional links
- The communication links are total reliable
- The transmission order of the links is FIFO

- The transmission delay is synchronous
- There exists a global unique ID assigned to each and every agent
- There are at least two agents involve in the process and all start at the same location called home base.
- The agents know total number of agent involve in the process
- The agents know the size of network
- There is only one fixed black hole exists in the network

6.2.3 The Pseudo Code

The black hole search algorithm in a ring topology with the Agent with Whiteboard model shows in this example is called Co-Located Agent Black Hole Search algorithm presented in [19].

PROTOCOL: Black Hole Search with Co-Located Agents

Let i denoted to the phase number of exploring E_i denoted to an explored and safe set of node U_i denoted to an unexplored set of node There are two agents called L and R n_0 denoted to be a Home Base for every agent Note that E_i and U_i are disjoint set of nodes in the network.

Start with $i = 1$, $E_i = \{n_0\}$, and $U_i = n_1, n_2, \dots, n_n - 1\}$

1. Divide U_i into two continuous disjoint part of almost equal sizes e.g. U_iL is to the left of n_0 and U_iR is to the right of n_0 .
2. Let agents L and R explore U_iL and U_iR respectively by using *Cautious Walk*.

3. Since U_iL and U_iR are disjointed, so, one of them contains the black hole. Therefore, there will be only one agent completes the exploration. Let say, R completes U_iR and turns nodes in U_iR into E_iR , and returned back to n_0 by following safe path through E_iR .
4. Once agent R arrived at n_0 . Then follows safe paths that agent L left in U_iL and turns each node into E_iR until the end of the safe path, say node v .
5. If $|E_iR|$ is equal to $|N - 1|$, then the outgoing unsafe port from v leads to a black hole; exit the protocol. Otherwise Increment i by 1, and divides U_iL leftover into two almost equal parts called $U_{i+1}L$ and $U_{i+1}R$. Then left a note on the Whiteboard.
6. Agent R follows safe path back from node v to the last explored node in U_iR , say v' and creates a new set of $E_{i+1}R$,
7. Agent R continue explore $U_{i+1}R$ from v' by using *Cautious Walk*. If agent L still alive and returns to node v and sees the note on the Whiteboard, then creates new $E_{i+1}L$ and continue explores $U_{i+1}L$ from v by using *Cautious Walk*
8. Continue Step3 to Step7 with the phase of i is in increasing order, and eventually the protocol will be terminated by one of the agent at Step5, let say at phase i' by agent R will have $|E_{i'}R|$ is equal to $|N - 1|$

PROCEDURE: Cautious Walk

1. An agent moves from node u to v via an unexplored port p by labelling the port p into active, once arrived at node v , the agent returns to u immediately via port p and changes the label of port p in node u to safe.

2. Then the agent resume step1 for other ports on u or returns to node v via port p , and resume step1 at node v .
3. No agent enters the port that has label *active*.

6.2.4 The Implementation of Co-Locate Agent Black Hole Search Protocol in Java with DisJ

The following Java codes show a Java Class definition of the Co-Locate Agent Black Hole Search protocol for the Mobile Agent with Whiteboard model. The implemented Class must extend BoardAgent Abstract Class in order to access the Whiteboard communication APIs and to notify the simulation system to create a Mobile Agent with Whiteboard processor to simulate the protocol. The BoardAgent Abstract Class represents a main computing entity. In this case it is a mobile agent that performs computations of the algorithm in the Mobile Agent with Whiteboard model.

```
public class BlackHoleSearch extends BoardAgent
```

The BlackHoleSearch Class is an implemented class that extends the BoardAgent Abstract Class. The implemented Class requires states for both agent and node objects in which it will be used for the computations and the simulation presentations. The user must define the states of the objects in respect to the rules, which includes the variable visibility must be public, the variable type must be an integer, and the variable name of the state should begin with the word state that is non-case sensitive. The state variables can be used for both agent and node in the network topology. The following Java codes are examples of defining variables for the entity object.

```
public static final int STATE_NODE_UNKNOWN = 0;
public static final int STATE_NODE_CLEAN = 1;
```

```
public static final int STATE_NODE_TO_BH = 2;

public static final int STATE_AGENT_WORKING = 4;
public static final int STATE_AGENT_FOUND_BH = 5;
public static final int STATE_AGENT_DONE = 6;
```

The BoardAgent Abstract Class provides an abstract function for the user to implement called `init()`. The function is used when an agent is initiated at the beginning of the protocol. The `init()` function is used by every agent that is defined in the topology. The following Java codes are the implementation of `init()` function of the Co-Locate Agent Black Hole Search protocol.

```
public void init() {
    this.become(STATE_AGENT_WORKING);
    this.setNodeState(STATE_NODE_CLEAN);

    this.round = 1;
    int o = (this.getNetworkSize() - 1)/2;
    int e = this.getNetworkSize() - o - 1;
    List<String> ports = this.getOutPorts();
    List<String> info = this.readFromBoard();
    String p = null;
    System.out.println("Board: " + info);
    if(info.isEmpty()){
        // first agent
        this.numReq = 0;
        p = ports.get(0);
```

```
this.appendToBoard("active:" + p);
}else{
// second agent
this.numReq = e;
String s = info.get(0);
String[ ] v = s.split(":");
for(int i =0;i < ports.size(); i++){
p = ports.get(i);
if(!p.equals(v[1])){
break;
}
}
this.appendToBoard("active:" + p);
}
// set for next exploring
this.moveTo(p);
}
```

In the function `init()` presented above, there are examples of an agent accesses a Whiteboard at a node that the agent is currently resides for reading and writing, `this.readFromBoard()` and `this.appendToBoard("active:"+p)` functions respectively. The `this.readFromBoard()` function returns a list of all information currently on the Whiteboard, and the `this.appendToBoard("active:"+p)` function writes the given parameter to the Whiteboard. The another internal interesting function that used in the `init()` function is `this.moveTo(p)`, which transmits an agent out through a given outgoing port of a node that it currently resides. Also the `BoardAgent` Abstract Class provides another abstract function, called `arrive()` for the user to implement.

The function is called when an agent arrives at a node. The function provides a parameter called `port` which is an incoming port of the node that the agent is entered. The following Java codes show the definition of the function `arrive()` for the `BoardAgent` Abstract Class.

```
public void arrive(String port)
```

The `BoardAgent` Abstract Class provides internal API functions for the user to access agent and node objects information. For example, in the `init()` function shown above, the function `this.become(STATE_AGENT_WORKING)` allows the user to set a current state of the agent object that calls the function. The function `this.getOutPorts()` provides a list of outgoing ports of the node that the agent currently resides in. The `BoardAgent` Abstract Class provides two other abstract functions for the user to implement: the `alarmRing()` and the `notification()` functions. The `alarmRing()` function is called when the internal alarm clock of the agent rings. The `notification()` function allows the agent to register activities occurring within the node that the agent currently resides in (e.g. another agent arrives at the node, another agent accesses the Whiteboard, etc.). In the Co-Locate Agent Black Hole Search protocol, the `alarmRing()` and the `notification()` functions are not required. Therefore, the user may leave them as empty functions as shown in the following Java codes.

```
public void alarmRing() {
```

```
}
```

```
public void notified(NotifyType type) {
```

Figure 6.5: A snapshot of DisJ during the simulation of the Co-Locate Black Hole Search protocol

Figure 6.6: A snapshot of the Agent View during the simulation of DisJ

}

For the full detail of the implementation of the Co-Locate Agent Black Hole Search protocol is available in Appendix B, and the full detail of the usage of the DisJ simulation system is available in the user manual of DisJ in Appendix E. Figure 6.5 and Figure 6.6 are snapshots of the DisJ simulation system during the simulation of the Co-Locate Black Hole Search protocol.

6.2.5 The Simulation Results

The following are the simulation results example that DisJ provide in text format in the Eclipse console.

```
*****Simulation for board_co.gph is successfully over*****
```

```
***** STATISTIC REPORT *****
```

```
Total Agents moved: 19
```

```
Total Board read: 23
```

```
Total Board write: 9
```

```
Total Board delete: 3
```

```
Total Average delay time has been accumulated: 3
```

```
State STATE_AGENT_WORKING moved 14
```

```
State STATE_AGENT_FOUND_BH moved 5
```

```
Agent State STATE_AGENT_DONE count 1
```

```
Node n1 has been visited 3
```

```
Node n5 has been visited 2
```

```
Node n4 has been visited 3
```

```
Node n3 has been visited 5
```

```
Node n2 has been visited 5
```

```
Node n6 has been visited 1
```

```
Board at n0 has min Read accesses: 0
```

Figure 6.7: A Graph Comparison Results of the Protocol with Three Different Locations of Black Hole

Board at n0 has min Write accesses: 0

Board at n0 has max Read accesses: 0

Board at n4 has max Write accesses: 4

The text output shown in this case study is just an example to demonstrate the ideas that the DisJ simulation system can capture the simulation activity and information from the network environment and the protocol execution, and presents it to the user. Therefore, we leave further detail of implementation of presentation for future works.

6.2.6 The Results Analysis

We simulated the Co-Locate Black Hole Search protocol several times with different locations of a black hole and a home base, and different sizes of Ring network. As shown in Figure 6.7, we categorize the simulation results into three different cases: i) a location of a black hole is exact opposite a home base, ii) a location of a black hole is beside a home base, and iii) a location of a black hole is randomly selected but it is not fall into one of the prior two cases. The simulation results show that the first case where the protocol requires less number of moves to complete the task than other cases. The following paragraphs discuss a brief detail of analysis of each case based on the assumptions stated earlier and the results provided by DisJ.

A location of a black hole is exact opposite a home base

The idea of the protocol is to perform an exploration of the network in stages that are equally divided between two agents at each stage. At stage one the whole network is unexplored except the home base, therefore, there are two zones that cover the whole network and one of the zone contains a black hole. The zone that contains a black hole will have a black hole located at the end of the zone. The agent that explore the zone that contains a black hole will die at the last node of the zone. The another agent will finish another zone safely and traverses back to find a last safe node that were explored by the dead agent. Once the agent reaches a last safe node of the dead agent, it realizes that the number of safe nodes is equal to the size of the network minus one. Then, the black hole is located between a last safe node of the dead agent and his last safe node. Therefore, the protocol in this case requires only one stage of exploration and the number of moved required by the protocol is minimum. Figure 6.8 shows a Whiteboard of the last safe node of the dead agent where another agent arrives and found that the whole network has been explored and a black hole is found

Node ID	isAlive	Node State	# Token	# A	Whiteboard
n30	True	STATE_NODE_CLEAN	0	0	safe:left active:right
n32	False	STATE_NODE_UNKNOWN	0	0	
n31	True	STATE_NODE_TO_BH	0	0	
n34	True	STATE_NODE_CLEAN	0	0	
n33	True	STATE_NODE_TO_BH	0	1	
n35	True	STATE_NODE_CLEAN	0	0	

Figure 6.8: A snapshot of the DisJ view that show the Whiteboard of a last safe node of the dead agent in an ordered ID Ring topology of size 50, where a home base is at n7 and a back hole is exact opposite the home base, n32

during the first stage.

A location of a black hole is beside a home base

In this case, a black hole is located at a first unexplored node of one of a zone, which means at first stage an agent of the zone that contains a black hole dies at first move and never explored the zone at all. The alive agent will have to explore the entire network in various stages. At first stage the alive agent explores one half of the network, the second stage the agent explores one forth, third stage the agent explores one eighth and so on until the agent explored all the nodes in the network except one. Each stage is initialized at a last safe node of the dead agent, in this case, the home base. The alive agent has to repeatedly travel back and forth between the home base and the starting node of a new zone of each stage. Figure 6.9 shows there are six stages required by the protocol to finish the task for the network size of 50 nodes. Therefore, the number of moves required by the protocol to complete the task in this case is maximum.

Agent View	Node View	Statistic View			
Node ID	isAlive	Node State	# Token	#	Whiteboard
n5	True	STATE_NODE_TO_BH	0	1	active:left safe:right
n4	True	STATE_NODE_CLEAN	0	0	round:2:12
n3	True	STATE_NODE_CLEAN	0	0	round:3:6
n2	True	STATE_NODE_CLEAN	0	0	round:4:3
n9	True	STATE_NODE_CLEAN	0	0	round:5:1
n8	True	STATE_NODE_CLEAN	0	0	round:6:1
n7	True	STATE_NODE_TO_BH	0	0	
n6	False	STATE_NODE_UNKNOWN	0	0	

Figure 6.9: A snapshot of the DisJ view that show the Whiteboard of a last safe node of the dead agent in an ordered ID Ring topology, where a home base is at n7 and a back hole is besides the home base, n6

A location of a black hole is randomly selected

In this case the number of stages required by the protocol is based on how close the location of a black hole is located to the home base because the number of nodes explored by a dead agent is less when a black hole is closer to a home base. Hence, the another agent has to create many stages to explore the left over unexplored nodes of the dead agent. For examples, the maximum number of stages is may equal to the cases where the black hole is beside the home base, and the minimum number of stages is equal to two where the location of a black hole is beside a node that located exact opposite a home base. The graph shows in Figure 6.7 illustrates that the number of moves is closed to the maximum rather than the minimum because each extra stage an agent has to traveling almost around the ring of the network. Also at each stage the number of unexplored nodes is divided equally into two zones until the zone contains only one unexplored node. This means the protocol requires only two stages if a black hole is located next to a node that located exact opposite a home base. Otherwise, at least three stages is required (see Figure 6.10). Therefore, the number of moves of this case is more likely closer to a maximum requirement

Agent View	Node View	Statistic View				
Node ID	isAlive	Node State	# Token	# A	Whiteboard	
n29	True	STATE_NODE_CLEAN	0	0	safe:left	
n26	False	STATE_NODE_UNKNOWN	0	0	active:right	
n27	True	STATE_NODE_TO_BH	0	1	round:2:3	
n24	True	STATE_NODE_CLEAN	0	0	round:3:2	
n25	True	STATE_NODE_TO_BH	0	0	round:4:1	
n10	True	STATE_NODE_CLEAN	0	0		

Figure 6.10: A snapshot of the DisJ view that show the Whiteboard of a last safe node of the dead agent in an ordered ID Ring topology of size 50, where a home base is at n7 and a back hole is at n26

than a minimum requirement.

