RFIDSim - A Discrete Event Simulator for Radio Frequency Identification Systems

by

Kenneth Kwan-Wai Yu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author .....
Department of Electrical Engineering and Computer Science
August 22, 2003

Certified by .....
Sanjay E. Sarma
Associate Professor of Mechanical Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

RFIDSim - A Discrete Event Simulator for Radio Frequency Identification Systems

by

Kenneth Kwan-Wai Yu

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents *RFIDSim*, a discrete event process-oriented simulator designed to model Radio Frequency Identification (RFID) communication. The simulator focuses on the discovery and identification process of passive powerless RFID tags - a category of low cost tags that harvest all of their energy from the radio frequency signals sent by the reader. *RFIDSim* currently supports the Auto-ID Class 0 and Class 1 UHF RFID protocols, and is designed to be an extensible framework for additional protocol implementations, as well as a modular structure that partitions the reader / tag logic separately from the signal modeling.

Thesis Supervisor: Sanjay E. Sarma
Title: Associate Professor of Mechanical Engineering

Acknowledgments

I must begin the ackwowledgements section by thanking Daniel Engels, my unofficial thesis supervisor. Without Daniel, this thesis and simulator simply would not have come to fruition. Despite his busy schedule, he would always try to make time to help me out with any problems I have. Daniel should also be credited for his patience in bringing me up to speed on RFID technologies, as well as the key aspects of signaling, some of which I'm still trying to absorb. He is also always willing to provide constructive comments on keeping the simulator design flexible and extensible. At many times during the project, the frustration from not understanding the concepts I need to know has been almost unbearable, but Daniel has always been there to instill enough confidence in me to keep the project moving.

I would also like to thank Sanjay Sarma, my official thesis supervisor, for entrusting me with the responsibily to write this simulator. Signaling and simulations were almost entirely new concepts to me when the project first started, and I'm thankful to Sanjay that I now know enough to sound like an RFID expert (except when I'm in the presence of Daniel or Sanjay).

I would like to thank Tom Scharfeld. Although our interactions have been slim, his thesis on RFID was extremely valuable as a reference, as it contained everything I needed to know for writing this simulator.

I would like to thank my family for providing me with their support all these years.

Finally, I would like to thank my apartment roommates for making my last year at MIT enjoyable and fun. I would also like to thank Wesley and Chun for keeping the friendship strong since the two years they have graduated from MIT. Last but not least, I would like to thank Vivian for turning the two toughest years of my MIT career into the best ones.

Contents

1 Introduction **17**

1.1 Radio Frequency Identification 17

1.2 Acceptance Roadblocks 19

1.3 Auto-ID Center 20

1.4 Problems 20

1.4.1 Algorithm evaluation methods 20

1.4.2 Current simulation solutions do not fit the needs 22

1.5 Motivations behind creating *RFIDSim* 22

1.6 Thesis Overview 23

2 Terminology **25**

2.1 Discovery Process 25

2.2 Readers 25

2.3 Tags 25

2.4 Asymmetric communication 26

2.4.1 Restrictions imposed on wireless communications 26

2.4.2 Tags are powerless / stateless 26

2.4.3 Tags are passive 27

2.4.4 Reader signals are broadcasted to all tags 27

2.4.5 Tags communicate with the reader through a channel relative to the reader 27

2.4.6 Reader has complete control of the communication 27

2.5 Signaling methods 29

2.5.1 Reader Modulation 29
2.5.2 Tag Demodulation 30
2.5.3 Tag Modulation 30
2.5.4 Reader Demodulation 30
2.6 Constraints 30
2.7 Discovery Algorithm using Tree-Walking 32
2.7.1 Using identification codes for tree construction 32
2.7.2 Tree-walking algorithm 32

3 Design Goals **35**
3.1 Level of abstraction between logic and signal modeling 35
3.1.1 Developers should not have to understand everything 35
3.1.2 Modifying protocol logic without knowledge of signaling assumptions 36
3.1.3 Changes in signal modeling assumptions should not break the protocol logic 36
3.1.4 Complete models or objects can be replaced 36
3.1.5 Code clarity 37
3.2 Support for future protocols 37
3.2.1 Consistent simulation results 37
3.2.2 Shorter implementation periods for new protocols. 37
3.3 Comprehensive analysis tools for simulation results 38
3.3.1 Controlled environments for testing 38
3.3.2 Extensive logging and debugging features 38

4 Design **39**
4.1 Process-oriented Discrete Simulator 39
4.2 Design Overview 40
4.2.1 By function 40
4.2.2 By level of generality 41
4.2.3 By level of participation 41

4.3 Significant actors in the simulation 42
4.3.1 Reader 42
4.3.2 Reader Transmitter 43
4.3.3 Tag 43
4.3.4 Tag Transmitter 44
4.3.5 Tag Receiver 44
4.3.6 Reader Receiver 44
4.4 Significant non-actors and supporting functions 45
4.4.1 Signal 45
4.4.2 ITM 46
4.4.3 Signal Library 46
4.4.4 Logic Library (Class 1 only) 46
4.5 Signal Handling for Tag processes 47
4.5.1 Looking ahead 47
4.5.2 Preemptive Signal Handling 48
4.6 Unreachable Tags 50
4.7 Asymmetric communication in *RFIDSim* 51

5 Implementation **53**
5.1 Language and Tools used 53
5.1.1 Portability 53
5.1.2 Object-Oriented Nature 53
5.1.3 Mature support for graphical user interfaces 53
5.1.4 The chosen simulator framework is implemented in Java . . . 54
5.2 Classes Overview 54
5.3 Common Java classes 54
5.3.1 Signal class 54
5.3.2 ITM class 58
5.3.3 RFIDObject class 58
5.3.4 Reader class 59

5.3.5 ReaderSend class 59
5.3.6 Tag class 59
5.3.7 TagSend class 59
5.3.8 TagReceive class 60
5.3.9 RFIDModel class 60
5.3.10 SimLog class 61
5.3.11 SigLib class 61
5.3.12 RiseAndFall class 62
5.4 Signal Modeling 62
5.4.1 Distance and Noise function 62
5.4.2 Delay function 62
5.4.3 Edge Detection Algorithm 63
5.4.4 Flat Signal Detection Algorithm 63
5.5 Class 0 64
5.5.1 ClassZeroITM class 64
5.5.2 ClassZeroReader class 64
5.5.3 ClassZeroTag class 64
5.5.4 ClassZeroTagReceive class 65
5.5.5 ClassZeroModel class 65
5.5.6 ClassZeroSigLib class 65
5.6 Class 1 65
5.6.1 ClassOneITM class 65
5.6.2 ClassOneReader class 66
5.6.3 ClassOneTag class 66
5.6.4 ClassOneTagReceive class 66
5.6.5 ClassOneModel class 67
5.6.6 ClassOneSigLib class 67
5.6.7 ClassOneLogLib class 67
5.7 Other objects used by *RFIDSim* 67
5.7.1 Experiment class 67

5.7.2 Environment class 68
5.7.3 StopCondition class 68
5.8 User Interface 68
5.8.1 EnvironmentPanel class 68
5.8.2 ClassZeroPanel 68
5.8.3 ClassOnePanel 69
5.8.4 RFIDWindow class 69
5.8.5 ResultsWindow class 69
5.8.6 MapDialog class 69
5.8.7 SimLogTableModel class 69
5.8.8 SignalPanel class 69
5.8.9 RAndFPanel class 70

6 Usage **71**
6.1 Building with source code 71
6.2 Execution requirements 72
6.3 Using *RFIDSim* 72

7 Conclusion **81**
7.1 Summary 81
7.2 Future work 82
7.2.1 Simulator performance 82
7.2.2 Refinement in the signal modeling assumptions 82
7.2.3 Support for powered tags 82
7.2.4 Coordination between readers 83

A Class 0 **85**
A.1 ITM 85
A.1.1 ID0 85
A.1.2 ID1 87
A.1.3 ID2 87

A.2 Reader Commands 87
A.2.1 Master reset 87
A.2.2 Calibration 88
A.2.3 Zero / One / Null 89
A.3 Tag States 90
A.3.1 Dormant State 90
A.3.2 Calibration State 90
A.3.3 Global Command Start State 90
A.3.4 Global Command State 90
A.3.5 Tree Start State 92
A.3.6 Tree Traversal State 92
A.3.7 Traversal Mute State 92
A.3.8 Singulated Command Start State 92
A.3.9 Singulated Command State 92
A.3.10 Singulated Command Mute State 92
A.4 Tag Backscatter Responses 93
A.4.1 Zero 93
A.4.2 One 93
A.5 Discovery Process 93
A.5.1 Algorithmic Overview 93
A.5.2 Signal Overview 95
A.5.3 Analysis 95

B Class 1 **97**

B.1 ITM 97
B.2 Reader Command Format 98
B.2.1 PREAMBL 98
B.2.2 CLKSYNC 98
B.2.3 SOF 98
B.2.4 CMD 99

B.2.5 P1 99
B.2.6 PTR 99
B.2.7 P2 99
B.2.8 LEN 99
B.2.9 P3 100
B.2.10 VALUE 100
B.2.11 P4 100
B.2.12 P5 100
B.2.13 EOF 100
B.3 Reader Commands 100
B.3.1 ScrollAllID 101
B.3.2 ScrollID 101
B.3.3 PingID 101
B.3.4 Quiet 101
B.3.5 Talk 101
B.3.6 Kill 102
B.3.7 ProgramID 102
B.3.8 VerifyID 102
B.3.9 LockID 102
B.3.10 EraseID 103
B.4 Reader Command Response Period 103
B.4.1 ScrollID Response Period 103
B.4.2 PingID Response Period 104
B.5 Tag States 104
B.5.1 Talk State 105
B.5.2 Quiet State 105
B.5.3 Killed State 105
B.6 Tag Backscatter Responses 105
B.6.1 ScrollID Reply 105
B.6.2 PingID Reply 106

B.6.3 VerifyID Reply 106
B.7 Discovery Process 107
B.7.1 Bin Modulation 107
B.7.2 Algorithmic Overview 108
B.7.3 Signal Overview 109
B.7.4 Analysis 109

List of Figures

1-1 Components of a complete RFID system. 18

2-1 The generalized communication between reader and tags. 28
2-2 Signal diagram of zero data symbol in the Auto-ID Class 0 protocol. . 29
2-3 Binary string "1101" represented as a binary tree path. 32

4-1 The interaction among the five main actors. 42
4-2 Illustrated walkthrough of Preemptive Signal Handling 49

5-1 *RFIDSim* classes categorized by function 55
5-2 *RFIDSim* classes categorized by level of generality 56
5-3 *RFIDSim* classes categorized by level of participation 57