6.3 A Black Hole Search algorithm for the Agent with Token Model

6.3.1 Problems

The section discusses the same problem as discussed in previous section, a black hole search. However, this section presents different communication model, called the Agent with Token model.

6.3.2 The Assumptions of the Network Environment

The black hole search algorithm that discussed in this section has the following assumptions and restrictions for the environment as follow.

- The network topology is a ring topology
- The communication links are bi-directional links
- The communication links are total reliable

- The transmission order of the links is uniformed random order
- The transmission delay is uniformed random
- There exists a global unique ID assigned to each and every agent
- There exists a global a unique ID assigned to each node in the network
- There are exact two agents involve in the process and all start at the same location called home base.
- The agents know total number of agent involve in the process
- The agents know the size of network
- There is only one fixed black hole exists in the network
- The agent can carry at most one token at a time

6.3.3 The Pseudo Code

The black hole search algorithm in a ring topology with the Agent with Token model shows in this example is called Ping Pong algorithm presented in [22]

PROTOCOL: Ping Pong

METARULES:

1. An agent always ensures that a token is lying a node u before going to node v on link $\{u, v\}$
2. An agent never enter an unknown link $\{u, v\}$ from u to v if there is a token on u unless it was dropped by the agent before it entered link $\{u, v\}$ to explore v .

Start with:

Agent A_1 and A_2 start at the same node, say n_0 , each agent has two variables called LastRight and LastLeft, which store a last safe node on the right and last safe node on the left that it had visited respectively.

1. At the beginning, one agent explores the right side of n_0 , say A_1 while another explores the left side of n_0 , say A_2 .
2. The agent set its state to be Explore Left or *Explore Right* according to the side of n_0 that it is exploring.
3. The agent exploring on the right by using *Cautious Walk in a Ring* procedure.
4. The agent exploring on the left by using *Double Cautious Walk in a Ring* procedure.
5. The agent with state *Explore Right* exits procedure *Cautious Walk in a Ring* at
 - (a) The agent exited at 1.a, the black hole is found, exit the protocol
 - (b) The agent exited at 1.b, the agent found another agent at state *Explore Right*. So, the agent changes its state to be *Explore Left*. Then go to its LastLeft and drops its token and traverses back to the right to pickup another token in order to perform *Double Cautious Walk in a Ring* at the LastLeft.
 - (c) The agent existed at 2.b.i, the agent changes its state to be *Explore Left* and starts explore to the left at it LastLeft by dropping its token at the LastLeft and traverses to the right to get another token.

6. The agent with state *Explore Left* exists procedure *Double Cautious Walk in a Ring* at
 - (a) The agent exited at 1, the agent is not allowed to perform *Double Cautious Walk in a Ring*
 - (b) The agent exited at 3.b.ii, the agent changes its state to be *Explore Right* and traverse back to its LastLeft to pick up its token and continue explore right from it LastRighth.
7. The end the protocol

PROCEDURE: Cautious Walk in a Ring

1. An agent moves from node u to v via an unexplored right port r by dropping a token on node u , once arrived at node v , update the LastRight to be v
 - (a) If counting number of node from the LastLeft to the LastRight is equal to $|N - 1|$ then exit the procedure
 - (b) If the agent found another token exist at v then exit the procedure
 - (c) Otherwise, returns to u immediately via port r .
2. Once the agent return to node u
 - (a) If there exist a token, the agent pickup the token and return to node v via the port r .
 - (b) Otherwise, the agent keeps traversing to the left of u until it found a token. While traversing to the left, the agent keeps updating the LastLeft if it passed the existing the LastLeft. Once a token is found, return to it LastRight. While traversing back to the LastRight

- i. If found another token at the LastRight, exit the procedure.
 - ii. Otherwise continue Step1.
- 3. The agent moves back to v with the token and resume Step1 on node v .
- 4. The end of the procedure

PROCEDURE: Double Cautious Walk in a Ring

- 1. If there exist a token at node u and the agent is at state *Explore Left*, and the agent is carrying a token, and the LastLeft of the agent is u . Otherwise exist procedure.
- 2. The agent moves from node u to v via an unexplored left port l with its token.
- 3. If the agent survives, drops its token at node v , updates the LastLeft node to be v , and return to u via the port l .
 - (a) If there is another token at u , pick it up and return to v via port l and continue Step1.
 - (b) Otherwise, the agent traverses to the right side of node u until it found another token or it reached the LastRight node.
 - i. If it found another token, pick it up and traverse back to the LastLeft node and continue Step1.
 - ii. Otherwise exit the procedure
- 4. The end of the procedure

6.3.4 The Implementation of Ping Pong Protocol in Java with DisJ

The following Java codes show a Java Class definition of the Ping Pong protocol for the Mobile Agent with Token model. The implemented Class, PingPong Class, must extend TokenAgent Abstract Class in order to access the Token communication APIs and to notify the simulation system to create a Mobile Agent with Token processor to simulate the protocol. The TokenAgent Abstract Class represents a main computing entity, in this case a mobile agent that performs computations of the algorithm in the Mobile Agent with Token model.

```
public class PingPong extends TokenAgent {
```

The PingPong Class requires states for both agent and node objects in which it will be used for the computations and the simulation presentations. The user must define the state of the objects in respect to the rules, which includes the variable visibility must be public, the variable type must be an integer, and the variable name of the state should begin with the word state that is non-case sensitive. The state variables can be used for both agent and node in the network topology. The following Java codes are examples of defining variables for the entity object.

```
/**
 * Initiation function for an agent
 */
public void init() {
String p = this.getLeftPort();
if(this.getNodeState() != STATE_EXPLORED){
// i am a Righty agent
```

```
this.setNodeState(STATE_EXPLORED);  
this.dropToken();  
this.righty = true;  
p = this.getAnotherPort(p);  
  
}else{  
    // i am a Lefty agent  
    this.righty = false;  
}  
  
this.numExplored = 1;  
this.lastLeft = this.lastRight = this.getNodeId();  
  
// starts exploring  
this.become(STATE_AGENT_FORWORD);  
this.moveTo(p);  
}
```

In the function `init()` shown above, there are examples of internal functions that the agent may use in the protocol, the function `this.dropToken()` that allows the agent to drop a Token in a node that the agent currently resides in, and the function `this.moveTo(p)` that transmits the agent out through a given outgoing port of the node that currently resides in. The `TokenAgent` Abstract Class also provides another abstract function called `arrive()`, for the user to implement. The function is used when the agent arrives at a node. The function provides a parameter called `port` which is an incoming port of the node that the agent is entered. The following Java codes show the definition of the function `arrive()` for the `TokenAgent` Abstract Class.

```
public void arrive(String port)
```

The `TokenAgent` Abstract Class provides internal API functions for the user to access agent and node objects information, for example, the function `this.become(STATE_AGENT_FORWORD)` allows the user to set a current state of the agent object that calls the function. The function `this.getOutPorts()` provides a list of outgoing ports of a node that the agent currently resides in. Also the `TokenAgent` Abstract Class provides two others abstract functions for the user to implement are the `alarmRing()` and the `notification()` functions. The `alarmRing` function is called when the internal alarm clock of an agent rings. The `notification()` function allows the agent to register activities occurring within a node that the agent currently resides in (e.g. another agent arrives at the node, another agent drops a Token at the node, etc.). In the Ping Pong protocol, the `alarmRing()` and `notification()` functions are not required. Therefore, the user may leave them as empty functions as shown in the following Java codes.

```
public void alarmRing() {  
  
}  
  
public void notified(NotifyType type) {  
  
}
```

For the full detail of the implementation of the Ping Pong protocol is available in Appendix C, and the full detail of the usage of the DisJ system is available in the user manual of DisJ in Appendix E. Figure 6.11, Figure 6.12, and Figure 6.13

Figure 6.11: A snapshot of DisJ during the simulation of the Ping Pong protocol

are snapshots of the DisJ simulation system during the simulation of the Ping Pong protocol.

6.3.5 The Simulation Results

The following are the simulation results example that DisJ provide in text format in the Eclipse console.

```
*****Simulation for token_pp.gph is successfully over.*****
```

```
***** STATISTIC REPORT *****
```

```
Total Token has been picked by all agents: 4
```

```
Total Token has been dropped by all agents: 5
```

Figure 6.12: A snapshot of the Node View during the simulation

Figure 6.13: A snapshot of graphical report during the simulation

Total Average delay time has been accumulated: 77

State STATE_AGENT_FORWORD moved 5 time(s)

State STATE_AGENT_PICKUP moved 2 time(s)

State STATE_AGENT_RETURN moved 2 time(s)

State STATE_AGENT_CONVERT moved 2 time(s)

State STATE_AGENT_RESET moved 2 time(s)

State STATE_AGENT_DB_PICKUP moved 3 time(s)

State STATE_AGENT_DB_RETURN moved 1 time(s)

State STATE_AGENT_MARKING moved 5 time(s)

Agent State STATE_AGENT_MARKING count 1

Node n1 has been visited 1 time(s)

Node n0 has been visited 3 time(s)

Node n5 has been visited 5 time(s)

Node n4 has been visited 6 time(s)

Node n3 has been visited 3 time(s)

Node n6 has been visited 4 time(s)

Agent 1 hold 0 token(s)

The output shown in this case study is just an example to demonstrate the ideas that the DisJ simulation system can capture the simulation activity and information from the network environment and the protocol execution, and presents it to the user. Therefore, we leave further detail of implementation of presentation for future works.

6.3.6 The Results Analysis

We simulated the Ping Pong protocol several times with different locations of a black hole and a home base, and different sizes of Ring network. As shown in Figure 6.14, we categorize the simulation results into four different case: i) a location of a black hole is exact opposite a home base, ii) a location of a black hole is randomly located on the path of a left agent, iii) a location of a black hole is randomly located on the path of a right agent, and iv) a location of a black hole is next to a home base. The simulation results show that the second case requires the most number of moves to complete the task and the forth case require less number of moves than any other cases. However all cases have the same trend of the number of moves required by the protocol in different simulations and network sizes. The following paragraphs discuss a brief analysis based on the assumptions stated earlier and the results provided by DisJ.

The graph in Figure 6.14 shows that the number of moves performed by agents is not much different among case i, ii, and iii because of three observations mentioning in the following

1. Note that a right agent performs "Cautious Walk" and a left agent performs "Double Cautious Walk", which mean most of a left agent moves are to chase a second token from a right agent.
2. The left agent has done very little exploration on the left side of the ring until a right agent dies in a black hole.
3. Note that the speed and order of transmission in the network links are uniformed random. A left agent maybe able to obtain a right agent token when a left agent moves faster than a right agent. Once a left agent obtains a second token, it

Figure 6.14: A Graph Comparison Results of the Protocol with Three Different Locations of Black Hole

has to travel all the way back to it last explored node on the left side of the ring. Also, once a right agent found it token has been missing, it has to travel all the way back in opposite direction until a token is found.

These three observations support the claim we made earlier to the first three cases, especially the third observation that dictates how different the number of moves of each simulation are. The distance of a black hole from a right agent defines the number of chances for a left agent to steal a right agent token. The higher number of stealing that a left agent makes, the higher number of moves requires by the protocol to complete the task. Therefore, the forth case where a black hole locates next to a home base (regardless of left or right) is the case that requires minimum the number of moves for the protocol to complete the task, because there is no token chasing. There will be only one agent performs an exploration. A right agent performs "Cautious Walk" all the way if a black hole is next to a home base to the left, otherwise a left agent performs "Double Cautious Walk" all the way if a black hole locates on the next to a home base to the right. A survived agent will find a black hole without itself get killed because it explored every single node in the network and at the last safe node it found that the number of explored nodes equal to the size of network minus one. The number of moves the protocol requires for an exploration in the forth case is computable as $((|network| - 3) \times 2) + (|network| - 2)$ plus the number of moves for marking both nodes that connected to a black hole, if required, is $|network| - 2$.

Chapter 7

Conclusion

In this thesis we have investigated several different existing distributed simulation systems and found that they are limited in term of usability, capability, and efficiency etc. For example, each existing system currently supports only the Message Passing model and does not have adversary supports. Therefore, we introduce the DisJ simulation system that is an independent platform discrete event simulation system that helps the user to develop, test and verify the reactive distributed algorithms more efficiently and cost effectively. In addition, unlike the existing systems, DisJ supports three reactive distributed communication models for a fixed Netscape environment e.g. the Message Passing model, the Mobile Agent with Whiteboard model, and the Mobile Agent with Token model. DisJ is an Eclipse Plug-in that includes the Java IDE and the topology editor, which allows the user to develop a reactive distributed algorithm in the Java language, and to create the network topology needed for a simulation of the algorithm. During a simulation of an algorithm, DisJ provides animation playback that shows the progress of the algorithm, and allows the user to inject an adversary interruption program to verify the algorithm. At the end of the simulation, DisJ generates reports of the simulation for the user to review, and allows the user to replay past simulations from recorded files without any alteration of the

simulation.

During the development of the DisJ simulation system we found that our development approach, i.e. the development of the simulation system as an Eclipse plug-in, allowed us to reuse many existing codes and plug-in applications in the simulation system, which saved us a lot of development and coding time. For example, the simulation system used, the Eclipse UI workbench, the Java JDT editor, the graphical framework editor, etc. Another interesting thing we found during the development of the simulation system was that the adaptation of the simulation system to various needs of the user is very important because most of the algorithms that the user wants to simulate are already available and have been studied and analyzed in theory. Therefore, the main expectation of the user has to the simulation system is the testing and verification of the algorithms. Examples from the case studies that we used for testing the DisJ simulation system demonstrated the remarkable utility of the system, especially for the case study of ring election using the As Far protocol with adversary script. This case study shows that the simulation results of the protocol are able to verify the correctness of the analysis of the protocol in the worst case scenario by providing results that match the results from the analysis of the protocol with less work and time required from the user.

Currently DisJ supports only the Netscape environment that is commonly used and known in academic communities and industrial applications. Also the implementation of the simulation presentations that have been currently implemented are for demonstrating the ideas and benefits of the collected information that are useful for the user. There is a lot of potential for DisJ to be enhanced and improved to go beyond this. For example we could enhance DisJ to support the Sensor network environment, expand user friendly interfaces, and provide more sophisticated reports and presentations, etc. These enhancements and improvements require more resources

and time for research and development than is the available for this thesis. These may be considered in future studies and developments.

List of References

- [1] T. Abdelzaher, M. Raynal, and N. Santoro. *Principle of Distributed System*. Springer, 2009.
- [2] M. Akhbarizadeh and M. Nourai. Hardware-based ip routing using partitioned lookup table. *IEEE/ACM Transactions on Networking (TON)*, 13(4):769–781, 2005.
- [3] C. Aniszczyk and R. Hudson. Create an eclipse-based application using the graphical editing framework. *GEF Documentation*, 2003.
- [4] U. Bandi, M. Dasaka, and P. Kumar. Design in reliability for communication designs. In *DAC '06 Proceedings of the 43rd annual Design Automation Conference*, 2006.
- [5] M. Ben-Ari. Distributed algorithms in java. In *ITiCSE '97 Proceedings of the 2nd conference on Integrating technology into computer science education*, pages 62–64, 1997.
- [6] M. Ben-Ari. Interactive execution of distributed algorithms. *ACM Journal of Educational Resource of Computing*, 1(2), 2001.
- [7] G. Booch. Object-oriented design. *ACM SIGAda Ada Letters*, 1(3), 1982.
- [8] P. Bracchi, B. Cukic, and V. Cortellessa. Performability modeling of mobile software systems. In *ISSRE '04 Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.
- [9] P. Bresciani, A. Perini, P. Giorgini, F. Giuchiglia, and J. Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. In *AGENTS '01 Proceedings of the fifth international conference on Autonomous agents*, 2001.

- [10] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5), 2000.
- [11] N. Sharygina J. Browne and R. Kurshan. A formal object-oriented analysis for software reliability: Design for verification. In *FASE '01 Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, 2001.
- [12] D. Carlson. *Eclipse Distilled*. Addison-Wesley Professional, 2005.
- [13] J. Chalopin, S. Das, and N. Santoro. Rendezvous of mobile agents in unknown graphs with faulty links. *DISTRIBUTED COMPUTING Lecture notes in Computer Science*, 4731, 2007.
- [14] J. Chalopin, E. Godard, Y. Metivier, and R. Ossamy. Mobile agent algorithms versus message passing algorithms. *PRINCIPLES OF DISTRIBUTED SYSTEMS 10th International Conference, OPODIS*, 4305, 2006.
- [15] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):28–283, 1979.
- [16] I. Chatzigiannakis, A. Kinalis, A. Poulakidas, G. Prasinos, and C. Zaroliagis. Dap: A generic platform for the simulation of distributed algorithms. In *ANSS '04 Proceedings of the 37th annual symposium on Simulation*, 2004.
- [17] E. Curry and P. Grace. Flexible self-management using the model-view-controller pattern. *IEEE Software*, 25:84–90, 2008.
- [18] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Effective elections for anonymous mobile agents. *ALGORITHMS AND COMPUTATION Lecture notes in Computer Science*, 4288, 2006.
- [19] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile search for a black hole in an anonymous ring. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 166–179, 2001.
- [20] G. Fishman. *Discrete-event simulation: modelling, programming, and analysis*. Springer, 2001.
- [21] D. Flanagan. *Java in a Nutshell*. O'Reilly, 2002.

- [22] P. Flocchini, D. Ilcinkas, and N. Santoro. Ping pong in dangerous graphs: Optimal black hole search with pure tokens. *DISTRIBUTED COMPUTING Lecture Notes in Computer Science*, 5218:227–241, 2008.
- [23] P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk. Multiple mobile agent rendezvous in a ring. *LATIN 2004: THEORETICAL INFORMATICS Lecture notes in Computer Science*, 2976, 2004.
- [24] P. Flocchini, B. Mans, and N. Santoro. Sense of direction in distributed computing. *Theoretical Computer Science - Special issue: Distributed computing*, 291(1-4), 2003.
- [25] P. Flocchini, A. Roncato, and N. Santoro. Computing on anonymous networks with sense of direction. *Theoretical Computer Science*, 301(1-3), 2003.
- [26] P. Flocchini and N. Santoro. Topological constraints for sense of direction. *International Journal on Foundations of Computer Science*, 9(2), 1998.
- [27] Corporate The MPI Forum. Mpi: a message passing interface. In *Supercomputing '93 Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, 1993.
- [28] Message Passing Interface Forum. Mpi: A message-passing interface standard (version 2.2). *MPI Documents*, 2009.
- [29] The Eclipse Foundation. Platform plug-in developer guide. *Eclipse 3.1 Documentation*, 2005.
- [30] R. Fujimoto. Parallel discrete event simulation. In *WSC '89 Proceedings of the 21st conference on Winter simulation*, 1989.
- [31] R. Fujimoto. Parallel and distributed simulation. In *WSC '99 Proceedings of the 31st conference on Winter simulation*, 1999.
- [32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [33] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

- [34] R. Gibson. *Object Oriented Technologies: Opportunities and Challenges*. Addison-Wesley, 1999.
- [35] A. Gokhale. *Introduction to Telecommunications*. Delmar Publisher, 2004.
- [36] E. Gomez-Martinez, S. Ilarri, and J. Merseguer. Performance analysis of mobile agents tracking. In *WOSP '07 Proceedings of the 6th international workshop on Software and performance*, 2007.
- [37] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The java language specification, java se 7 edition. *Java Language Specification*, 2012.
- [38] J. Gosling and H. McGilton. The java language environment. *Java White Papers*, 1996.
- [39] W. Grosso. *Java RMI*. O'Reilly, 2001.
- [40] Distributed Computing Group. Sinalgo - simulation for network algorithms. 2007.
- [41] A. Hac and Y. Huang. Location update and routing scheme for a mobile computing environment. *International Journal of Network Management*, 10(4):191–214, 2000.
- [42] F. Ingelrest, G. Barrenetxea, B. Schaefer, M. Vetterli, O. Couach, and M. Parlange. Sensorscope: Application-specific sensor network for environmental monitoring. *ACM Transactions on Sensor Networks (TOSN)*, 6(2), 2010.
- [43] Object Technology International. Eclipse platform technical overview. *Eclipse Corner Articles*, 2001.
- [44] L. Ivanovl. A modern course on parallel and distributed processing. *Journal of Computing Sciences in Colleges*, 21(6), 2006.
- [45] W. Jia and W. Zhou. *Distributed network systems: from concepts to implementations*. Springer, 2004.
- [46] Z. Jin and R. Gupta. Improving the speed and scalability of distributed simulations of sensor networks. In *IPSN '09 Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, 2009.
- [47] D. Johan. Take control of your properties. *Eclipse Corner Article*, 2003.

- [48] M. Johnson. Analysis of routing table update activity after resource failure in a distributed computer network. In *SIGCOMM '83 Proceedings of the symposium on Communications Architectures and Protocols*, pages 14–20, 1983.
- [49] M. Karagiannis, I. Chatzigiannakis, and J. Rolim. Wsnge: a platform for simulating complex wireless sensor networks supporting rich network visualization and online interactivity. In *SpringSim '09 Proceedings of the 2009 Spring Simulation Multiconference*, 2009.
- [50] A. Kiezun. Java development user guide. *JDT Documentation*, 2005.
- [51] G. King. Making the most of statistical analyses: Improving interpretation and presentation. *American Journal of Political Science*, 44(2), 2000.
- [52] X. Li, H. Frey, N. Santoro, and I. Stojmenovic. Focused coverage by mobile sensor networks. In *MASS 6th IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, 2009.
- [53] X. Li and N. Santoro. Zoner: A zone-based sensor relocation protocol for mobile sensor networks. In *WLN'06 Proc 6th IEEE Workshop on Wireless Local Networks*, 2006.
- [54] Y. Li and M. Thai. *Wireless Sensor Networks and Applications*. Springer, 2008.
- [55] S. Liang. *The Java Native Interface Programmer's Guide and Specification*. Addison-Wesley, 1999.
- [56] D. Magoni and J. Pansiot. Influence of network topology on protocol simulation. *NETWORKING ICN 2001 Lecture notes in Computer Science*, 2093, 2001.
- [57] K. Mehlhorn and S. Naher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [58] R. Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly, 2006.
- [59] H. Narayan, R. Govindan, and G. Varghese. The impact of address allocation and routing on the structure and implementation of routing tables. In *SIGCOMM '03 Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 125–136, 2003.
- [60] R. Norman. *Object-Oriented System Analysis and Design*. Prentice-Hall, 1996.

- [61] S. Northover and M. Wilson. *SWT: The Standard Widget Toolkit, Volume 1*. Addison-Wesley Professional, 2004.
- [62] R. Oechsle and T. Gottwald. Disaster (distributed algorithms simulation terrain): a platform for the implementation of distributed algorithms. In *ITiCSE '05 Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, 2009.
- [63] R. Otte, M. Roy, and P. Patrick. *Understanding CORBA: Common Object Request Broker Architecture*. Prentice Hall, 1995.
- [64] S. Oualline. *Practical C++ Programming*. O'Reilly, 2003.
- [65] C. Raghavendra, K. Sivalingam, and T. Znati. *Wireless Sensor Networks*. Springer, 2006.
- [66] E. Rechtin. *Systems Architecting: Creating & Building Complex Systems*. Prentice-Hall, 1991.
- [67] A. Reddy. Java coding style guide. *Sun Microsystems Developers White Papers*, 2000.
- [68] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1), 1992.
- [69] N. Santoro. *Design and Analysis of Distributed Algorithm*. John Wiley and Sons, 2007.
- [70] X. Li N. Santoro and I. Stojmenovic. Localized distance-sensitive service discovery in wireless sensor networks. In *FOWANC'08 Proc. 1st ACM International Workshop on Foundations of Wireless Ad Hoc and Sensor Networking and Computing*, 2008.
- [71] W. Schreiner. A java toolkit for teaching distributed algorithms. In *ITiCSE '02 7th annual conference on Innovation and technology in computer science education*, pages 111–115, 2002.
- [72] W. Schreiner. Daj - a toolkit for the simulation of distributed algorithms in java. *A Toolkit for the Simulation of Distributed Algorithms in Java*, 2008.

- [73] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2004.
- [74] S. Soltani, K. Misra, and H. Radha. On link-layer reliability and stability for wireless communication. In *MobiCom '08 Proceedings of the 14th ACM international conference on Mobile computing and networking*, 2008.
- [75] P. Spirakis and C. Zaroliagis. *Distributed Algorithm Engineering*. Springer-Verlag, 2002.
- [76] J. Spragins. Reliability problems in data communications systems. In *SIGCOMM '77 Proceedings of the fifth symposium on Data communications*, 1977.
- [77] D. Springgay. Creating an eclipse view. *Eclipse Corner Article*, 2001.
- [78] V. Sunderam. Pvm: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.
- [79] S. Toh. *Cognitive and Motivational Effects of Two Multimedia Simulation Presentation Modes on Science Learning*. Ph.D dissertation, University of Science Malaysia, 1998.
- [80] D. van Hook and J. Claving. Execution logging and replay: issues and approaches. In *the Fall Simulation Interoperability Workshop*, 1997.
- [81] E. Velasquez. *Mobile Agents in Distributed Search: A Comparative Study*. M.C.S Thesis - Carleton University, 2001.
- [82] E. Velasquez, N. Santoro, and A. Nayak. A mobile agent prototype for distributed search. In *MATA '01 Proceedings of the Third International Workshop on Mobile Agents for Telecommunication Applications*, 2001.
- [83] E. Velazquez and N. Santoro. Distributed facility location for sensor network maintenance. In *MSN 5th International Conference on Mobile Ad-hoc and Sensor Networks*, 2009.
- [84] E. Velazquez and N. Santoro. Efficient robot-based energy maintenance in wireless sensor networks. In *5th IEEE International Symposium on Wireless Pervasive Computing*, 2010.
- [85] G. Wainer. *Discrete-Event Modeling and Simulation*. Taylor and Francis Group, 2011.