6-1 The main window of *RFIDSim* (screenshot) 73
6-2 Choosing a different profile in *RFIDSim* (screenshot) 74
6-3 Dimensions dialog (screenshot) 74
6-4 Map dialog with new environment (screenshot) 75
6-5 Dialog after clicking on Environment (screenshot) 76
6-6 Results from simulation (screenshot) 77
6-7 Filtered results in a new window (screenshot) 77
6-8 Graphical representation of a Signal instance (screenshot) 78
6-9 Graphical representation of a RiseAndFall instance (screenshot) . . . 79

A-1 Identification code of the ID0 format for the Class 0 protocol. 86
A-2 Identification code of the ID1 format for the Class 0 protocol. 86

A-3 Identification code of the ID2 format for the Class 0 protocol. 87
A-4 Signal of a master reset command for the Class 0 protocol. 88
A-5 Signal of a master reset command for the Class 0 protocol. 88
A-6 Signal of a zero data symbol for the Class 0 protocol. 89
A-7 Signal of a one data symbol for the Class 0 protocol. 89
A-8 Signal of a null data symbol for the Class 0 protocol. 89
A-9 State Machine defined for Class 0 tags 91
A-10 Signal form of the commands sent by the reader at the beginning of the discovery process for the Class 0 protocol. 95

B-1 Identification code structure of a Class 1 ITM. 97
B-2 Modulated signal form of the binary string "11001010". 98
B-3 State Machine defined for Class 0 tags 104
B-4 Tree representation of walkable paths using bin modulation 107

Chapter 1

Introduction

This thesis presents *RFIDSim*, a discrete event process-oriented simulator designed to model communication between RFID readers and tags in a simulated environment. *RFIDSim* currently supports the Auto-ID Class 0 and Class 1 UHF protocols, and is designed to be an extensible framework for additional protocol implementations, as well as a modular structure that partitions the reader / tag logic separately from the signal modeling. The simulator focuses on the discovery and identification process of passive powerless RFID tags - a category of low cost tags that harvest all of their energy from the radio frequency signals sent by the reader.

1.1 Radio Frequency Identification

Radio Frequency Identification (RFID) is a special class of wireless communication that allows identification of objects through information embedded in an attached microchip without requiring line-of-sight. This allows computer systems to identify and recognize physical objects through a tag attachment. An RFID system consists of **tags**, microchips that contain unique identifying information about a physical object, and **readers**, transmitters that initiates the discovery and identification process of tags in a surrounding environment [1]. A more thorough introduction to RFID systems exist in Scharfeld's Masters Thesis on RFID [2] and Finkenzeller's *RFID Handbook: Radio-Frequency Identification Fundamentals and Applications* [3].

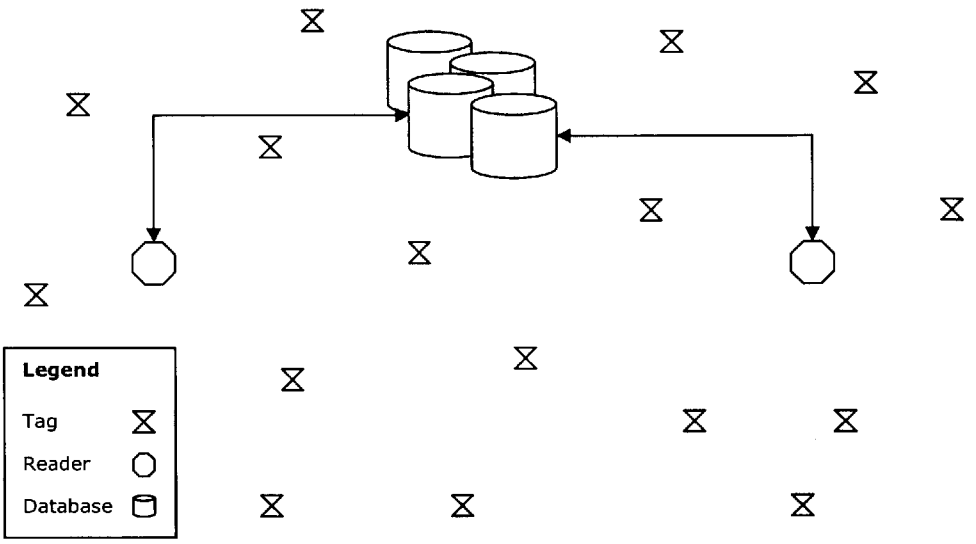

Figure 1-1: Components of a complete RFID system.

Figure 1-1 shows the essential components of a complete RFID system. The readers are controlled and linked to the backend component (a server linked to a database inventory system), while the tags are dispersed in the environment.

The backend component drives and initiates the discovery process of tags in the environment. The readers return their results to the server when the discovery process is completed. While the backend components that controls the readers serves a vital role in mapping the discovered information from RFID tags to respective products, the scope of this paper focuses primarily on the discovery process between RFID readers and RFID tags.

Although RFID has been introduced and deployed commercially since World War II for identifying aircraft, RFID technology has only recently been gaining adoption of a much larger scale within the non-aviation sectors. The facilitation of this proliferation has been mostly driven by technological advances that contributed to the shrinking size of microchip, as well as the pervasive use of inter-networked communications among businesses and consumer homes.

Many companies have identified the emerging potentials of RFID technology, and are exploring the potential RFID applications in their environment. For example, some corporations with large amounts of inventories are beginning to deploy RFID

technologies within their supply chain, hoping to benefit from the increased efficiency in managing and tracking inventory. Package delivery companies are also exploring the possibility of using RFID tags to track the locations of packages. Other companies that produce small and high priced products, such as Gillette, are using of RFID technologies at retail stores to lower theft [4].

1.2 Acceptance Roadblocks

The current leading factor inhibiting the ubiquitous use of RFID technology is the price of RFID tags. Although the use of passive powerless tags - a category of tags that are powered by radio frequency signals only - are less expensive than powered tags, the cost of passive tags is still a deterrent for most potential customers. In a supply chain environment, the potential customers that benefit the most from RFID are the customers that need to track and identify a large number of physical objects, which would bring the total investment to a significant amount. Therefore, the current use of tags in commercial applications has been relatively small-scaled and constrained in enclosed environments, where tags are detached and reused. While this form of application still brings an increased level of efficiency in tracking physical objects, there is much larger economical benefit to realize if the physical objects are bound to uniquely identifiable tags for their lifetime.

The following factors would contribute to lowering the overall costs of RFID tags:

Large company sponsors who show a dedicated commitment to deploy RFID tags in their environment. A strong commitment will convince RFID equipment manufacturers to invest in sufficient manufacturing capabilities to meet the anticipated strong demand, therefore lowering the price of RFID tags through economies of scale.

Manufacturers that adhere to standardized RFID communication protocols. Manufacturers that agree to adopt an open and published protocol will ease a purchaser's concern of a proprietary lock-in. The interoperability between readers and tags of different manufacturers will also drive down the price of a tag through

competition and commoditization of the physical tags.

1.3 Auto-ID Center

The Auto-ID Center was founded with the vision to revolutionize the way products are made, bought and sold, by labeling and connecting physical objects together with computers. The main goal of the Auto-ID Center is to define mature and reliable communication protocol standards for manufacturers to adhere to, driving the acceptance of RFID in the world.

Auto-ID Center is an industry-funded research program. The Auto-ID Center's sponsors and partners include market-leading companies from a diverse group of industries, such as Wal-Mark, Gillette, Department of Defence (DoD), US Postal Service (USPS), Phillip Morris, Procter & Gamble, Home Depot, UPS, Intel, PriceWaterhouseCoopers, Sun Microsystems, and Pepsi Co.

1.4 Problems

The Auto-ID initiative is still relatively young, and official protocol standards have only been recently published. In order to drive the acceptance of RFID technology, it is important to perform rigorous testing on the published protocol standards, as well as continually improving them based on the needs of the customers.

There are many development directions the protocols can take, especially in the areas of security, robustness under noise, and speed. However, there are two key factors slowing down the algorithm development process.

1.4.1 Algorithm evaluation methods

There are currently four methods to evaluate new discovery algorithms: theory, emulation, simulation and implementation.

The performance of discovery algorithms can be evaluated based on the theoretical runtimes. While theoretically we can approximate the performance

of a discovery algorithm, the approximation is not rigorous enough to prove the actual performance gain. The theoretical analysis must make several simplifying assumptions to become tractable to solve. These simplifying assumptions yield a theoretical model that can be far from reality.

The performance of discovery algorithms can be evaluated using a software simulator. The software simulator can be made very detailed and accurate for the tag-reader communication protocol, the signaling, and the environment. However, for a detailed simulator, long execution times are required to obtain a single point of data. A highly accurate simulator can realistically model reality; however, the number of simulated data points possible in a given period of time decreases, typically exponentially, as the accuracy of the simulator increases. Fewer data points provide limited analysis potential of the protocol.

The performance of discovery algorithms can be evaluated using programmable prototype tags in a lab environment. The use of programmable tags to change the reader and tag logic can provide more realistic evaluations of the discovery algorithms in a given environment. However, programmable prototype tags are often too inflexible to allow for drastic changes to the communication protocol, and it's only suitable for minor algorithm changes. These tags also provide unrealistic performance as compared to pure passive tags. This performance boost can mask potential problems with a silicon implementation.

The performance of discovery algorithms can be evaluated using manufactured prototype tags in a lab environment. By communicating with the manufacturer and requesting small batch quantities of prototype tags, it is possible to evaluate the major design changes that programmable tags do not allow. However, the manufacturing of such prototype tags in small quantities expensive and time consuming. Also, the long delay incurred from the manufacturing process of new tags lengthens the development cycle significantly. Furthermore, results obtained from the testing of tags in a lab environment can be unreliable and inaccurate, as even in lab environments there are many uncontrolled variables from noise interference that might skew the evaluation.

While each method has respective drawbacks, a well designed software simulator that strikes a fine balance between accuracy and performance will still provide more accurate, timely and controlled testing results than the other three available methods.

1.4.2 Current simulation solutions do not fit the needs

The RFID simulators currently available are supplied by the RFID manufacturers themselves. However, the simulators do not have the level of signal modeling we require. Some of the simulators do not factor in the distance between the readers and tags, and do not even simulate on the signal level. Both of these characteristics can have a significant impact on the performance of a protocol and must be evaluated.

The general network simulators currently available, such as *ns-2*, do not take into account the unique asymmetric nature of RFID communication. As a result, using such network simulators would require a large overhaul in the core assumptions of the system.

1.5 Motivations behind creating *RFIDSim*

The development of a new RFID simulator aims to overcome the deficiencies and simplistic nature of existing RFID simulators. Small tweaks or even entire new protocols can be developed on the simulator, and a recompile of the simulator can allow for elementary indications of performance.