- [86] C. Wang and H. Cheng. Optimization of network configuration in large distribution systems using plant growth simulation algorithm. *IEEE Transactions on Power System*, 23(1), 2008.
- [87] S. Xenos. Inside the workbench a guide to the workbench internals. *Eclipse Corner Article*, 2005.
- [88] C. Yang, C. Chen, Y. Chang, K. Chung, and J. Lee. Streaming support for java rmi in distributed environments. In *PPPJ '06 Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 53–61, 2006.
- [89] J. Zelle. *Python Programming: an Introduction to Computer Science*. Franklin, Beedle and Associate, 2004.

Appendix A

As Far Protocol in Java

```
package test;

import distributed.plugin.runtime.IMessage;
import distributed.plugin.runtime.engine.Entity;

/**
 * As Far Ring election with
 * - smallest win
 * - unique ID
 * - one way electing message
 * - all node are initiators
 * - global orientation
 * - total reliability
 *
 */
public class AsFarRingElectMsgPass extends Entity {
```

```
public static final int STATE_UNKNOWN = 0;
public static final int STATE_CANDIDATE = 1;
public static final int STATE_PASSIVE = 2;
public static final int STATE_LEADER = 3;
public static final int STATE_FOLLOWER = 4;

protected static final String MSG_ELECTION = "election";
protected static final String MSG_LEADER = "leader";

private static final String DIRECTION = "right";

private int myId;
private int leaderId;

public AsFarRingElectMsgPass() {
    super(STATE_UNKNOWN);
}

private int getId(String name){
    String id = name.substring(1);
    return Integer.parseInt(id);
}

@Override
public void init() {
    this.myId = this.getId(this.getName());
}
```

```
this.leaderId = 99;

this.become(STATE_CANDIDATE);
this.sendTo(MSG_ELECTION, DIRECTION, this.myId);
System.out.println(this.myId + " sent " + MSG_ELECTION + " to "
+ DIRECTION + " with " + this.myId);
}

@Override
public void receive(String portLabel, IMessage msg) {
    String msgLabel = msg.getLabel();
    Object content = msg.getContent();

    if(this.getState() == STATE_CANDIDATE){
        this.iAmCandidate(msgLabel, content);

    } else if (this.getState() == STATE_PASSIVE){
        this.iAmPassive(msgLabel, content);

    } else if (this.getState() == STATE_LEADER){
        if(msgLabel.equals(MSG_LEADER)){
            // arrival of my throne notification
            // done and terminated
            System.out.println(this.myId + ": I am a leader!");
        }

        // ignore the rests
```

```
} else {  
    // should not happen!  
}  
  
}  
  
/*  
    * Perform candidate protocols  
    */  
  
private void iAmCandidate(String msgLabel, Object msg){  
    if(msgLabel.equals(MSG_ELECTION)){  
        int id = Integer.parseInt(msg.toString());  
        System.out.println(this.myId + " received msg " + msgLabel  
            + " with " + id);  
  
        if(this.myId > id){  
            // become a loser  
            this.become(STATE_PASSIVE);  
  
            // relay the message  
            this.sendToOthers(MSG_ELECTION, id);  
            System.out.println(this.myId + " relays " + MSG_ELECTION  
                + " with " + id);  
  
        }else if(this.myId < id){  
            // still in the race  
            // kill the msg
```



```
}else{
// my msg has returned
this.become(STATE_LEADER);
this.leaderId = this.myId;
System.out.println(this.myId + " becomes elected leader");

// send notification
this.sendTo(MSG_LEADER, DIRECTION, this.myId);
}
} else{
// not possible coz leader msg cannot issue
// until the election is done
}
}

/*
 * Perform a loser work
 */
private void iAmPassive(String msgLabel, Object msg){
if(msgLabel.equals(MSG_ELECTION)){
int id = Integer.parseInt(msg.toString());
System.out.println(this.myId + " received msg "
+ msgLabel + " with " + id);

if(this.myId < id){
// kill the msg
```

```
}else{

    // relay the message
    this.sendToOthers(MSG_ELECTION, id);
    System.out.println(this.myId + " relays " + MSG_ELECTION
        + " with " + id);
}

}else if(msgLabel.equals(MSG_LEADER)){
    // Note: this not even need coz of curMin record
    int leaderId = Integer.parseInt(msg.toString());
    this.leaderId = leaderId;
    this.become(STATE_FOLLOWER);

    // relay to others
    this.sendToOthers(MSG_LEADER, this.leaderId);
    System.out.println(this.myId + " relays " + MSG_LEADER
        + " with " + this.leaderId);

} else{
    // not possible coz there are only 2 types of message
}

}

@Override
public void alarmRing() {
}
```

}

Appendix B

Co-Locate Black Hole Search Protocol in Java

```
package test;

import java.util.List;

import distributed.plugin.runtime.engine.BoardAgent;

/**
 * Whiteboard:
 * A Black Hole search in bidirectional un-oriented Ring with
 * 2 co-locate agents, known N (number of node), known K
 * (number of agent), total reliability and FIFO
 *
 */
public class BHC extends BoardAgent {
```

APPENDIX B. CO-LOCATE BLACK HOLE SEARCH PROTOCOL IN JAVA184

```
public static final int STATE_NODE_UNKNOWN = 0;
public static final int STATE_NODE_CLEAN = 1;
public static final int STATE_NODE_TO_BH = 2;

public static final int STATE_AGENT_WORKING = 3;
public static final int STATE_AGENT_FOUND_BH = 4;
public static final int STATE_AGENT_DONE = 5;

private boolean explored;
private boolean confirm;
private boolean traverse;
private boolean forward;
private int round;
private int numReq;
private int numDone;

public BHC() {
    super(STATE_NODE_UNKNOWN);
    this.round = 0;
    this.numReq = 0;
    this.numDone = 0;
    this.confirm = false;
    this.explored = false;
    this.traverse = false;
    this.forward = false;
```

```

}
```

```

public void init() {
    this.become(STATE_AGENT_WORKING);
    this.setNodeState(STATE_NODE_CLEAN);

    this.round = 1;
    int o = (this.getNetworkSize() - 1)/2;
    int e = this.getNetworkSize() - o - 1;
    List<String> ports = this.getOutPorts();
    List<String> info = this.readFromBoard();
    String p = null;
    System.out.println("Board: " + info);
    if(info.isEmpty()){
        // first agent
        this.numReq = o;
        p = ports.get(0);
        this.appendToBoard("active:" + p);
    }else{
        // second agent
        this.numReq = e;
        String s = info.get(0);
        String[] v = s.split(":");
        for(int i =0;i < ports.size(); i++){
            p = ports.get(i);
            if(!p.equals(v[1])){
```

```

break;
}
}
this.appendToBoard("active:" + p);
}
// set for next exploring
this.moveTo(p);
}

/*
 * My first visit to this node, and it is the first
 * visit from this port
 */
private void myFirstVisit(String port){
// First arrived safely from safe node
this.appendToBoard("safe:" + port);

// Then go back to say that this node also safe
this.explored = true;
this.confirm = true;
this.moveTo(port);
}

/*
 * Confirming port check to a previous node that this port
 * is safe then get back to the port to finish of the work

```

```

*/
private void confirmVisit(List<String> info, String port){
    String s = null;
    String[] v = null;
    for(int i =0; i < info.size(); i++){
        s = info.get(i);
        v = s.split(":");

        if(v.length == 2){
            // get my port status
            if(v[1].equals(port)){
                if(v[0].equals("active")){
                    this.removeRecord(s);
                    this.appendToBoard("safe:" + port);
                }
            }
        }else{
            // someone left a note
            // reset my work plan
            int share = Integer.parseInt(v[2]);
            this.numReq = this.numReq - share;
            this.round = Integer.parseInt(v[1]);
            this.removeRecord(s);
        }
    }

    // back to the port

```



```
this.confirm = false;
this.numReq--;
this.moveTo(port);
}

public void arrive(String port) {

    // traversing back
    if(this.traverse == true){
        this.traversBack(port);
        return;
    }

    // move to my current end territory
    if(this.forward == true){
        this.toMyEnd(port);
        return;
    }

    // keep exploring
    List<String> info = this.readFromBoard();
    if(info.isEmpty()){
        // I am the first to this node
        this.myFirstVisit(port);
        this.setNodeState(STATE_NODE_CLEAN);
```

```

}else {
String s = null;
String[] v = null;

if(this.confirm == true){
this.confirmVisit(info, port);
return;
}

System.out.println("Agent: " + this.getAgentId() + " numReq " + numReq);
// my second visit after confirm safe port to previous node
if(this.explored == true){
// check my round
if(numReq == 0){
// complete round, go backward to find partner
//current location
System.out.println("Start traversing back from node: " + this.getNodeId());
System.out.println("whiteboard: " + this.readFromBoard());
this.traverse = true;
this.numDone = 0;
String p = this.getAnotherPort(port);
this.traversBack(p);
return;
}

if(info.size() == 1){
// still the first at this node

```

```

s = info.get(0);
v = s.split(":");
if(v[1].equals(port)){
if(v[0].equals("active")){
// should not happen, i just marked safe on my first arrival
System.err.println("should not happen 1");
}

// find another port to explore
List<String> ports = this.getOutPorts();
String p = null;
for(int i =0; i < ports.size(); i++){
p = ports.get(i);
if(!p.equals(port)){
this.appendToBoard("active:" + p);

// move to new port
this.explored = false;
this.moveTo(p);
}
}
}else{
// should not happen
// i just visited on my first arrival
System.err.println("should not happen 2");
}

```

```

}else{
// other has visited this node from another port
for(int i =0; i < info.size(); i++){
s = info.get(i);
v = s.split(":");
if(v[0].equals("active")){
if(v[1].equals(port)){
// should not happen
// a port that i came from is safe
System.err.println("should not happen 3");
}else{
// someone is exploring it right now!
// so wait.
this.registerHostEvent(NotifyType.BOARD_UPDATE);
}
}else {
if(v[1].equals(port)){
// my port, ignore it

}else{
// another port that was explored and marked
// safe by other (No black hole)!!! or
// repeat my node territory (backward exploring)!!
// should not happen
System.err.println("should not happen 4");
break;

```

```

    }
    }
    }
    }

    } else {
        // my first visit but other has been here before
        this.myFirstVisit(port);
    }
    }
    }

    private void traversBack(String port){
        String p = this.getAnotherPort(port);
        this.numDone++;
        System.out.println("Travers back to port " + p);
        if(isSafe(p)){
            this.moveTo(p);

        }else{
            // we found under process node
            // done traversing
            this.traverse = false;

            // check whether is it a last safe node
            if(this.numDone == (this.getNetworkSize() - 1)){

```

```

// we found black hole
this.become(STATE_AGENT_FOUND_BH);

// the active port is a port to BH
this.setNodeState(STATE_NODE_TO_BH);
System.out.println("Agent: " + this.getAgentId() + " found BH from Node " + thi;

// move to my end node
this.numDone = 0;
this.forward = true;
this.toMyEnd(p);

}else{
// compute new share

int remain = this.getNetworkSize() - this.numDone;
System.out.println("Found the struggling node " + this.getNodeId() + " with rem:

int share = remain/2;
this.round++;
this.numReq = share;

// post to board that i will do extra share from my end
this.appendToBoard("round:" + round + ":" + share);

// move to my end node

```

```

this.numDone = 0;
this.forward = true;
this.toMyEnd(p);
}
}
}

```

```

private void toMyEnd(String port){
String p = this.getAnotherPort(port);
if(isSafe(p)){
this.moveTo(p);

}else{
// arrive at my end
this.forward = false;
System.out.println("Arrived my end Node: " + this.getNodeId() + " numReq " + th
if(this.getState() != STATE_AGENT_FOUND_BH){
// arrive at my end of previous round
List<String> info = this.readFromBoard();
String s = null;
String[] v = null;
System.out.println("Board " + info);
if(info.size() > 1){
for(int i = 0; i < info.size(); i++){
s = info.get(i);
v = s.split(":");

```

```

if(!v[0].equals("safe")){
    this.explored = false;
    this.confirm = false;
    // continue exploring
    System.out.println("Heading to " + v[0] + " port " + v[1]);
    this.moveTo(v[1]);
    break;
}
}
}else{
    this.explored = false;
    this.appendToBoard("active:" + p);
    this.moveTo(p);
}
}else{
    this.become(STATE_AGENT_DONE);
    this.setNodeState(STATE_NODE_TO_BH);
}
}
}

private boolean isSafe(String port){
    List<String> info = this.readFromBoard();
    boolean safe = false;
    String s = null;
    String[] v = null;

```



```

for(int i =0; i < info.size(); i++){
    s = info.get(i);
    v = s.split(":");
    if(v.length == 2){
        if(v[0].equals("safe") && v[1].equals(port)){
            safe = true;
            break;
        }
    }
}

return safe;
}

/*
 * Get opposite port from a given port in a ring
 */
private String getAnotherPort(String port){
    List<String> ports = this.getOutPorts();
    String p = null;
    for(int i =0; i < ports.size(); i++){
        p = ports.get(i);
        if(!p.equals(port)){
            return p;
        }
    }

    return p;
}

```

```
}
```

```
public void notified(NotifyType arg0) {
```

```
}
```

```
public void alarmRing() {
```

```
}
```

```
}
```

Appendix C

Ping Pong Protocol in Java

```
package test;

import java.util.List;

import distributed.plugin.runtime.engine.TokenAgent;

/**
 * Token:
 * A Black Hole search in bidirectional oriented Ring with known N,
 * known K = 2, agents are started at same home base, No FIFO,
 * 1 pebble/agent, total reliability and unique node ID.
 *
 * NOTE: a Righty performs Cautious walk to the right,
 * a Lefty performs Double Cautious walk to the left.
 *
 */
public class PP extends TokenAgent {
```

```
// Global states of node and agent

public static final int STATE_UNKNOWN = 0;
public static final int STATE_EXPLORED = 1;
public static final int STATE_TO_BH = 2;


public static final int STATE_AGENT_SLEEP = 3;
public static final int STATE_AGENT_FORWORD = 4;
public static final int STATE_AGENT_PICKUP = 5;
public static final int STATE_AGENT_SEARCH = 6;
public static final int STATE_AGENT_RETURN = 7;
public static final int STATE_AGENT_CONVERT = 8;
public static final int STATE_AGENT_RESET = 9;


public static final int STATE_AGENT_DB_PICKUP = 10;
public static final int STATE_AGENT_DB_RETURN = 11;
public static final int STATE_AGENT_RESET_LEFT = 12;
public static final int STATE_AGENT_CONVERT_RIGHT = 13;


public static final int STATE_AGENT_MARKING = 14;


// Instance variables
private boolean righty;
private int numExplored;
private String lastRight;
```

```
private String lastLeft;

public PP(){
    super(STATE_AGENT_SLEEP);
    this.righty = true;
    this.numExplored = 0;
    this.lastRight = null;
    this.lastLeft = null;
}

/*
 * Get me a left port of this node
 */
private String getLeftPort(){
    List<String> ports = this.getOutPorts();
    for(int i =0; i < ports.size(); i++){
        String p = ports.get(i);
        if(p.startsWith("l")){
            return p;
        }
    }
    return null;
}

/*
```

```
* Get an opposite port of a given port of this node
*/
private String getAnotherPort(String port){
    List<String> ports = this.getOutPorts();
    String p = null;
    for(int i =0; i < ports.size(); i++){
        p = ports.get(i);
        if(!p.equals(port)){
            return p;
        }
    }
    return p;
}

public void arrive(String port) {
    System.out.println(this.getAgentType() + " agent " + this.getAgentId()
        + " arrives at " + this.getNodeId());

    // black hole is found
    if(this.getState() == STATE_AGENT_MARKING){
        this.numExplored--;
        if(this.numExplored > 0){
            // continue traverses back to another end
            this.moveTo(this.getAnotherPort(port));
        } else {
            // found another node that connects to BH

```

```
// mark this node
this.setNodeState(STATE_TO_BH);
}
return;
}

if(this.getState() == STATE_AGENT_FORWORD){
if(this.righty){
// I am a Righty
this.setNodeState(STATE_EXPLORED);
this.numExplored++;
this.lastRight = this.getNodeId();

if(this.hasFoundBH()){
// found a node that connects to BH
this.become(STATE_AGENT_MARKING);

// mark this node
this.setNodeState(STATE_TO_BH);

// go back to another node that connects
// to BH
this.numExplored--;
this.moveTo(port);
return;
```

```
}
```

```
// found another token lies here and I am
```

```
// not yet explored the whole network
```

```
if(this.countHostToken() == 1){
```

```
// There is another Righty, I have to become a Lefty
```

```
this.become(STATE_AGENT_RESET_LEFT);
```

```
// go back to pick up my token
```

```
this.moveTo(port);
```

```
return;
```

```
}
```

```
// go back to pick up a token
```

```
this.become(STATE_AGENT_PICKUP);
```

```
this.moveTo(port);
```

```
} else {
```

```
// I am a Lefty
```

```
this.setNodeState(STATE_EXPLORED);
```

```
this.numExplored++;
```

```
this.lastLeft = this.getNodeId();
```

```
// Found a node that connected to BH
```

```
if(this.hasFoundBH()){
```

```
// found a node that connects to BH
```



```
this.become(STATE_AGENT_MARKING);

// mark this node
this.setNodeState(STATE_TO_BH);

// go back to another node that connects
// to BH
this.numExplored--;
this.moveTo(port);
return;
}

// drop a token
this.dropToken();

// go back and find 2nd token
this.become(STATE_AGENT_DB_PICKUP);
this.moveTo(port);
}

} else if(this.getState() == STATE_AGENT_PICKUP){
// a Righty comes to pickup its token
if(this.countHostToken() == 1){
// a token still here
this.pickupToken(1);

// return to the last right
```

```
this.become(STATE_AGENT_RETURN);

this.moveTo(port);

}else{

// a token is gone

this.become(STATE_AGENT_SEARCH);


// continue searching to the left
this.moveTo(this.getAnotherPort(port));
}

} else if(this.getState() == STATE_AGENT_SEARCH){

// a Righty continue searching for a token
if(this.countHostToken() == 1){
this.pickupToken(1);


// returning to the last right
this.become(STATE_AGENT_RETURN);
this.moveTo(port);
} else {

// continue searching to the left
this.moveTo(this.getAnotherPort(port));
}

} else if(this.getState() == STATE_AGENT_RETURN){
if(this.getNodeId().equals(this.lastRight)){

// a Righty arrives at lastRight
this.dropToken();
```

```
// continues explore
this.become(STATE_AGENT_FORWORD);
this.moveTo(this.getAnotherPort(port));
} else {
// continue journey to lastRight
this.moveTo(this.getAnotherPort(port));
}
} else if (this.getState() == STATE_AGENT_RESET_LEFT){
// a Righty has to become a Lefty
this.become(STATE_AGENT_CONVERT_RIGHT);

// NOTE: a token must be here, since another
// Righty has it own token at a lastRight of
// this agent
this.pickupToken(1);

// going back to lastLeft
this.moveTo(this.getAnotherPort(port));

} else if (this.getState() == STATE_AGENT_CONVERT_RIGHT) {
// a Righty prepares to convert to a Lefty
if(this.getNodeId().equals(this.lastLeft)){
// arrives at lastLeft
this.dropToken();

// a Righty becomes a Lefty
```

```
this.righty = false;

// go back and find 2nd token
this.become(STATE_AGENT_DB_PICKUP);
this.moveTo(port);
} else {
// a Righty continues going back to lastLeft
this.moveTo(this.getAnotherPort(port));
}

// ***** //
// ***** Lefty ***** //
// ***** //

} else if (this.getState() == STATE_AGENT_DB_PICKUP){
// a Lefty tries to find a 2nd token
if(this.countHostToken() == 1){
// found 2nd token
this.pickupToken(1);
```

```
// return to lastLeft
this.become(STATE_AGENT_DB_RETURN);
this.moveTo(port);
} else {
// still no 2nd token found
if(this.lastRight.equals(this.getNodeId())){
// it is a Lefty lastRight
// reseting to be a Righty
this.become(STATE_AGENT_RESET);

// return to lastLeft
this.moveTo(port);
} else {
// it is not a Lefty lastRight
// continue search for 2nd token
this.moveTo(this.getAnotherPort(port));
}
}
} else if (this.getState() == STATE_AGENT_DB_RETURN){
if(this.getNodeId().equals(this.lastLeft)){
// a Lefty arrives at lastLeft
if(this.countHostToken() == 1){
// a Lefty 1st token still here
this.become(STATE_AGENT_FORWARD);

// continue explores to the left with 2nd token
```

```
this.moveTo(this.getAnotherPort(port));
} else {
    // a Lefty 1st token is gone
    // drop a token
    this.dropToken();

    // go back and find another 2nd token
    this.become(STATE_AGENT_DB_PICKUP);
    this.moveTo(port);
}
} else {
    // a Lefty not yet arrives at its lastLeft
    // continue moving to the left
    this.moveTo(this.getAnotherPort(port));
}
} else if (this.getState() == STATE_AGENT_RESET){
    // a Lefty continues going to lastLeft
    if(this.getNodeId().equals(this.lastLeft)){
        // at lastLeft and found a token
        // NOTE: There always exists a token at lastLeft of Lefty
        // because Righty needs only one token, and it always
        // picks up a first found token
        this.pickupToken(1);

        // a Lefty is ready to convert to be a Righty
        this.become(STATE_AGENT_CONVERT);
```

```
// reset node counter
this.numExplored = 1;

// going to lastRight before converts to a Righty
this.moveTo(port);
} else {
// not at Lefty lastLeft
this.moveTo(this.getAnotherPort(port));
}
} else if (this.getState() == STATE_AGENT_CONVERT){
// a Lefty continues going to a lastRight
// still tracking number of explored nodes
this.numExplored++;

if(this.getNodeId().equals(this.lastRight)){
// arrives at lastRight
// become a Righty
this.righty = true;

// continue exploring to the right
this.dropToken();
this.become(STATE_AGENT_FORWORD);
this.moveTo(this.getAnotherPort(port));
} else {
// not at Lefty lastRight
```

```
this.moveTo(this.getAnotherPort(port));
}
}
}

public void notified(NotifyType type) {}

public void alarmRing() {}

/**
 * Initiation function for an agent
 */
public void init() {
    String p = this.getLeftPort();
    if(this.getNodeState() != STATE_EXPLORED){
        // i am a Righty agent
        this.setNodeState(STATE_EXPLORED);
        this.dropToken();
        this.righty = true;
        p = this.getAnotherPort(p);

    }else{
        // i am a Lefty agent
        this.righty = false;
    }
    this.numExplored = 1;
```



```
this.lastLeft = this.lastRight = this.getNodeId();

// starts exploring
this.become(STATE_AGENT_FORWORD);
this.moveTo(p);
}

/*
 * Agent identity printout function
 */
private String getAgentType(){
    if(this.righty){
        return "Righty";
    }else{
        return "Lefty";
    }
}

/*
 * Checking for every node has been explored except a BH
 */
private boolean hasFoundBH(){
    return (this.numExplored == this.getNetworkSize() - 1);
}
}
```

Appendix D

DisJ Plug-in Implementation in Java

See attached CD-ROM

Appendix E

DisJ User Manual: Cookbook

TABLE AND CONTENTS

CHAPTER 1. OVERVIEW	1
CHAPTER 2. PRELIMINARIES.....	2
CHAPTER 3. CREATING A TOPOLOGY.....	5
CHAPTER 4. CREATING A PROTOCOL.....	23
CHAPTER 5. RUNNING A SIMULATION.....	34
CHAPTER 6. GRAPHICAL AND STATISTICAL REPORTS.....	40
CHAPTER 7. CONCLUSION.....	43
CHAPTER 8. SAMPLE CODES	44

Chapter 1. Overview

This cookbook is a step-by-step description of the process of entering, executing and testing a reactive distributed algorithm using DisJ. It also describes how to use DisJ to simulate the algorithm and display the results in graphical interactive.

Here are four simple steps for using DisJ.

- 1 Install DisJ plug-in into Eclipse
- 2 Define a topology in DisJ graph editor
- 3 Write a protocol in Java language in Eclipse JDT
- 4 Execute the protocol in the defined topology using DisJ

Throughout the cookbook we will use two an example protocols; first protocol is “*As Far*” that represent message passing model for leader election in a ring with bidirectional links network, and second protocol is “*Black Hole Search*” that represent mobile agent with token model for counting number of nodes in a ring with bidirectional link network (see Chapter 8)

Chapter 2. Preliminaries

2.1 Install Eclipse

The DisJ is an Eclipse™ plug-in, therefore we need to have Eclipse™ (version 3.0 or higher) installed. Eclipse is an open platform built by an open source community, which it can be downloaded at <http://www.eclipse.org/downloads/>. There are many types of builds available, but we recommend “Eclipse RCP” or “Eclipse IDE for Java Developer” packages. This cookbook uses Eclipse version 3.4 as examples and demonstrations, different versions of Eclipse may varies a bit based on the versions used.

2.2 Install GEF libraries

Since DisJ graphical interface parts built on top of Graphical Editor Framework Plug-in (GEF), therefore, DisJ plug-in requires GEF Plug-in to be installed. GEF runtime library can be downloaded at <http://www.eclipse.org/gef/downloads/index.php>. There are many types of packages release, but we recommend “All-In-One” that includes Draw 2D, GEF and Zest. If you downloaded *Eclipse RCP* package, which might already include GEF plug-in.

2.3 Install DisJ

DisJ installation is similar to any others Eclipse Plug-ins by unzip the jar file and place the folder “org.carleton.scs.disJ_1.1.0” into *ECLIPSE_HOME/plugins* directory.

To make sure that the plug-in was successfully installed, restart Eclipse™ . Then, go to the menu *Help* → *About Eclipse SDK*

Then a dialog box shown below must pop up on the screen

Then click on “*Plug-in Details*” button and a dialog with list of current installed plug-ins of the Eclipse will be listed which included “*DisJ Simulator*” (if the installation has been done correctly) as shown in the picture below.

Note that any Plug-ins that has been installed into Eclipse successfully, the name of the Plug-ins will be listed in this dialog box, which include our GEF Plug-ins as well. Therefore, this is a good place to check and verify installed Plug-ins in your Eclipse.

2.4 Changing Perspective

After Eclipse has started a workbench is opened; the workbench can be viewed from many different perspectives; the default is the “*Resource Perspective*”. Since we are going to develop the distributed algorithm in Java™ it is better to switch to the “*Java Perspective*”. This will automatically forces user to follow the rules and best practices in software development with Java.