The simulator eliminates the need for the use of prototype tags to test minor modifications to the algorithm, and allows the use of prototype tags only when the new logic of the algorithm is fully matured through testing on the RFID simulator. Such a simulator will also allow testing of tags in large quantities and controlled environments, so uncontrolled noise variables wont distort evaluation results.

A simulator designed to solve these problems will be algorithm-oriented, with detail paid to the level of simulation for signal modeling.

1.6 Thesis Overview

The rest of the thesis is outlined as follows.

Chapter 2 defines and explains the terminology that will be used throughout the thesis. The chapter will also explain the nature and characteristics of RFID communications, the technologies and techniques that readers and tags use to communicate back and forth.

Chapter 3 defines the design goals outlined in the early development of *RFIDSim*, taking into account the problems and motivations described in section 1.4 and 1.5.

Chapter 4 presents the final design of *RFIDSim*, which includes a design overview, a description of all the significant actors in a simulation, an explanation on how signal modeling is handled in simulations, and a complete walkthrough on how communications between RFID readers and tags are simulated.

Chapter 5 discusses the implementation of *RFIDSim*'s design, with the focus on specific implementation details that were not relevant to the overall design.

Chapter 6 explains the requirements a user needs to compile and execute *RFIDSim* successfully, the features of the *RFIDSim* user interface, and the proper steps a user should take in order to run simulations on *RFIDSim*.

Chapter 7 summarizes the work done thus far on *RFIDSim*, and cites areas where *RFIDSim* can be extended or improved as potential future work.

Appendix A outlines the design of the Auto-ID Class 0 protocol. For the detailed specification of the protocol, please refer to the published specification [5].

Appendix B outlines the design of the Auto-ID Class 1 protocol. For the detailed specification of the protocol, please refer to the published specification [6].

Chapter 2

Terminology

2.1 Discovery Process

In a given RFID environment, the key objective is to accurately discover the identification of all tags in the environment, using interrogation processes initiated by the reader. The very nature of an RFID system places many constraints of how the readers and tags can effectively communicate with each other.

2.2 Readers

Readers are devices used to identify RFID tags in a given environment. They are powered devices that can transmit signals over a selected channel, and can also listen to selected channels for responses from tags. When a reader sends a signal, every tag in the vicinity will pick up the command and attempt to execute the command.

2.3 Tags

Tags are powerless and passive chips with antennas, and each tag has a Identifier Tag Memory (ITM). The unique identification code (Electronic Product Code, also known as EPC) is embedded in the ITM.

Tags are powerless, and they do not have any form of batteries attached to the tags,

and are powered by the wireless signal emitted by the readers during the discovery process.

Tags are also passive, so they have no transmitting capabilities. The only way for tags to communicate with a reader is by backscattering its response during a carrier wave signal from a reader. As a result, the communication channel they reply on will be relative to the channel the reader is communicating on. The reader listens to those channels for the backscatter replies.

2.4 Asymmetric communication

The communication between RFID readers and tags is asymmetric, with readers initiating interrogation commands and tags responding passively. The following factors contribute to RFID communication's asymmetric nature:

2.4.1 Restrictions imposed on wireless communications

Existing regulations limit the reader-to-tag communication bandwidth, but not the tag-to-reader communication bandwidth. Therefore, while the reader is restricted to communicating over a narrow pipe for communication, the tag has no such restrictions, and has the potential to communicate more information in the same period of time when compared to the reader. Conversely, the tag only has access to a weak energy source harvested from readers, while the reader communication strength is significantly more powerful than the tag communication strength.

2.4.2 Tags are powerless / stateless

Tags do not have an energy source without an interrogation by the reader. The lack of an internal energy source places a constraint on the ability for tags to maintain their state. A temporary state can be maintained if the tag's energy source is continually renewed by reader signals, but without power, the state will only for hold for a limited amount of time.

2.4.3 Tags are passive

The tag cannot initiate commands because it is powerless, therefore it only communicates when powered by readers. As a result, all communications are initiated and driven by the reader, and a tag can only reply to the reader during an allotted time period by the reader itself.

2.4.4 Reader signals are broadcasted to all tags

Although the strength of a reader's antenna and the direction of the antenna can be used to allow the reader to broadcast a command to only a group of tags in a certain direction, it still does not allow the reader to accurately communicate with one single tag at a time.

Because of the broadcasting nature of the reader's signals, any reliable attempts to selectively communicate with a single tag or a selected group of tags will be performed at the protocol level. (For example, the tag can be logically programmed to behave differently based upon tag's unique identity)

2.4.5 Tags communicate with the reader through a channel relative to the reader

Because all tags will be replying through the same channels, the replies could potentially generate collision problems. Therefore, in order to extract information from the backscatter channels, the design of the protocol will have to take into account of the possible collisions with backscatter replies.

2.4.6 Reader has complete control of the communication

Because tags are passive, powerless, stateless, and unable to initiate communication with the reader, the tag is often required to respond with specified timing bounds. Therefore, the reader is only listening during predefined time periods. Also, because

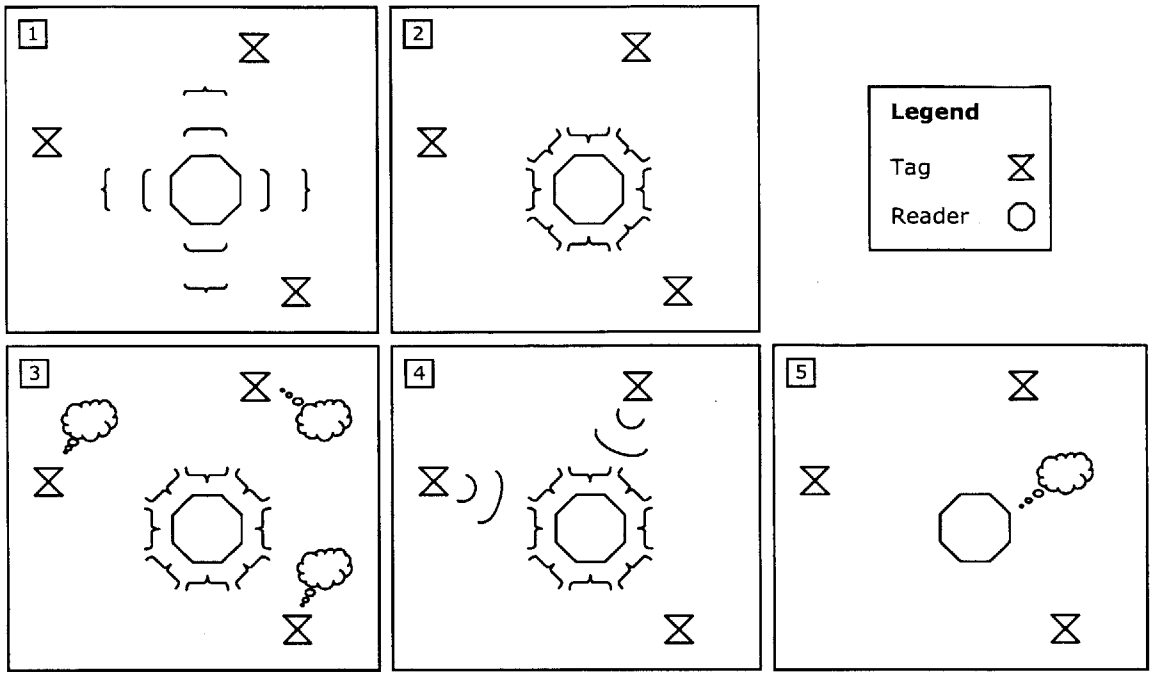

Figure 2-1: The generalized communication between reader and tags.

readers are not limited by energy, memory or computation power, the discovery algorithm is driven by the reader.

The following communication shown in figure 2-1 is how readers and tags communicate for all RFID protocols:

1. The reader broadcasts a signal.

2. Reader waits for a reply for a timed period (if it is expecting a response)

3. Tags are powered from receiving the signals, and logically processes the command.

4. Tags backscatters a reply or stays silent depending on the command.

5. Reader analyzes any backscattered responses receives at the end of timed period.

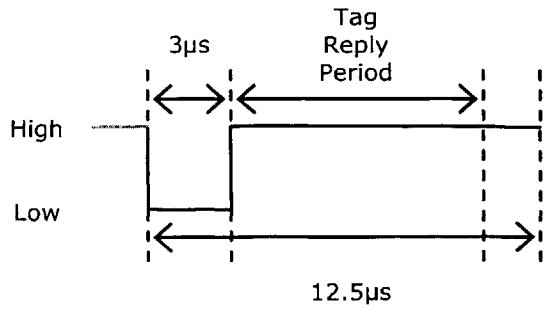

Figure 2-2: Signal diagram of zero data symbol in the Auto-ID Class 0 protocol.

2.5 Signaling methods

The encoding of symbols in RFID communication is different for reader communication and tag communication, due to the computing capability differences of readers and tags.

2.5.1 Reader Modulation

Because tags have simple functionality, they are not capable of receiving complex communication signaling from a reader. The readers in RFID protocols primarily uses Amplitude Shift Keying (ASK) to modulate the commands to the tags, because ASK modulation is detectable with very simple circuitry.

In ASK modulation, amplitude shifts in the signals are used to indicate a high or a low signal, and the rising edge and the falling edge is used to modulate and demodulate the signals.

In addition to modulating commands using ASK, the amplitude rise and falls are also used to synchronize the clocks of the tags.

Figure 2-2 shows a signal diagram of a zero symbol in the Auto-ID Class 0 protocol.

After the modulation of the signal, if the reader was expecting a backscatter reply from the command, the reader allots a time period where it sends at carrier-wave frequency, allowing the tags to backscatter.

2.5.2 Tag Demodulation

After the tag's clock has already been synchronized from an earlier signal, the tag evaluates the signals based on the timings of the amplitude rise and falls of the signal. The tag then logically process the command presented by the reader, and if necessary, generates the necessary backscatter response to the reader in the allotted time period.

2.5.3 Tag Modulation

Because the tag backscatter rests upon the carrier-wave of the reader, the response is usually fairly weak and low in energy. As a result, different protocols have implemented different tag modulation methods. The details of the tag modulation methods will be discussed in the appendices A and B, and can also be referenced in the published specifications [5] [6].

2.5.4 Reader Demodulation

During the allotted time period, the readers listen on certain channels for any backscatter responses (the channels vary depending on the protocol design). The resulting backscatter is then demodulated and evaluated.

2.6 Constraints

The nature of how readers and tags communicate in each other (described in section 2.4) has placed several constraints on designing a systematic method to identify tags during the discovery process. This section will outline the constraints, as well as describe a generic discovery algorithm designed to take these constraints into consideration.

Constraint: Reader signals are broadcasted to all tags - all tags are logically programmed to behave in the same behavior, and there is no effectively way to target the communication at one single tag or a selective group of tags. Therefore, every tag will respond in the same behavior with every reader command sent.