Click at icon at a top of your right hand to the list of available perspectives (as shown below).

Then select Java perspective as shown below

Chapter 3. Creating a Topology

3.1 Creating a Java project

For a best practice in software development, first we need to create a Java Project by right click at “*Package Explorer*” view, then select *New* → *Java Project*

The project creation wizard dialog box will pop up as below and type the project name in lower case (for a best practice)

Leave the default setting as it is then click *Next* → select *Libraries* tab

In order to add “DisJ” runtime libraries and API for developing algorithms, we will add “*disJ.jar*” which located at *ECLIPSE_HOME/plugins/org.carleton.scs.disj_1.1.0/* by clicking at “*Add External JARs...*” button and select *disJ.jar* from above location. The dialog box will look like the following. Then click “*Finish*” button.

The workbench under Java Perspective will look like the following.

3.2 Creating a Graph File (.gph)

Right click at the Java Project in “*Package Explorer*”, and select “*New → File*” (see below);

The a file creation dialog pops up as shown below, then fill up a file name with extension file type “.gph”

Then click “Finish”.

The workbench will look like the following

Then double click at newly created graph file, ring.gph, which will open up “DisJ Graph Editor” on the workbench and looks like the following

3.3 Creating a graph in DisJ Graph Editor

The DisJ Graph Editor shows up on the editor view. We start drawing by expanding the “*Palette*” on a top right corner of the editor tab by clicking at icon

And it looks like the following

There are two ways to define a graph:

- 3.3.1 Drawing Objects** - this is a basic drawing tool based on simple objects such as node, uni-directional link and bi-directional link.
- 3.3.2 Topology Template Library**- this provides a ready-made set of topologies that can be edited such as Ring, Tree, Spatial, Complete, Mesh, Hyper Cube, Arbitrary and Torus.

Creating a graph with topology editor is as simple as others drawing/graph editing applications, the user can choose to create a graph manually from drawing objects or use a readymade topology library template to generate a graph.

Here is an example of creating a bidirectional ring with 10 nodes using the topology template library.

Here is the snapshot when a Ring of size 10 is created.

Once, the graph has been drawn, the user can check if there is the correct number of links and nodes by opening a “*Properties View*”, which shows the current states and properties of the graph.

To open the “*Properties View*”, go to menu bar, select *Window->Show View->Properties*; if it does not list, select *Window->Show View->Others*.

Once the dialog shows up, expand a “*General*” folder and select “*Properties*”.

In addition to the graph properties, there are properties of the nodes and links which will be described later in detail. The following is snapshot of our workspace with properties view.

3.4 Element Properties

3.4.1 Graph Properties

Property	Value
G01 Total Node	10
G02 Total Link	10
G03 Total Messages Received	0
G04 Total Messages Sent	0
G05 Global Message Flow Type	FIFO
G06 Global Delay Type	Synchronous
G07 Global Delay Seed	1
G08 Protocol	
G09 Maximum Token Agent can Carr	1
G10 Number of Agent at Start	0
G11 Current Number of Agent	0

3.4.1.1 Graph Name (not editable)

This is the name of the file where the graph has been created.

3.4.1.2 Total Node

Total number of nodes in this graph

3.4.1.3 Total Link

Total number of links in this graph

3.4.1.4 Total Messages Received

Total number of transmissions that have been received from the start of the simulation until now

3.4.1.5 Total Messages Sent

Total number of transmissions that have been sent from the start of the simulation until now

3.4.1.6 Global Message Flow Type

Transmission flow type of every links in this graph, there are 2 types, FIFO and No Order. If there is more than one type in a graph, this field is set to "Mix Order"

3.4.1.7 Global Delay Type

Type of transmission delay of every links in this graph, there are 5 types, Synchronous, Random Uniformed, Random Poisson, Random Customs, Customs. If there is more than one type, this field is set to "Custom"

3.4.1.8 Global Delay Seed

A number of simulation time unit required for transmission on each link in the graph when it is in Synchronous. If there is more than one value, this field is set to "-1" (negative value is for indication purpose only).

3.4.1.9 Protocol

A name of protocol that currently being loaded into this graph

3.4.1.10 Maximum Token Agent Can Carry

A maximum number of tokens that agent in this graph can carry. This is used only in Agent with Token model

3.4.1.11 Number of Agents at Start

A number of agents exist in the graph at the beginning of simulation. This value is used only in Agent with Token and Agent with Whiteboard models

3.4.1.12 Current Number of Agents

A number of agents currently exist. This value is used only in Agent with Token and Agent with Whiteboard model

3.4.2 Node Properties

3.4.2.1 Node Name

A Name of selected node (editable)

3.4.2.2 User Input

Input text field that the user can input text value into the algorithm during the simulation. The algorithm can retrieve this input during the simulation but not guarantee the time of visibility of the input to the algorithm.

3.4.2.3 Initiator

This property specifies whether this node is an initiator of the simulation by setting a Boolean flag, true indicates this node is initiator otherwise false. There must be at least one node in a graph to be an initiator in order to start the simulation. The only initiator node executes function "init()" during the simulation and it will execute only once per simulation per initiator node.

3.4.2.4 Alive

This property indicates whether this node is alive (up and running) or shutdown (crashed)

3.4.2.5 Breakpoint

This property is used for debugging purpose. Once a node's breakpoint is enabled, the execution will be suspended as soon as a transmission arrives at the node. (Not yet support)

3.4.2.6 Number of Messages Received

Total number of transmissions that have been received by this node from the start of the execution of the protocol until now

3.4.2.7 Number of Messages Sent

Total number of transmissions that have been sent by this node from the start of the execution of the protocol until now

3.4.2.8 Outgoing Ports

A list of all outgoing ports available at this node (local labels)

3.4.2.9 Incoming Ports

A list of all incoming ports available at this node (local labels)

3.4.2.10 States Transition

This property reports the sequence of states transition of this node that has been changed from the start of the simulation until now

3.4.2.11 Number of Agent Hosted

Total number of agents that this node hosted has “Home Base” at the beginning of the simulation. This is used by Agent with Whiteboard and Agent with Token models only

3.4.2.12 Current Number of Agents

Total number of agents currently resides at this node. This is used by Agent with Whiteboard and Agent with Token models only

3.4.2.13 Current Number of Tokens

Total number of Tokens currently located at this node. This is used by Agent with Whiteboard and Agent with Token models only

3.4.3 Link Properties

3.4.3.1 Edge ID

An ID of this link

3.4.3.2 Type of Direction

The type of communication link (bi-directional or uni-directional)

3.4.3.3 Source Name

The name of a source node of this link

3.4.3.4 Source Port Name

The local source port's name of this link (editable)

3.4.3.5 Target Name

The name of target node of this link

3.4.3.6 Target Port Name

The local target port's name of this link (editable)

3.4.3.7 Message Flow Type

This property specifies the transmission flow type inside this link. There are only two types: FIFO and No Order.

3.4.3.8 Delay Type

Type of transmission delay of this link, there are 4 types, Synchronous, Random Uniformed, Random Poisson, and Random Customs

3.4.3.9 Delay Seed

A number required for generating a random number for the transmission delay on this link, or a delay time unit used in Synchronous environment

3.4.3.10 Reliable

This property is for setting reliability of this link; true means NO transmission is lost, otherwise it is lost with uniformly distributed random with probably that set in "*Probability of Failure*".

3.4.3.11 Probability of Failure

This property specifies the probability rate that a transmission is lost on this link, if a *Reliable* of this link is set to "False" (unreliable).

3.4.3.12 Total In Traffic

Total number of transmissions entering into this link until now

3.4.3.13 Total Out Traffic

Total number of transmissions departing from this link until now

3.5 Difference type of Topologies

DisJ Graph Editor provides a topology template library that offers a set of readymade graphs as followed:

3.5.1 Ring Graph

3.5.2 Tree Graph

3.5.3 Complete Graph

3.5.4 Spatial Graph

3.5.5 Mesh Graph

3.5.6 Mesh Graph

3.5.7 Torus Graph

3.5.8 Random Connected Graph

3.5.9 Random Forest Graph

3.5.10 Summary

The user can create a graph by using a basic drawing objects or using a topology template library that provides a common readymade graph. Once a graph is created, it can be modified and saved. There are properties required to be configured that the user must keep in mind and do it **before** start the simulation.

A graph that created by the topology template library can mix with different types of topologies into one graph and save it as a single file.

Chapter 4 Creating a Protocol

4.1 Creating a Protocol File (Java Class)

Creating a protocol is the same as creating Java class and file, which better to follow Java programming best practices by first creating a package by “*Right Click*” at “*src*” icon under the project folder, then select *New* → *Package* as shown below

The dialog pops up, then type in a package name in lowercase i.e. my_test as shown below, then click “Finish button”

The package name will be appeared under “src” icon folder. Now, create a new Java class by “Right Click” at “my_test” package, then select *New* → *Class* as show below

A file creation dialog pops up, then type in a Java Class name i.e. *AsFar*. Since *AsFar* is a protocol for Message Passing Model, therefore the supper class of this class is “**distributed.plugin.runtime.engine.Entity**”, click at “*Browse*” button to find and select the supper class (see Section 4.2) then “*check*” the checkbox “*Constructor from superclass*” and click “*Finish*” button

There might be Java error message show at constructor. Make sure the visibility is public and has no argument, and call `super(state)` inside as shown below.

```
// Entity states
public static final int STATE_SLEEP = 1;
public static final int STATE_WORKING = 2;
public static final int STATE_DONE = 3;

// Default constructor
public AsFar() {

    // define initial state of entity
    super(STATE_SLEEP);

}
```

The workbench will be looked like as follow

4.2 Rules and Conventions.

DisJ provides three types of distributed computing models and each has its own super class for the user to extend as follow

- Message Passing Model →
"distributed.plugin.runtime.engine.Entity"
- Agent with Whiteboard Model →
"distributed.plugin.runtime.engine.BoardAgent"
- Agent with Token Model →
"distributed.plugin.runtime.engine.TokenAgent"

Therefore, a user class has to extend one of these super classes based on the model, which it is an entry point for the DisJ simulator to execute and perform a simulation with the user protocol.

There are conventions for writing a distributed algorithm with DisJ that the user has to follow.

- 4.2.1 The entry class of the protocol **MUST extends** one of above super classes based on the model that the user is working on and creates an **EMPTY parameter Constructor**(default Constructor). This class does **not** need “public static void main(String[])” (even if the user has one it will never be used)
- 4.2.2 State for Entity in distributed protocol must be “**public static final int**” or “**public final int**” and the name of the state must begin with “*state*” or “*_state*” which the cases are insensitive
- 4.2.3 Assign an initial state to an entity by passing a state value into “**super (int initState)**” inside the default constructor created in 4.2.1
- 4.2.4 The user should **NOT** invoke any DisJ API inside the *constructor*, since the entity object has not yet been created
- 4.2.5 There are three methods that the user class has inherited from **Entity** of *Message Passing model* and the user might need to implement

public void init(): this method is invoked on a node if and only if the node property “*Initiator*” is set to be True (*see Chapter 3.4.2.3*)

public void alarmRing(): this method is invoked when an internal alarm clock of a node is rang. The alarm time can be set at a node by invoking method `setAlarm(int countDown)` (*see Chapter 4.2.10*)

public void receive(String inPortLabel, IMessage msg): this method is invoked when a node receives a message. This method is a main message processing functions of node algorithm

- 4.2.6 There are four methods that user class has inherited from **TokenAgent** of *Agent Token model*, and **BoardAgent** of *Agent Whiteboard model* and user might need to implement

public void init(): this method is invoked on every agent

public void alarmRing(): this method is invoked when an internal alarm clock of an agent is rang. The alarm time can be set at the agent by invoking method `setAlarm(int countDown)`

public void arrived(String viaPortLabel):this method is invoked when an agent arrived at a node.

public void notified(NotifyType type):this method is invoked when there is registered action happening at a node where an agent currently resides and registered to a given notification type. This method is a function that activates agent that waiting for something to happen at the. In order to register event at a node, the agent must call method **registerHostEvent(NotifyType)** at current resides node with preferred type of event, the register will be removed automatically when the agent leave the node. There are four **NotifyType** as follow

- AGENT_ARRIVAL: Activate when any agent arrived at this node
- AGENT_DEPARTURE: Activate when any agent leaved this node
- TOKEN_UPDATE: Activate when number of tokens at this node has been changed
- BOARD_UPDATE: Activate when whiteboard of this node has been modified

4.2.7 DisJ library has provided standard programming interfaces (API) for the user to access communication and infrastructure libraries for the reactive distributed algorithms. The following are examples of common and useful API that can be used in the algorithm. A complete list can be seen at Java Doc of DisJ.

Message Passing Model

```
sendTo ()  
become ()  
getState ()  
getName ()  
getInPorts ()  
getOutPorts ()
```

Agent with Token Model

```
moveTo ()  
become ()  
getAgentId ()  
getNodeId ()  
getState ()  
getNodeState ()  
getInPorts ()  
getOutPorts ()  
countHostToken ()  
countMyToken ()  
dropToken ()  
pickupToken ()
```

Agent with Board Model

```
moveTo ()  
become ()  
getAgentId ()  
getNodeId ()  
getState ()  
getNodeState ()  
getInPorts ()  
getOutPorts ()  
readFromBoard ()  
removeFromBoard ()  
appendToBoard ()
```

4.3 Coding a Protocol

The convention for distributed algorithm in DisJ is based on reactive model defined in *state x event* $\rightarrow$ *action* (see Chapter 8)

This section discusses how to convert “*state-event*” driven pseudo algorithm to “*even-state*” driven in Java algorithm. In order to make it easy to understand, The example uses a Ring Election protocol named “*As Far*” with multiple initiators, unique ID, bidirectional links, synchronous and total reliable environment.