Solution: While the logical behavior of every tag is identical, they are uniquely identified by their identification code. The identification code can be used to logically instruct the tag to behave differently. For example, a reader signal can be broadcasted with the instruction "Only reply if your identification code is 0111010011101101111". The identification code can also be used as a seed for introducing random numbers into the algorithm, which will also create uniqueness among the tags.

Constraint: Backscatter response from tags can be subject to collision - tags are designed to backscatter in a channel that's relative to the reader's signal. Therefore backscatter responses from tags are subject to collision, and any information encoded in the backscatter response will be lost. With an unknown number of tags in the field and limited frequency allocation, it is not realistic to instruct tags to all reply in their own individual channel.

Solution: Using the solution described above, tags can be instructed to reply only if their identification code matches or does not match a certain pattern. Also, as we will see with the protocol design of Class 0, the encoding of the information does not necessarily have to be modulated in the backscatter response. In the design of Class 0, tags can backscatter in one of the two available channels, and the information is encoded in the decision of which channel the tag chooses to reply on. Therefore, for each reader command, there is two bits of information encoded in the reply, regardless of collision.

Constraint: Tags are powerless / stateless - The powerless nature of tags places a constraint on the ability for tags to track state effectively. Any states held by the tag will be lost shortly without a signal from a reader.

Solution: The reader should bear the responsibility of tracking as much state as possible. The discovery algorithm used by the reader should also ensure that there is enough power to continuously power the tag for the duration of the discovery algorithm, in order to preserve the few internal states the tags might keep track of.

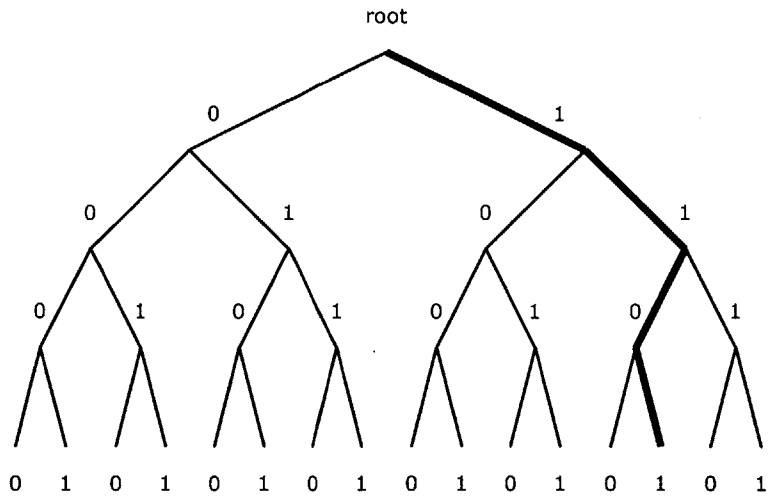

Figure 2-3: Binary string "1101" represented as a binary tree path.

2.7 Discovery Algorithm using Tree-Walking

This section describes a fairly elementary tree-walking algorithm that discovers and identifies all listening tags in the environment. The algorithm performs tree-walking on a binary-tree based on the uniqueness of the tags' identification codes.

2.7.1 Using identification codes for tree construction

Each RFID tag's identification code is encoded in binary and has a fixed length l. If we construct a binary tree of depth l, using the MSB of the identification code as the root, and define the left branch as bit "0", and the right branch as bit "1", each tag's identification code is a tree-walk from the root of the tree to a leaf. Furthermore, the uniqueness of the identification codes guarantees that at most only one tag will reach any leaf. Figure 2-3 shows a binary tree of depth 4, and the path tag with the identification code "1101" have been taken.

2.7.2 Tree-walking algorithm

The following reader algorithm allows a reader to walk from the root of binary tree to a leaf node, effectively identifying a single tag. The process of successfully identifying

a single tag is known as singulation.

discovered_id = "";

d = lengthOf(discovered_id);

while (discovered_id is not complete) {

Reader announces: "I am now at level d. those tags that begin with discovered_id please respond with your next bit.";

if (bit0 == yes) // there is a tag that matches the path, and the tag's next bit is 0

discovered_id = discovered_id + "0"; // move down the "0" branch

else if (bit1 == yes) // there is a tag that matches the path, and the tag's next bit is 1

discovered_id = discovered_id + "1"; // move down the "1" branch

else

terminate; // no tags replied

}

// we are here, and we did not terminate, so discovered_id is complete

Reader announces: "I have discovered a tag with id discovered_id";

Analysis - the reader attempts to walks down the path of an identification code of tag. As the reader walks down, it repeatedly interrogates all the tags to provide him with the next bit of the constructed string, which is the next branch of the path the reader has taken so far. Tags that do not match the path that the reader has taken do not respond. If the reader only gets a response from one branch, that means there is at least one tag with an identification code that walks down that specific branch. The algorithm chooses branch "0" first if available, and terminates at the discovery of one completed identification code. A completed algorithm would silence the discovered tag, and repeat the above algorithm until all tags are silenced (discovered).

The discovery algorithms used in Class 0 and Class 1 uses more advanced versions of the tree-walking algorithm, but the fundamental idea is similar.

Chapter 3

Design Goals

The ultimate design goal is to encourage the use of the simulator to facilitate the testing of new protocol modifications or protocol designs. In order to enforce this goal, it is important to design the simulator to be easily modifiable, maintainable, and extensible.

3.1 Level of abstraction between logic and signal modeling

One of the goals derived from the key objective is to create a high level of abstraction between the logical components and the signal modeling components of the simulator.

3.1.1 Developers should not have to understand everything

One of the observations made early on in the design of the simulator is the different domains of knowledge required for the logical implementation and the signal modeling implementation. If the two components were heavily intertwined in the implementation of the simulator, the learning curve to understanding how the simulator works would be fairly steep. It would also be difficult for a developer to improve the simulator, if modifications to one component of the simulator breaks several other unrelated components. Creating a level of abstraction between logic and signal mod-

eling components the simulator allows different components of the simulator to be independently modified and extended.

3.1.2 Modifying protocol logic without knowledge of signaling assumptions

The logic implementation of the reader / tag (algorithms) belongs to a different domain of knowledge compared to the signal modeling portion of the simulator (signals processing). As one of the primary uses of the simulator is to facilitate the testing of new protocol modifications, it should be assumed that the logical algorithms will be modified the most. Therefore, it is important that a person extending or modifying the logic portion of the reader / tag does not have to bother with the details of how the signals are modeled deep inside the simulator.

3.1.3 Changes in signal modeling assumptions should not break the protocol logic

The signal modeling component design and implementation should not be influenced by assumptions of any particular protocol design, as such assumptions might possibly break the protocol when signal modeling assumptions are modified. It might also create the possibility of generating skewed simulator performance results for that particular protocol.

Therefore, the signal implementation should remain as generalized as possible, allowing the logical components of the simulator to function properly, and not rely too heavily on underlying assumptions.

3.1.4 Complete models or objects can be replaced

It is possible to envision that for testing purposes, some components might be replaced with more light weight components to allow for simulations of a larger amount of tags, or some components might be replaced with a more accurate representation of the

signal waveform. By designing the components to be as modular as possible, and creating a level of abstraction around the components, different components can be replaced or extended to without breaking too many assumptions.

3.1.5 Code clarity

Core logic components, such as the discovery algorithm of a reader, or the state transition algorithm of a tag, should be abstracted as much as possible, in order to improve readability at the expense of verbosity and redundancy. The simulator implementation should allow a developer to modify the reader or tag logic without implicit knowledge of the signal's internal data structure, or the general framework of the simulator. Commands sent by reader and backscattered responses by tags should be abstracted into simple methods in the reader and tag logic.

3.2 Support for future protocols

Another goal that follows the key objective is to create an extensible framework that allows future protocols to be easily implemented. Code that can be shared or has a common foundation (mostly residing in the signal modeling components) should be grouped together and exposed to all protocols.

3.2.1 Consistent simulation results

The accuracy of the simulation is important for testing different protocols. Therefore, a large foundation of shared methods and objects would help improve the consistency of the behavior, as well as allow the user to evaluate different protocols fairly in the context of the simulator.

3.2.2 Shorter implementation periods for new protocols.

Because most RFID discovery protocols share a lot of fundamental characteristics and behaviors, a growing library of generic components would also contribute to

increasingly shorter development times to implement other protocols.

Developing new protocols should involve identifying what generic components can be reused, and implementing the unique features offered only by the new protocol.

3.3 Comprehensive analysis tools for simulation results

The final design consideration is to provide comprehensive analysis tools to analyze simulation results, as rare unexpected behavior in the protocol design or the simulator itself sometimes appear in only very unique conditions.

3.3.1 Controlled environments for testing

The simulator design should be deterministic, allowing simulation runs to be replicated given identical input variables.

3.3.2 Extensive logging and debugging features

Extensive logs of communications and debugging tools should be provided by the simulator to assist the user in tracking down the cause of such behavior.

Chapter 4

Design

This chapter presents the design of *RFIDSim*.

4.1 Process-oriented Discrete Simulator

RFIDSim is a process-oriented discrete simulator. A discrete simulation is a form of simulation where all actions within the simulation can be modeled in discrete points in time. For example, a signal is broadcasted from the reader at time t, and is received at time $t+k$ by a tag. Discrete simulations are different from continuous simulations, which are more suited for simulations such as fluid dynamics, where changes happen continuously and can only be modeled by equations.

RFIDSim is process-oriented, as opposed to being event-oriented. The change of state in an event-oriented discrete simulations are driven by events that occur in discrete points of time. Changes in a process-oriented discrete simulation are driven by actions of the individual processes towards one another. While the difference between the two types of simulation almost seems nominal (it is true that any process-oriented simulation can be modeled as a event-oriented simulation and vice versa), the focus of the simulation is the most important factor to consider. Because the main actors of an RFID simulation are the readers and tags, and the discovery process focuses on how readers and tags change the state of each other, *RFIDSim* is process-oriented.

Every action in the design of *RFIDSim* is a process, and has its own life cycle

that begins at a certain time, and ends at a certain time. Even the simulation of a signal arriving at a reader or a tag is a process, as the life cycle begins when the tag begins to receive, and terminates when the tag has received the entire signal.

4.2 Design Overview

The components of *RFIDSim* can be categorized in three different dimensions - by function, level of generality, and level of participation.

4.2.1 By function

The following are the major groups when categorized by function:

Logical

Components that belong to this category simulates at the logical level, such as the reader object and tag object. The logical level is defined by the logical commands used by the readers and tags, as well as the algorithms that contribute to the discovery process of tags in an environment.

Signal

Components that belong to this category simulates at the signal level, such as the signal library. The signal level is defined by the low-level modeling of radio-frequency waves (generated by readers and tags), as well as the edge detection, noise and distance functions.

System

Components that belong to this category are part of the backend of the simulation framework, such as the simulation logging component and the model component.