The following snapshot shows the declaration of class, state, and default constructor of “*As Far*” protocol in Java with DisJ. Also, shows an implementation of `init()` method for the protocol.


```

File Edit Source Refactor Navigate Search Project Run Window Help
package my_test;

import distributed.plugin.runtime.IMessage;
import distributed.plugin.runtime.engine.Entity;

/**
 * Leader election protocol with multiple initiators, unique ID,
 * synchronous and total reliable communication.
 *
 * The initiators (candidates) will send messages with its ID to all
 * directions, and a message with smallest ID will kill every candidates
 * and return back to the sender. A candidate with smaller ID will kill
 * all arrival messages with bigger ID as well.
 */
public class AsFar extends Entity {

    // all possible states
    public static final int STATE_SLEEP = 0;
    public static final int STATE_ELECTION = 1;
    public static final int STATE_PASSIVE = 2;
    public static final int STATE_FOLLOWER = 3;
    public static final int STATE_LEADER = 4;

    private static final String MSG_LABEL_ELECTION = "Election";
    private static final String MSG_LABEL_NOTIFY = "Notify";

    // tracking that both of my messages have returned
    private boolean leftMin;
    private boolean rightMin;

    public AsFar() {
        super(STATE_SLEEP);
    }

    public void init() {
        String myId = this.getName();
        this.sendToAll(MSG_LABEL_ELECTION, myId);
        this.become(STATE_ELECTION);
    }
}

```

Below is an example of implementing **Entity.receive()** method

```

public void receive(String incomingPort, IMessage message) {

    String msg = (String) message.getContent();
    String msgLabel = message.getLabel();

    if (this.getState() == STATE_SLEEP) {
        this.sendToOthers(MSG_LABEL_ELECTION, msg);
        this.become(STATE_PASSIVE);
    }
    else if (this.getState() == STATE_PASSIVE) {
        if (msgLabel.equals(MSG_LABEL_NOTIFY)) {
            this.sendToOthers(MSG_LABEL_NOTIFY, msg);
            this.become(STATE_FOLLOWER);
        }
        else {
            this.sendToOthers(MSG_LABEL_ELECTION, msg);
        }
    }
    else if (this.getState() == STATE_ELECTION) {
        if (msgLabel.equals(MSG_LABEL_ELECTION)) {
            this.electing(incomingPort, msg);
        }
        else {
            // should not happen!!
        }
    }
    else if (this.getState() == STATE_LEADER) {
        if (msgLabel.equals(MSG_LABEL_NOTIFY)) {
            System.out.println("Election completed: " + this.getName() + " is a leader");
        }
        else {
            // should not happen!!
        }
    }
}

/**
 * Helping function that will compare a received ID with my ID
 */
private void electing(String incomingPort, String id) {

    String myId = this.getName();

    if (id.compareTo(myId) < 0) {
        this.sendToOthers(MSG_LABEL_ELECTION, id);
        this.become(STATE_PASSIVE);
    }
    else if (id.compareTo(myId) == 0) {
        // my election msg has returned
        if (this.leftMin == true)

```

An inherited function that the protocol does not need, the user can leave it empty as function **Entity.alarmRing()** as shown below

```

public void init() {
    String myId = this.getName();
    this.sendToAll(MSG_LABEL_ELECTION, myId);
    this.become(STATE_ELECTION);
}

@Override
public void alarmRing() {
    // this protocol does not need this function
}

public void receive(String incomingPort, IMessage message) {

    String msg = (String) message.getContent();
    String msgLabel = message.getLabel();

```

4.4 Setting State Color

In order to see real-time state update in the DisJ editor during the simulation; the user must specify a color for each state that defined in the protocol (see Chapter 4.2.2) by focus on the workbench of the DisJ editor, then go to menu bar, select Simulator→Add States as shown below

When a dialog box shows up, give a value (number) that match to the state value in the protocol and select the color from a “Color Palette”, the value will be recorded in number-color pair; if a state are given twice, it will override the existing record. Once the graph is saved, the state-color pairs will be saved along with the graph file.

If the user wants to remove or see the list of state-color pairs, go to menu *Simulator* → *Remove State*; a dialog box shows a list of state-color pairs. To remove, check the checked box at a pair that wants to remove and click Remove as shown below.

Note that the simulator will ignore the state-color pairs that do not match the states declared in protocol. Therefore, only match state-color pairs are used during the simulation.

Chapter 5. Running a Simulation

There are few things required before running the simulator and it is described as follow.

5.1 Loading protocol into a Topology

Once topology (a graph file) and a protocol are created, to simulate the protocol in the simulator, the user has to load the protocol into the topology by focus at DisJ editor workbench, then go to the menu and select *Simulator* → *Load Java Class* as shown below

Once a dialog shows up; there are three models to select, in this example is *Message Passing Model*. Then type in a *Fully Qualified Class Name* of the entry class (a class that implement **distributed.runtime.engine.Entity**), in this example is "**my_test.AsFar**" (as show below) then click OK. Remember that all the protocol classes and the graph files (.gph) must be located under a same project folder, in this example is "*my_project*". Each graph file is independent from other graph files of a same project, means loading the entry class into one graph file does not affect other graph file.

Important: If a protocol has been loaded into a graph file, then later on a **.java** file of the protocol has been **modified**, the graph file has to be closed and reopen again in order to reload a new modified Java source into the graph file.

5.2 Run/Resume the simulator

Once the class is loaded into the topology; then go to menu select *Simulator* → *Run/Resume*, now DisJ is simulating the protocol. In order to simulate another protocol with a same graph, the user must reload a new protocol into the graph.

5.3 Stop the simulator

In order to stop the simulation, go to menu and select *Simulator* → *Stop* at anytime once the simulation is started. After the simulation stopped (issue stop command) all current states and information are wiped out. Therefore, the user cannot see any information display during the simulation in the “*Properties View*” anymore. Therefore, before rerun or run a new a simulation, user must make sure that the simulator is stopped in order to reset the states of the graph.

Note: if the user save the graph while in state during the simulation, the state of the simulation also is saved into a graph file, and the original state of graph will not be able to retrieve by the DisJ simulator.

5.4 Suspend the simulator

In order to pause the simulation, go to menu and select *Simulator* → *Suspend* at anytime once the simulation is started, and to resume the execution by go to menu and select *Simulator* → *Run/Resume* as well. Remember that suspend does not wipe out the current states and information therefore, the user can observe the current states in graph and “*Properties View*” and the DisJ simulator can continue simulate from current state.

5.5 Step Next

This feature is not yet support.

5.6 Setting Processing Speed

This feature allows the user to setup the speed of the processor executes the protocol, by first the simulation **MUST** be **suspended** then go to menu and select *Simulator* → *Process Speed* and the process speed scale will show up and allows the user to slide in order to adjust the current speed of this process as shown below

5.7 Replay Record

This feature allow the user to *replay* the simulation that has been run based on a saved record file (.rec) Currently, every simulation will be **saved automatically** once the simulation is ended under the *project/bin/* in this example is "*../my_project/bin*", the .rec file name will be a name of a topology (graph file .gph) e.g. *ring.rec* in this example. Therefore, the .rec file will be override automatically if the simulation has been run on a same graph file more than once, so, it is the user responsibility to rename a .rec file if the user willing to save the record file for further usage.

In order to replay, the user select an icon , a Open File dialog pops up, then select a .rec file e.g. ring.rec. The replay starts immediately after the dialog is closed. The user may use suspend, resume, adjust speed and stop as if the simulation is running. Note that the results, statistic report and behavior of the replay is the exactly the same as what happening during the simulation of the record and cannot be edited by the DisJ simulator.

Here is a snap shot when DisJ simulator is executing

The screenshot shows the DisJ simulator interface. The main window displays a ring graph topology with 10 nodes and 10 links. The nodes are labeled n0 through n9. The interface includes a Package Explorer on the left, a Palette on the right, and a Properties window at the bottom.

Package Explorer:

- disj
 - my_project
 - src
 - my_test
 - AsFar.java
 - CountAgent.java
 - CountNetwork.java
 - IRE System Library
 - Referenced Libraries
 - ring.gph

Properties Window:

Property	Value
G00 Graph Name	ring.gph
G01 Total Nodes	10
G02 Total Links	10
G03 Total Messages Received	12
G04 Total Messages Sent	24
G05 Global Message Flow Type	FIFO
G06 Global Delay Type	Synchronous
G07 Global Delay Seed	1
G08 Protocol	my_test.AsFar
G09 Max Number of Tokens per Agent	1
G10 Number of Agents at Start	0
G11 Current Number of Agents	0

To view a node/link properties by selecting a node/link in editor at anytime during simulation execution

The screenshot shows the DisJ simulator interface with a selected node (n7) highlighted. The Properties window at the bottom displays the properties for the selected node.

Properties Window:

Property	Value
N00 Node Name	n7
N01 User Input	
N02 Initiator	True
N03 Alive	True
N04 Breakpoint	Disable
N05 Number of Messages Received	8
N06 Number of Messages Sent	12
N07 Outgoing Ports	[left, right]
N08 Incoming Ports	[left, right]
N09 States Transition	[STATE_ELECTION, STATE_PASSIVE, STATE_FOLLOWER]
N10 Number of Agents Hosted	0
N11 Current Number of Agents	0
N12 Current Number of Tokens	0

Mesh
Hyper Cube
Torus_1
Connected

Problems Javadoc Declaration Properties Console Search

Property	Value
L00 Edge ID	e7
L01 Type of Direction	Bi-Directional
L02 Source Name	n7
L03 Source Port Name	right
L04 Target Name	n8
L05 Target Port Name	left
L06 Message Flow Type	FIFO
L07 Delay Type	Fixed
L08 Delay Seed	1
L09 Reliable	True
L10 Probability of Failure	0
L11 Total In Traffic	10
L12 Total Out Traffic	5

Chapter 6. Graphical and Statistics Reports

DisJ simulator provides basic information and statistic reports. Currently there are two types of reports, Java Console print out and DisJ View reports, which are not 100% complete (based on build milestone). However, it still provides useful information for the user to use and analyze the simulating protocol. The following are snapshots of each type of reports

The Console output reports

```
DisJ Console
****Simulation for ring.gph is successfully over.****
***** STATISTIC REPORT *****
Total Message has been sent: 92
Total Message has been received: 92
Total Message has entered link: 92
Total Message has leaved link: 46
Total Average delay time has been accumulated: 4

State STATE_FOLLOWER has 9
State STATE_LEADER has 1

Message Election has been created 36
Message Notify has been created 10
States VS #Msg Sent
STATE_ELECTION = 12
STATE_PASSIVE = 60
STATE_FOLLOWER = 18
STATE_LEADER = 2
```

DisJ View reports

Opening “DisJ View” in Eclipse is the same way as opening “Properties View” by go to menu and select *Window* → *Show View* → *Other...* as shown below

Once the Show View dialog pop up, select *DisJ* → *DisJ View* as shown below

Chapter 7. Conclusion

DisJ is a lightweight tool for writing, test and simulate the reactive distributed algorithm in Java™. DisJ provides virtual processors in a given network topology (graph) with graphical interaction that allows the user to view the protocol during the simulation. Also, DisJ abstracts the user from setting up sophisticate infrastructure while preserve (almost) reality of distributed environment. Moreover, DisJ provides a nice Java™ Development Tool (JDT) from Eclipse™, which helps the user to write and debug a complex protocol with Java™ language. The replay function allows the user to re-visual the past simulation without modifying the results. Finally, the statistic report provides different levels of information that DiJ collects during the simulation and the user can use it for analysis offline.

DisJ simulator still ongoing project and there are a lot of rooms for improvement. Therefore, any feedbacks and contributions are welcome.

Chapter 8. Sample Codes

The protocol is expressed as a set of rules of the form “*state* x *event* $\rightarrow$ *action*”, according to the reactive model defined in DADA.

In the convention used in DADA, rules are grouped by *state*, listing the difference “*event* $\rightarrow$ *action*” for that state.

Example 1:

A Protocol “*As Far*”, which elect a leader among the initiators in a bidirectional link Ring.

States: S = {ASLEEP, ELECTING, PASSIVE, FOLLOWER, LEADER}
S(init) = {ASLEEP}
S(terminate) = {FOLLOWER, LEADER}

Restrictions: RI U Bidirectional Ring

ASLEEP:

Spontaneously

begin

send (“Election”, id(x)) to all;
become ELECTING;

end

Receiving(“Election”, id)

begin

send (“Election”, id) to others;
become PASSIVE;

end

ELECTING

Receiving(“Election”, id)

begin

if id < min then

send (“Election”, id) to others;
become PASSIVE;

else if id = min from both directions then

send (“Notify”) to all;
become LEADER;

endif

end

PASSIVE

```

    Receiving( "Election", id)
    begin
        send ("Election", id) to others;
    end

    Receiving( "Notify")
    begin
        send("Notify") to others;
        become FOLLOWER;
    end

```

LEADER

```

    Receiving( "Notify")
    begin
        done election
    end

```

In DisJ, the dual convention is used grouping the rules by *event*, listing the difference “*state*→*action*” for that event.