4.2.2 By level of generality

The following are the major groups when categorized by level of generality:

General

Components that belong to this category are objects that are protocol-agnostic, or abstract objects that provide the core functionalities, but lack the specific algorithms to execute those functionalities. Components in this category include the edge detection function, the signal object, and the abstract reader object that implements the broadcast function.

Protocol

Components that belong to this category are specific to the protocol implementation, that are either components that inherited functionality from a component in the general category, or components that are too unique and specific to the protocol to belong in the general category. Components in this category include the Class 0 reader object, and the Class 1 logic library.

4.2.3 By level of participation

The following are the major groups when categorized by level of participation:

Actors

Components that belong to this category are processes in the simulation. Such components begin their life cycle at some discrete time in the simulation, and terminate at some discrete time. Components in this category include the reader and tag objects, and the tag receiver object.

Non-Actors

Components that belong to this category are objects that serve some role in the simulation, but they are not processes that changes state or directly causes state

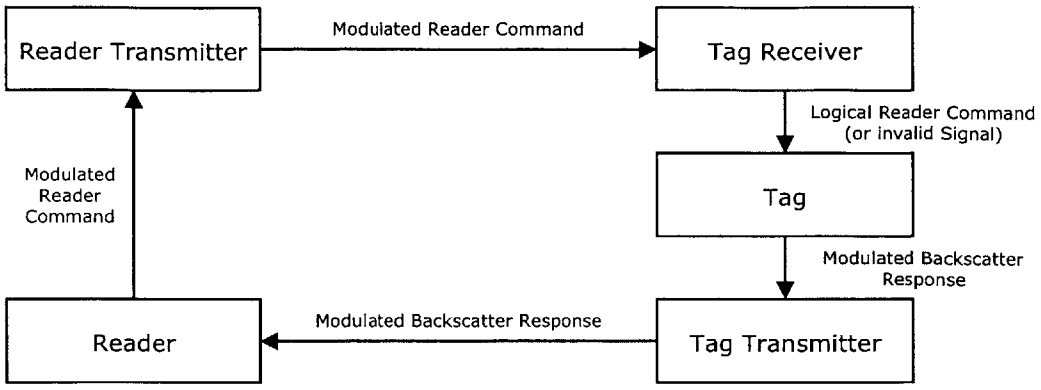

Figure 4-1: The interaction among the five main actors.

changes in other processes. Components in this category include the signal and ITM objects.

Supporting backend

Components that belong to this category are merely libraries of functions and variables that other actors and non-actors call, or they perform roles such as initializing the simulation. They are not objects that serve any role in the simulation. Components in this category include the signal library, or the Class 1 logic library.

4.3 Significant actors in the simulation

The significant actors in the simulation are either the representation of a reader or tag, the representation of some component of a reader or tag, or the representation of an action in progress initiated by a reader or tag.

Figure 4-1 shows an overview of the communication loop formed among the five actors - reader, reader transmitter, tag, tag transmitter, and tag receiver.

4.3.1 Reader

The reader simulates the logical algorithm of the discovery process, and is spawned at the beginning of the simulation. When the life cycle of a reader begins, the reader

immediately begins the discovery process. The life cycle of a reader terminates when the discovery algorithm terminates.

The reader interacts with the reader transmitter when the reader is ready to broadcast a signal to the surrounding tags, and with the tag transmitter when the tag transmitter adds the signal to the buffer of the reader.

4.3.2 Reader Transmitter

The reader transmitter simulates the process of a signal being received, sent from one reader to one tag. Because of the delays of radio-frequency signals due to distance, and different distortions of the signals due to noise, for each command a reader sends out, there are n reader transmitters spawned, where n is the number of tags in the field. The life cycle of the reader transmitter begins when the signal from one reader is scheduled to reach the targeted tag, and ends when the reader transmitter adds the signal to the buffer of the tag receive and notifies the tag receiver of a new signal in the buffer.

The reader transmitter interacts with the reader when the reader is ready to broadcast a signal to the surrounding tags, and with the tag receiver when the reader transmitter adds the signal to the buffer of the tag receiver.

4.3.3 Tag

The tag simulations the logical algorithm of the state machine during the discovery process, and is spawned at the beginning of the simulation. When the life cycle begins, the tag goes to sleep, which is completely different, definition-wise, from the RFID protocol definitions of "dormant" or "quiet". The life cycle of a tag never terminates (unless a permanent "kill" or similar command is sent, of which it is sent on a passive infinite loop that ignores all logic commands received).

The tag interacts with the tag receiver when the tag spawns the tag receiver at the tag's initialization, and interacts with the tag transmitter when the tag is ready to broadcast a backscatter response.

4.3.4 Tag Transmitter

The tag transmitter simulates the process of a signal being received, sent from one tag to one reader. Because of the delays of radio-frequency signals due to distance, and different distortions of the signals due to noise, for each backscatter response a tag sends out, there are n tag transmitters spawned, where n is the number of readers in the field. The life cycle of the tag transmitter begins when the backscatter response from a tag is scheduled to reach the targeted reader, and ends when the tag transmitter has added the signal to the buffer of the reader.

The tag transmitter interacts with the tag when the tag is ready to broadcast a backscatter response, and interacts with the reader when the tag transmitter adds the signal to the buffer of the reader.

4.3.5 Tag Receiver

The tag receiver simulates the receiver component of a tag, and the tag receiver receives all incoming signals. Therefore, there is only one corresponding tag transmitter process for each tag in a simulation. The life cycle begins when the tag initializes itself and spawns the tag receiver at the beginning of the simulation. The life cycle of a tag receiver never terminates.

The tag receiver interacts with the tag when the tag spawns the tag receiver at the tag's initialization, and interacts with the reader transmitter when the reader transmitter adds the signal to the buffer of the tag receiver.

4.3.6 Reader Receiver

There exists no reader receiver actor, process, object or component in *RFIDSim*. The reason is because of the asymmetric nature of the communication (refer to section 2.4.6). When a tag receiver receives a signal in its buffer, the process immediately preempts and attempts to evaluate the signal. However, a reader generally waits a period of time for all tag backscatter responses to collect in the buffer before evaluating the signals. Therefore it's sufficient to let the reader process handle the signal buffer

after waiting a predefined period of time.

Tags have receivers for several reasons. The tag's clock speed is calibrated by the reader commands, so the demodulation depends on timings decided by the reader during calibration. As a result, moving the signal handling code of the tag process into a separate tag receiver process not only improves code clarity, but also maintains the level of abstraction between a logical component (tag process) and a signal component (tag receiver process).

Furthermore, the separation is necessary because of the way different protocols handle invalid signals. For Class 0, an invalid reader command influences the internal state of the tag and the evaluation of future reader commands. Therefore, when a Class 0 tag receiver process interprets an invalid signal, it passes the invalid response back to the tag process to perform a state transition. For Class 1, an invalid reader command does not affect the internal state of the tag. Hence, when a Class 1 tag receiver process interprets an invalid signal, it simply ignores the signal and awaits the next reader command.

4.4 Significant non-actors and supporting functions

The following components in *RFIDSim* are not processes that exists on the simulator timeline, but they serve critical roles during the discovery process.

4.4.1 Signal

A signal object is a representation of a radio-frequency wave sent from a reader or tag to another. As a result, a signal object includes information such as:

- The energy / amplitude level of a signal at a given time
- The time when the signal is sent by an actor
- The time when the signal is received by an actor
- The length of the signal
- The owner that created the signal
- An internal message ID

- The channel the signal was sent in

The signal object also includes several functions that manipulate signal objects:

- Noise function

- Amplitude scaling function (used for factoring energy decay in distance function)

- Signal merging function (combines two signals into one - the function accurately concatenates continuous signals that were broken up into one continuous signal)

4.4.2 ITM

An ITM object is a representation of the Identifier Tag Memory. It contains the complete binary strings of the EPC and CRC, and also provide several functions related to ITMs (CRC computation / verification, parity bit computation / verification)

4.4.3 Signal Library

The signal library object is one of the most significant objects in the RFID simulator. The signal library provides abstracted functions for modulating signals, and abstracted functions for demodulating signals. The library also supplies to readers and tags the variable parameters defined in protocol specifications.

The signal library also contains signal manipulation functions (which indirectly calls the signal manipulation functions in the signal object), such as noise and distance functions, as well as edge detection functions.

The consolidation of all the signal modulation / demodulation / detection / manipulation functions into one object helps improve code clarity, as well as provide an additional layer of abstraction.

4.4.4 Logic Library (Class 1 only)

The logic library object is exclusive to Class 1, because the format of the Class 1 reader commands and tag backscatter responses are highly structured and bit-oriented. As a result, there is a relatively large number of construction and deconstruction binary-string manipulation functions for Class 1. Because it makes no sense from a design or

semantic point of view to include these binary-string functions into the signal library object, a logic library object is created to house these functions.

4.5 Signal Handling for Tag processes

In any given RFID discovery process where reader commands are modulated in ASK, a tag demodulates and identifies the reader command by the time intervals between the falling edges of the signal and the subsequent rising edges. The edge detection is handled by a sudden fall or sudden rise in the energy level of the signal being received by the tag. However, in a discrete event simulator, it is computationally unrealistic to model each tag process to detect the rising and falling edges by monitoring the energy level in real-time.

4.5.1 Looking ahead

Because a tag receiver actually receives the entire signal in full when the signal object arrives in the tag receiver's buffer, a computationally less-expensive edge detection algorithm would involve looking ahead at the entire signal for the next rising edge or falling edge, and reporting back to the tag the amount of time left before the next amplitude shift would arrive. This solution would allow the tag receiver to start an internal timer when the algorithm reports a falling edge, and allow the tag receiver to stop the internal timer at the time the signal is supposed to rise again.

However, such a solution is still unreliable and unrealistic for two main reasons:

1. In simulations where there are multiple readers in the environment, a signal broadcast by one reader might be sent to the tag receiver, while the tag receiver is already holding the timer for a previous signal sent by another reader. In a real tag, the tag receiver will instantly detect the extra amplitude increase gained from receiving another signal, and evaluate the resulting signal appropriately (even though there is a high probability that the reader command is now invalid). In our current solution, the tag receiver has no accurate way of simulating this behavior, since the edge detection algorithm only reports to the tag the time before the next amplitude

shift, and does not report the energy level received so far.

Even if we modify the algorithm to somehow the tag receiver whether it should start / stop a timer because a new signal has created a spontaneous rise in this instant, it is very difficult to verify the correctness of such an algorithm. With more than two readers, it is hard to guarantee that the algorithm won't get confused and lose track of when the next amplitude shift will happen.

2. Assuming that the edge detection can be implemented and proved correctly, the burden of identifying and demodulating an incoming signal would lie on the tag receiver algorithm, including signals such as the Class 0 calibration signal, which includes dozens of rising and falling edges. The resulting tag receiver implementation would be extremely long (because in-between every rising and falling edge, there could be incoming signals from other readers), extremely confusing, and almost impossible to debug.

4.5.2 Preemptive Signal Handling

RFIDSim attempts to solve the problem of reliable edge detection by preemptively processing the entire signal (looking ahead in time), and reprocessing the signals received in the past (looking back in time) when needed. Figure 4-2 presents an illustrated example of Preemptive Signal Handling works.

Looking Ahead

When the tag receiver first receives a signal in its buffer, it runs the edge detection algorithm on the entire signal. The edge detection algorithm returns to the tag receiver an array containing the absolute time values of all the rising and falling edges in the entire signal.

Given all the rising and falling edges of the signal, the tag receiver runs various demodulation / verification functions from the signal library object. If the signal can be properly identified as a reader command, the tag receiver will preemptively assume that there will be no more incoming signals, stores the current timestamp

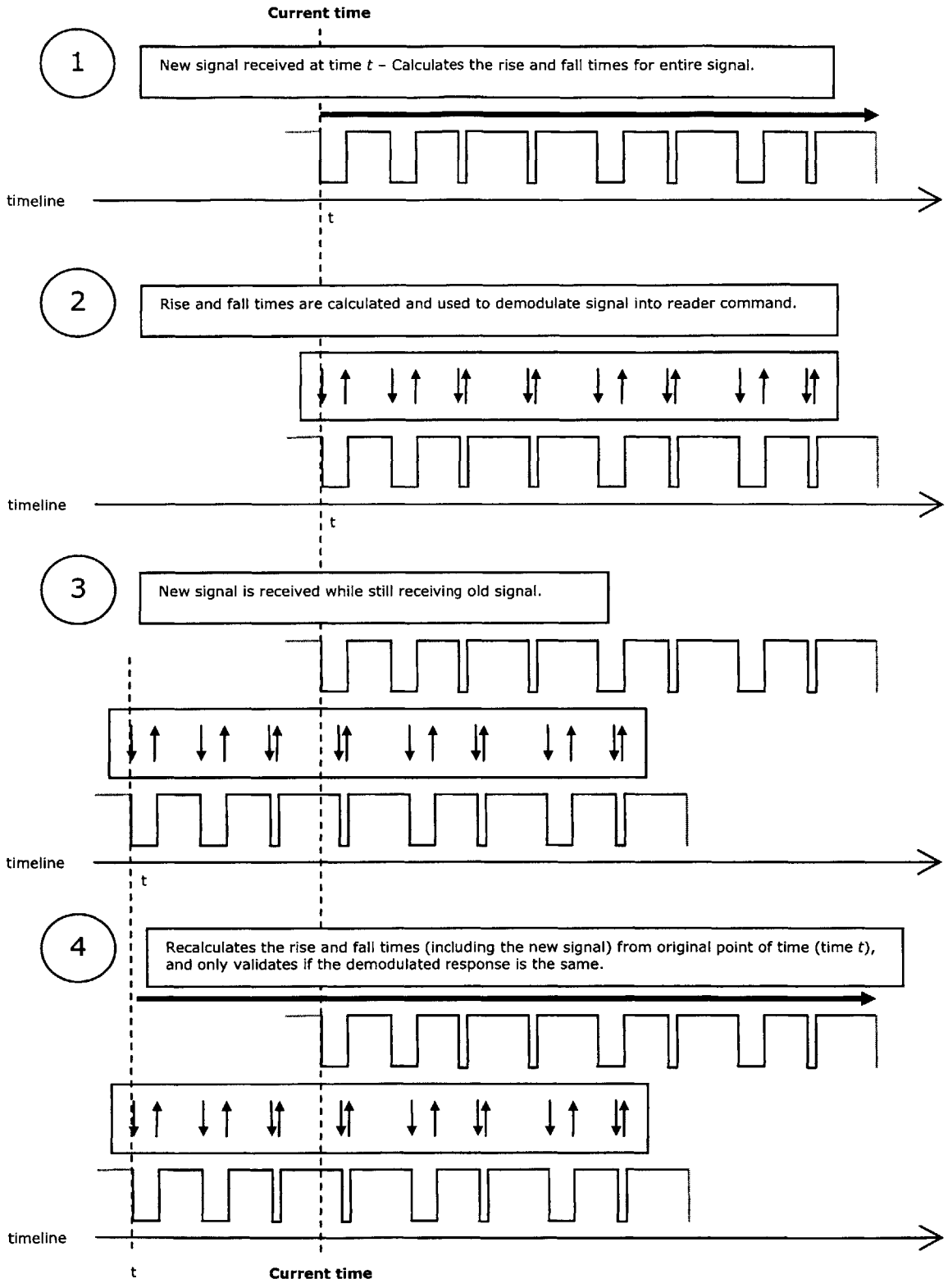

Figure 4-2: Illustrated walkthrough of Preemptive Signal Handling

into memory, and proceeds to wait until the end of the current signal.

If there are no new signals during the duration of the current signal, the tag receiver successfully alerts the tag process at the end of the current signal.

Looking Back

However, if the tag receiver is alerted to a new signal in the buffer before the end of the current signal is reached, the tag receiver will merge the two signals, and compute the rising and falling edges of the merged signal starting from the beginning of the first signal. By doing so, the tag receiver is able to analyze the new signal as part of the original signal, therefore successfully simulating the ability to continually monitoring the energy levels.

Also, there is no longer any need to worry about special cases to handle new signals while the internal timer is running. The rise and fall information is passed onto the demodulation / verification functions, which would objectively tell *RFIDSim* what the current evaluation of the signal is now. If the identification of the reader command is the same as the previous, the tag receiver can go back to waiting for the original signal to end. If the signal is invalid, the tag receiver can handle the invalid command appropriately, depending on the protocol implementation. Finally, this also allows the tag receiver algorithm to remain relatively clean and uncluttered.

4.6 Unreachable Tags

Radio-frequency waves are omni-directional, and sometimes a tag will sometimes receive weaker duplicates of the same reader signal that reached the tag from bouncing off a wall. However, in certain situations, the converging copies of the signal will cancel each other out when it reaches the tag, even if the tag is in close proximity of the reader. This general phenomenon is referred to as a *multi-path null.*

In order to simulate this behavior, a reader randomly computes a select percentage (defined by the user) of tags that will be considered unreachable. The selection of unreachable tags is recomputed after a frequency hop.

4.7 Asymmetric communication in *RFIDSim*

This section describes how the asymmetric communication between RFID readers and tags discussed in section 2.4 is modeled in *RFIDSim*. The following walkthrough of the simulation focuses on the repeating loop defined at the end of section 2.4.6.

The reader broadcasts a signal.

The reader calls its internal broadcast function, which takes in a signal generated by a modulation function in the signal library object.

For each tag in the environment, if the reachable lookup table indicates that the tag is reachable in this channel, the broadcast function creates a reader transmitter.

Each reader transmitter (which corresponds to one tag) takes the signal generated earlier, applies the noise and distance function (given the distance between the reader and the tag), and instructs the reader transmitter to sleep for a time period of d, where d is the time delay it takes for the signal to arrive to the tag receiver.

Reader waits for a reply for a timed period if it expects a reply.

The reader sleeps for a timed period if it expects a reply in the buffer.

Tags are powered from receiving the signals, and logically processes the command.

Each reader transmitter wakes up, adds the signal to the buffer of the corresponding tag receiver, alerts the tag receiver of a new signal, and exits.

Tag receiver preemptively handles the signal, and correctly identifies the reader command. Tag receiver waits for a time period of d, where d is the length of the signal.

Tag receiver passes the logical reader command to the tag, alerts the tag of a new command, and returns to waiting for incoming signals.

Tag logically processes the command, and determines whether it should create a backscatter response or stay silent.

Tags backscatters a reply or stays silent depending on the command.

For each that that chooses to backscatter a reply, the tag calls its internal broadcast function, which takes in the signal generated by a backscatter modulation func-

tion in the signal library object. The backscatter modulation function also takes in the channel to respond and the energy level as parameters, which were retrieved from the tag receiver.

For each reader in the environment, the broadcast function creates a tag transmitter.

Each tag transmitter (which corresponds to one reader) takes the signal generated earlier, applies the noise and distance function (given the distance between the reader and the tag), and instructs the reader transmitter to sleep for a time period of d, where d is the time delay it takes for the signal to arrive to the reader.

Each tag transmitter wakes up, adds the signal to the buffer of the corresponding reader, and exits.

Reader looks at any backscattered responses at the end of timed period.

The reader wakes up, and selects all backscattered responses that were sent the channel(s) it's listening to, and evaluates the signals.

Chapter 5

Implementation

5.1 Language and Tools used

Java was chosen as the language for implementation, for various reasons:

5.1.1 Portability

Implementing *RFIDSim* in Java grants the simulator implicit portability to many different platforms, including Linux, Windows, and Mac OS X.

5.1.2 Object-Oriented Nature

The object-oriented nature of the Java language allows the componentized design of *RFIDSim* to be implemented with relative ease.

5.1.3 Mature support for graphical user interfaces

A graphical user interface would significantly enhance the usability of the application, because a visual presentation of the simulated environment would allow the user to place reader and tag objects with a mouse. Also, signal objects, as well as objects that embed the rising and falling times of a signal, can be visually displayed.

The Java Swing package has been in development for over 5 years, and has matured into a robust and flexible graphical package that works on all platforms that support

Java.

5.1.4 The chosen simulator framework is implemented in Java

The DESMO-J discrete simulation framework for Java was chosen as a foundation for *RFIDSim*. The framework consists of an entire package of tools and objects that model a time-line based simulation, with many fundamental simulation principles modeled directly into the framework. Objects that inherit the abstract simulation processes automatically inherit all the features provided by DESMO-J.

The decision to use a third-party framework was made early on in the development process, because adopting a mature simulator framework is considerably less risky than implementing one from ground up.

5.2 Classes Overview

All classes implemented for *RFIDSim* can be categorized in the three dimensions mentioned in section 4.2.

Figures 5-1, 5-2, and 5-3 illustrates the categorization of the Java classes by function, level of generality, and level of participation respectively.

5.3 Common Java classes

This section describes the functionality presented by the Java classes that are shared or inherited by all protocols.

5.3.1 Signal class

The `Signal` class represents a complete radio-frequency signal in the simulation.