Example 2:

The same protocol from *Example 1*, expressed using the dual convention.

States: S = {ASLEEP, ELECTING, PASSIVE, FOLLOWER, LEADER}
 S(init) = {ASLEEP}
 S(terminate) = {FOLLOWER, LEADER}

Restrictions: RI U Bidirectional Ring

SPONTANEOUSLY:

```

    Asleep
    begin
        send ("Election", id(x)) to all;
        become ELECTING;
    end

```

RECEIVING

```

    Asleep ( "Election", id)
    begin
        send ("Election", id) to others;
        become PASSIVE;
    end

```



```

    end

    Passive ("Election", id)
    begin
        send ("Election", id) to others;
    end

    Passive ("Notify", msg)
    begin
        send ("Notify", msg) to others;
        become FOLLOWER
    end

    Electing ("Election", id)
    begin
        if id < min
            send("Election", id) to others;
            become PASSIVE

        else if id = min from both directions then
            send("Notify") to all;
            become LEADER;

        endif
    end

    Leader("Notify")
    begin
        done election
    end
end

```

Example 3:

The same protocol from *Example2*, expressed using Java Code.

```

package my_test;

import distributed.plugin.runtime.IMessage;
import distributed.plugin.runtime.engine.Entity;

/**
 * Leader election protocol with multiple initiators, unique ID,
 * synchronous and total reliable communication.
 *
 * The initiators (candidates) will send messages with its ID to all
 * directions, and a message with smallest ID will kill every
 * candidates
 * and return back to the sender. A candidate with smaller ID will kill
 * all arrival messages with bigger ID as well.
 */

```

```

*/
public class AsFar extends Entity {

    // all possible states
    public static final int STATE_SLEEP = 0;
    public static final int STATE_ELECTION = 1;
    public static final int STATE_PASSIVE = 2;
    public static final int STATE_FOLLOWER = 3;
    public static final int STATE_LEADER = 4;

    // all message label will be used in the protocol
    private static final String MSG_LABEL_ELECTION = "Election";
    private static final String MSG_LABEL_NOTIFY = "Notify";

    // tracking that both of my messages have returned
    private boolean leftMin;
    private boolean rightMin;

    /**
     * Default constructor
     */
    public AsFar() {
        super(STATE_SLEEP);
    }

    public void init() {
        String myId = this.getName();
        this.become(STATE_ELECTION);
        this.sendToAll(MSG_LABEL_ELECTION, myId);
    }

    @Override
    public void alarmRing() {
        // this protocol does not need this function
    }

    public void receive(String incomingPort, IMessage message) {

        String msg = (String) message.getContent();
        String msgLabel = message.getLabel();

        if (this.getState() == STATE_SLEEP) {
            this.become(STATE_PASSIVE);
            this.sendToOthers(MSG_LABEL_ELECTION, msg);
        } else if (this.getState() == STATE_PASSIVE) {
            if (msgLabel.equals(MSG_LABEL_NOTIFY)) {
                this.become(STATE_FOLLOWER);
                this.sendToOthers(MSG_LABEL_NOTIFY, msg);
            } else {
                this.sendToOthers(MSG_LABEL_ELECTION, msg);
            }
        } else if (this.getState() == STATE_ELECTION) {
            if (msgLabel.equals(MSG_LABEL_ELECTION)) {
                this.electing(incomingPort, msg);
            }
        }
    }
}

```

```

        }else{
            // should not happen!!
        }

    } else if (this.getState() == STATE_LEADER) {
        if (msg.equals(MSG_LABEL_NOTIFY)){
            System.out.println("Election completed: " +
                this.getName()
                + " is a leader");
        }else{
            // should not happen!!
        }
    }
}

/*
 * Helping function that will compare a received ID with my ID
 */
private void electing(String incomingPort, String id) {

    String myId = this.getName();

    if (id.compareTo(myId) < 0) {
        this.become(STATE_PASSIVE);
        this.sendToOthers(MSG_LABEL_ELECTION, id);

    } else if (id.compareTo(myId) == 0) {
        // my election msg has returned
        if (this.leftMin == true)
            this.rightMin = true;
        else
            this.leftMin = true;

    }

    if (this.rightMin && this.leftMin) {
        this.become(STATE_LEADER);

        // send notify to one direction only
        this.sendTo(MSG_LABEL_NOTIFY, incomingPort, "I " +
            myId + " is your leader");

    }
}
}

```

Example 4:

The Black Hole Search protocol with Co-Locate agents using Java Code.

```

package test;

import java.util.List;

import distributed.plugin.runtime.engine.BoardAgent;

```

```

/**
 * Whiteboard:
 * A Black Hole search in bidirectional un-oriented Ring with
 * 2 co-locate agents, known N (number of nodes), known K
 * (number of agents), total reliability and FIFO
 */
public class BHC extends BoardAgent {

    public static final int STATE_NODE_UNKNOWN = 0;
    public static final int STATE_NODE_CLEAN = 1;
    public static final int STATE_NODE_TO_BH = 2;

    public static final int STATE_AGENT_WORKING = 3;
    public static final int STATE_AGENT_FOUND_BH = 4;
    public static final int STATE_AGENT_DONE = 5;

    private boolean explored;
    private boolean confirm;
    private boolean traverse;
    private boolean forward;
    private int round;
    private int numReq;
    private int numDone;

    public BHC() {
        super(STATE_NODE_UNKNOWN);
        this.round = 0;
        this.numReq = 0;
        this.numDone = 0;
        this.confirm = false;
        this.explored = false;
        this.traverse = false;
        this.forward = false;
    }

    public void init() {
        this.become(STATE_AGENT_WORKING);
        this.setNodeState(STATE_NODE_CLEAN);

        this.round = 1;
        int o = (this.getNetworkSize() - 1)/2;
        int e = this.getNetworkSize() - o - 1;
        List<String> ports = this.getOutPorts();
        List<String> info = this.readFromBoard();
        String p = null;
    }
}

```

```

        System.out.println("Board: " + info);
        if(info.isEmpty()){
            // first agent
            this.numReq = 0;
            p = ports.get(0);
            this.appendToBoard("active:" + p);
        }else{
            // second agent
            this.numReq = e;
            String s = info.get(0);
            String[] v = s.split(":");
            for(int i =0;i < ports.size(); i++){
                p = ports.get(i);
                if(!p.equals(v[1])){
                    break;
                }
            }
            this.appendToBoard("active:" + p);
        }
        // set for next exploring
        this.moveTo(p);
    }

    /*
     * My first visit to this node, and it is the first
     * visit from this port
     */
    private void myFirstVisit(String port){
        // First arrived safely from safe node
        this.appendToBoard("safe:" + port);

        // Then go back to say that this node also safe
        this.explored = true;
        this.confirm = true;
        this.moveTo(port);
    }

    /*
     * Confirming port check to a previous node that this port
     * is safe then get back to the port to finish of the work
     */
    private void confirmVisit(List<String> info, String port){
        String s = null;
        String[] v = null;
        for(int i =0; i < info.size(); i++){
            s = info.get(i);
            v = s.split(":");

```

```

        if(v.length == 2){
            // get my port status
            if(v[1].equals(port)){
                if(v[0].equals("active")){
                    this.removeRecord(s);
                    this.appendToBoard("safe:" + port);
                }
            }
        }else{
            // someone left a note
            // reset my work plan
            int share = Integer.parseInt(v[2]);
            this.numReq = this.numReq - share;
            this.round = Integer.parseInt(v[1]);
            this.removeRecord(s);
        }
    }
    // back to the port
    this.confirm = false;
    this.numReq--;
    this.moveTo(port);
}

```

```

public void arrive(String port) {

    // traversing back
    if(this.traverse == true){
        this.traversBack(port);
        return;
    }

    // move to my current end territory
    if(this.forward == true){
        this.toMyEnd(port);
        return;
    }

    // keep exploring
    List<String> info = this.readFromBoard();
    if(info.isEmpty()){
        // I am the first to this node
        this.myFirstVisit(port);
        this.setNodeState(STATE_NODE_CLEAN);
    }else {
        String s = null;
        String[] v = null;
    }
}

```

```

        if(this.confirm == true){
            this.confirmVisit(info, port);
            return;
        }
        System.out.println("Agent: " + this.getAgentId() + " numReq " +
numReq);
        // my second visit after confirm safe port to previous node
        if(this.explored == true){
            // check my round
            if(numReq == 0){
                // complete round, go backward to find partner
                //current location
                System.out.println("Start traversing back from
node: " + this.getNodeId());
                System.out.println("whiteboard:          "          +
this.readFromBoard());

                this.traverse = true;
                this.numDone = 0;
                String p = this.getAnotherPort(port);
                this.traversBack(p);
                return;
            }

            if(info.size() == 1){
                // still the first at this node
                s = info.get(0);
                v = s.split(":");
                if(v[1].equals(port)){
                    if(v[0].equals("active")){
                        // should not happen, i just marked
                        safe on my first arrival
                        System.err.println("should          not
                        happen 1");
                    }

                    // find another port to explore
                    List<String> ports = this.getOutPorts();
                    String p = null;
                    for(int i =0; i < ports.size(); i++){
                        p = ports.get(i);
                        if(!p.equals(port)){
                            this.appendToBoard("active:"
+ p);

                            // move to new port
                            this.explored = false;
                            this.moveTo(p);

```

```

    }
}

}else{
    // should not happen
    // i just visited on my first arrival
    System.err.println("should not happen 2");
}
}else{
    // other has visited this node from another port

    for(int i =0; i < info.size(); i++){
        s = info.get(i);
        v = s.split(":");
        if(v[0].equals("active")){
            if(v[1].equals(port)){
                // should not happen
                // a port that i came from is

                System.err.println("should
safe
not happen 3");

                }else{
                    // someone is exploring it

                    // so wait.

                    this.registerHostEvent(NotifyType.BOARD_UPDATE);
                }
            }else {
                if(v[1].equals(port)){
                    // my port, ignore it

                }else{
                    // another port that was
                    // safe by other (No black
                    // repeat my node territory
                    // should not happen

                    System.err.println("should
not happen 4");

                    break;
                }
            }
        }
    }
}
}

```



```

        } else {
            // my first visit but other has been here before
            this.myFirstVisit(port);
        }
    }
}

private void traversBack(String port){
    String p = this.getAnotherPort(port);
    this.numDone++;
    System.out.println("Travers back to port " + p);
    if(isSafe(p)){
        this.moveTo(p);

    }else{
        // we found under process node
        // done traversing
        this.traverse = false;

        // check whether is it a last safe node
        if(this.numDone == (this.getNetworkSize() - 1)){
            // we found black hole
            this.become(STATE_AGENT_FOUND_BH);

            // the active port is a port to BH
            this.setNodeState(STATE_NODE_TO_BH);
            System.out.println("Agent: " + this.getAgentId() + " found
BH from Node " + this.getNodeId());

            // move to my end node
            this.numDone = 0;
            this.forward = true;
            this.toMyEnd(p);

        }else{
            // compute new share

            int remain = this.getNetworkSize() - this.numDone;
            System.out.println("Found the struggling node " +
this.getNodeId() + " with remain " + remain);

            int share = remain/2;
            this.round++;
            this.numReq = share;

            // post to board that i will do extra share from my end
            this.appendToBoard("round:" + round + ":" + share);

```

```

        // move to my end node
        this.numDone = 0;
        this.forward = true;
        this.toMyEnd(p);
    }
}

private void toMyEnd(String port){
    String p = this.getAnotherPort(port);
    if(isSafe(p)){
        this.moveTo(p);

    }else{
        // arrive at my end
        this.forward = false;
        System.out.println("Arrived my end Node: " + this.getNodeId() + "
numReq " + this.numReq);
        if(this.getState() != STATE_AGENT_FOUND_BH){
            // arrive at my end of previous round
            List<String> info = this.readFromBoard();
            String s = null;
            String[] v = null;
            System.out.println("Board " + info);
            if(info.size() > 1){
                for(int i = 0; i < info.size(); i++){
                    s = info.get(i);
                    v = s.split(":");
                    if(!v[0].equals("safe")){
                        this.explored = false;
                        this.confirm = false;
                        // continue exploring
                        System.out.println("Heading to " +
v[0] + " port " + v[1]);
                        this.moveTo(v[1]);
                        break;
                    }
                }
            }else{
                this.explored = false;
                this.appendToBoard("active:" + p);
                this.moveTo(p);
            }
        }else{
            this.become(STATE_AGENT_DONE);
            this.setNodeState(STATE_NODE_TO_BH);
        }
    }
}

```

```

        }
    }
}

private boolean isSafe(String port){
    List<String> info = this.readFromBoard();
    boolean safe = false;
    String s = null;
    String[] v = null;
    for(int i =0; i < info.size(); i++){
        s = info.get(i);
        v = s.split(":");
        if(v.length == 2){
            if(v[0].equals("safe") && v[1].equals(port)){
                safe = true;
                break;
            }
        }
    }
    return safe;
}

/*
 * Get opposite port from a given port in a ring
 */
private String getAnotherPort(String port){
    List<String> ports = this.getOutPorts();
    String p = null;
    for(int i =0; i < ports.size(); i++){
        p = ports.get(i);
        if(!p.equals(port)){
            return p;
        }
    }
    return p;
}

public void notified(NotifyType arg0) {
}

public void alarmRing() {
}
}

\end{verbatim}

```

© Copyright

Nothapol Piyasin, 2012