The amplitude levels of a signal is represented by an array of double, together with a double scale variable that defines the granularity of the signal (The variable scale has units of (time units / array entry). Therefore the smaller the scale, the

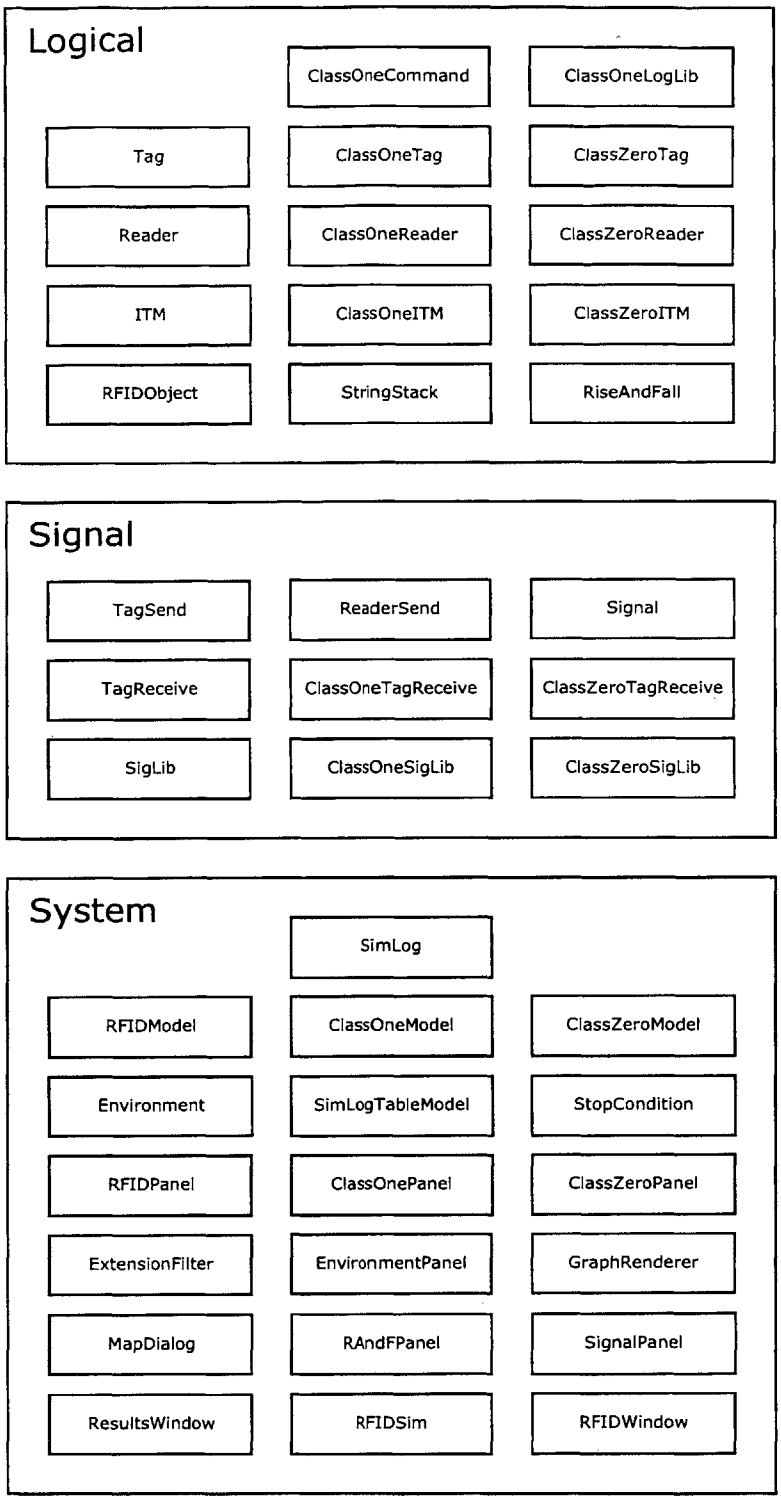

Figure 5-1: *RFIDSim* classes categorized by function

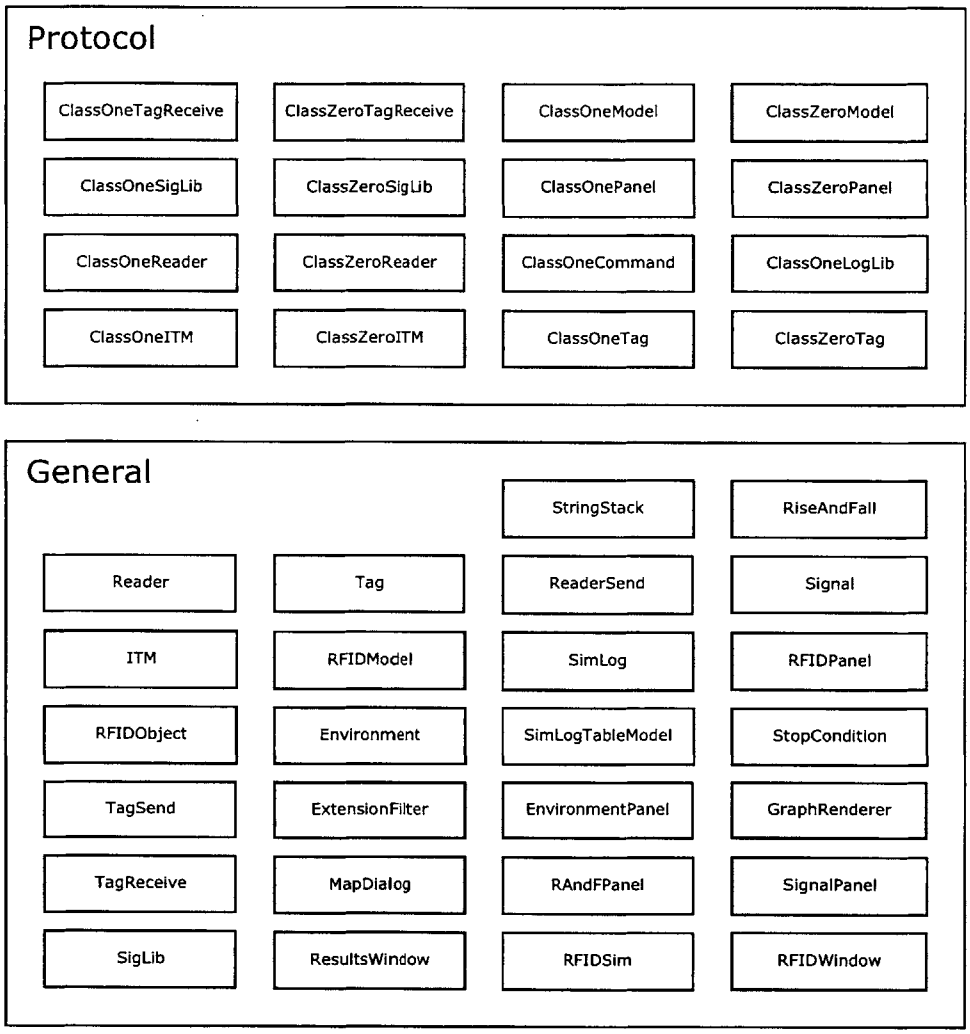

Figure 5-2: *RFIDSim* classes categorized by level of generality

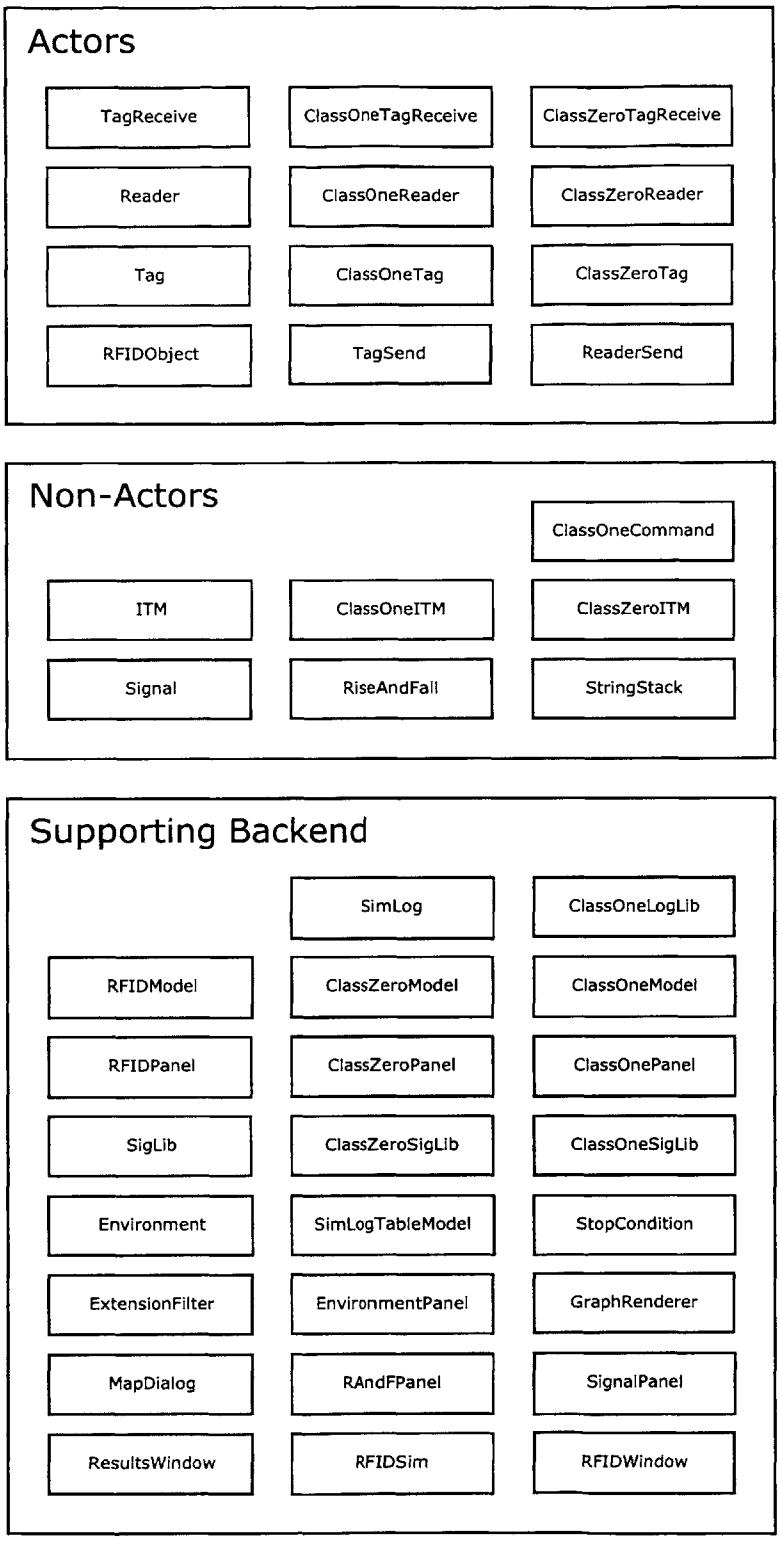

Figure 5-3: *RFIDSim* classes categorized by level of participation

more refined the signal representation will be, but the amount of allocated memory will also increase)

The `Signal` class provides signal manipulation methods to create noise, amplify (or decay) a signal, and merge with another `Signal`.

5.3.2 ITM class

The `ITM` class represents the Identifier Tag Memory of an RFID tag. The abstract class is inherited by another class for a specific protocol implementation.

The `ITM` class provides several methods related that allow CRC computation / verification and parity bit computation / verification.

It also contains the complete binary strings of the EPC and CRC, and an integer value `epcType` that indicates the format of the ITM. The static `epcType` values differ for each protocol, and they are defined in the inheriting classes.

5.3.3 RFIDObject class

The `RFIDObject` class extends the `SimProcess` class, which is part of the DESMO-J simulation framework package. The `SimProcess` class represents an actor in a process-oriented DESMO-J simulation, any actors in the simulation is required to extend the `SimProcess` class and override the abstract `lifecycle` method. The abstract `RFIDObject` class is extended by a `Reader` class or `Tag` class.

The `RFIDObject` class contains the `x`, `y`, `z` locations in the simulation environment. The positions are mainly used to calculate the distance between two `RFIDObject` instances.

The `RFIDObject` class also holds an instance of the `SimLog` class during a simulation run, and provides protected methods for the inheriting `Reader` and `Tag` instances to log a message in `SimLog`. The protected methods automatically logs the identity of the `RFIDObject` (`Reader` or `Tag`), as well as the instance's internal ID.

5.3.4 Reader class

The `Reader` class extends the `RFIDObject` class. The abstract class is inherited by another class for a specific protocol implementation.

The `Reader` class contains the buffer that collects the backscatter responses from `Tag` instances, as well as a Boolean array that determines whether a given `Tag` instance is reachable. The `Reader` class also provides protected methods to frequency hopping, clearing the buffer, and public methods to allow `TagSend` instances to add backscatter responses to the buffer. The protected broadcast method in the `Reader` class is used by the inheriting classes to deliver a noise-processed, amplitude-decayed, and delayed signal to every tag.

5.3.5 ReaderSend class

The `ReaderSend` class begins its life cycle when the `Signal` instance it contains is simulated to reach a `Tag` instance, and terminates after the `Signal` instance is sent to the buffer of the `Tag` instance's corresponding `TagReceive` instance, and the `TagReceive` instance is waken up.

5.3.6 Tag class

The `Tag` class extends the `RFIDObject` class. The abstract class is inherited by another class for a specific protocol implemention.

The `Tag` class contains an `ITM` instance and a `TagReceive` instance. The class also contains one integer array and one double array, which holds the channel and energy responses respectively. The protected broadcast method in the `Tag` class is used by the inheriting classes to deliver a noise-processed, amplitude-decayed, and delayed signal to every reader.

5.3.7 TagSend class

The `TagSend` class begins its life cycle when the `Signal` instance it contains is simulated to reach a `Reader` instance, and terminates after the `Signal` instance is sent to

the buffer of the Reader instance.

5.3.8 TagReceive class

The TagReceive class simulates the receiver of a Tag instance. The abstract class is inherited by another class for a specific protocol implementation.

The TagReceive class contains the buffer that collects the reader commands sent from Reader instances, one integer array and one double array, which holds the channel and energy responses respectively. The TagReceive class also provides public methods to return the channel and energy responses, and a public method to allow ReaderSend instances to add reader commands to the buffer. The class also provides a protected method that purges expired Signal instances in the buffer (a Signal instance is considered expired when the time received plus the length of the signal is less than the current time).

5.3.9 RFIDModel class

The RFIDModel class extends the Model class, which is part of the DESMO-J simulation framework package. The Model class is used to initialize the actors in a DESMO-J simulation, and any simulator built on top of DESMO-J is required to extend the Model class and override the abstract initialization method. The abstract class is inherited by another class for a specific protocol implementation.

Because there is only one instance of the Model class for every simulation, and the object is referenced in private in all DESMO-J SimProcess instances, the RFIDModel class includes several global variables that are shared by all instances in the simulation:

A Random instance - every instance during the simulation run that wishes to generate a random variable uses the Random instance provided by RFIDModel. The Random instance is instantiated with seed provided by the user. This ensures that an RFID simulation with the same seed, same reader and tag positions, same protocol, and same protocol parameters will run deterministically.

A SimLog instance - every SimProcess actor that wishes to log a message uses the

SimLog instance provided by RFIDModel. The SimLog is handed over to the graphical user interface classes at the end of a simulation run.

Array of Reader and array of Tag - All Reader instances and Tag instances participating in the simulation is referenced in these two arrays.

5.3.10 SimLog class

The SimLog class provides methods to log either a plain text message, a plain text message with a Signal instance, or a plain text message with a RiseAndFall instance. The command sequence, a string that is manually enforced by the implementation to track the order of method invocation, is also entered into SimLog (The command sequence shows where the logMessage appeared from, and is mostly used for debugging).

When a Reader instance or a Tag instance calls one of the logging methods of the SimLog instance, the method also adds the log message and command sequence into respective Vectors, unless the strings already exists. These two Vectors are later used in the graphical user interface to filter out messages selectively, in order to improve readability. However, because typical log messages include a unique number at the end of the message, and command sequences usually carries a long chain of method calls, the methods strips out anything after the first ":" from the log messages, and command sequences are tokenized by the string character ":" and added individually to the Vector.

5.3.11 SigLib class

The SigLib class provides signal modeling methods that adds noise to a Signal instance, decreases the energy level of a Signal instance, merges several Signal instances together, computes the delay it takes for a Signal instance to travel, and performs edge detection and flat signal detection. The abstract class is inherited by another class for a specific protocol implementation.

The class also stores variable parameters that affect the behavior of the signal

modeling methods.

5.3.12 RiseAndFall class

An instance of the `RiseAndFall` class is returned by the edge detection method in a `SigLib` instance. The `RiseAndFall` class provides public methods to return number of amplitude shifts there are in the corresponding `Signal` instance, as well as the absolute times of falling and rising edges.

5.4 Signal Modeling

This section describes the implementation details of the signal modeling algorithms used in *RFIDSim*.

5.4.1 Distance and Noise function

The distance portion of the function computes the distance between the `Tag` instance and the `Reader` instance, and calculates the amount of amplitude decay the `Signal` instance should encounter. The amplitude decay factor is then uniformly applied to the internal double array of the `Signal` instance.

The noise portion of the function then proceeds to add noise to the `Signal` instance before the instance is delivered to its intended recipient. The variable `noiseRange` is used to determine the amount of noise added, which is the maximum percentage that the amplitude will fluctuate. For example, a noise range of 0.2 will cause the new amplitude to randomly vary between 80% and 120% of the original amplitude.

5.4.2 Delay function

The delay function returns a propagation delay that is calculated proportionally to the distance between a `Tag` instance and a `Reader` instance.

5.4.3 Edge Detection Algorithm

The edge detection algorithm runs in a while loop as it traverses the internal energy array. Each iteration of while loop runs one of the two smaller mutually exclusive while loops, one when the method is looking for a fall and one when the method is looking for a rise.

In order to handle jitters in a signal, the method stores a running average of the average energy level present in a signal as it traverses through the internal array. Whenever the internal level drops or rises above a certain percentage set forth by the `SigLib`, the edge detection flags the array index relative to the internal array, and flips the Boolean necessary to quit the small internal while loop, so it exits the larger while loop iteration and begins looking for the next amplitude shift.

The edge detection algorithm terminates when the entire array is traversed, and returns a `RiseAndFall` instance.

Protocol implementations such as Class 0 sends out continuous signals, and the falling edge that signifies the beginning of a data symbol is actually dependent on the energy level from the previous signal. Therefore the edge detection takes in a parameter that specifies the previous power level detected right before the current signal begins.

5.4.4 Flat Signal Detection Algorithm

The flat signal detection method is used primarily in Class 0, but the method is integrated in the generic signal library.

The flat signal detection method returns the length of time before there is a amplitude shift. This is useful for identify the master reset signal of the Class 0 protocol, where the tag would automatically reset if a high flat signal of more than 400 microseconds is detected.

5.5 Class 0

This section describes the functionality presented by the Java classes implemented to simulate the Class 0 discovery process.

5.5.1 ClassZeroITM class

The `ClassZeroITM` class extends the `ITM` class. The class defines the EPC types available for Class 0, and also provides a method that verifies the EPC, taking into account the `epcType` defined by the instance.

5.5.2 ClassZeroReader class

The `ClassZeroReader` class extends the `Reader` class, and implements the Class 0 discovery algorithm in the `lifeCycle` method.

The class provides an abstracted method send that performs message logging method calls and the broadcasting of the modulated reader command. As a result, the five reader commands can be sent from invoking `send(`i`)`, where i is 0 to 4 inclusive.

The class also provides an abstracted method `analyzeBackScatter` that performs message logging method calls, and the identification of backscatter responses in the buffer. The method returns one of the four possible combinations of responses - only tone zero, only tone one, both tones, and no responses.

5.5.3 ClassZeroTag class

The `ClassZeroTag` class extends the `Tag` class, and implements the Class 0 tag state machine in the `lifeCycle` method.

The class provides a public method `getState`, which returns the state the instance currently belongs in. The class also provides two abstracted methods `backScatterOne` and `backscatterZero`, which performs the broadcasting of the backscatter response.

5.5.4 ClassZeroTagReceive class

The `ClassZeroTagReceive` class extends the `TagReceive` class, and implements a loop in the `lifeCycle` method that performs preemptive signal handling to identify the reader command. The `ClassZeroTagReceive` instance wakes up the `ClassZeroTag` instance even if the received signal is invalid.

The class provides a public method `getResponse`, which returns the demodulated identify of the reader command, as well as a public method `getBackScatterLength`, which replies the proper length of the back scatter, dependent on the received data symbol.

5.5.5 ClassZeroModel class

The `ClassZeroModel` class extends the `RFIDModel` class, and implements the abstract methods that instantiate and initialize the `ClassZeroTag` and `ClassZeroReader` instances.

5.5.6 ClassZeroSigLib class

The `ClassZeroSigLib` class extends the `SigLib` class, and provides modulation and demodulation methods for Class 0 reader commands and tag backscatter responses. The class also stores variable parameters that affect the modulation and demodulate of the Class 0 signals.

5.6 Class 1

This section describes the functionality presented by the Java classes implemented to simulate the Class 1 discovery process.

5.6.1 ClassOneITM class

The `ClassOneITM` class extends the `ITM` class. The class defines the EPC types available for Class 1, and also provides a method that verifies the EPC, taking into

account the epcType defined by the instance.

5.6.2 ClassOneReader class

The ClassOneReader class extends the Reader class, and implements the Class 1 discovery algorithm in the lifeCycle method.

The class provides a collection of abstracted methods send*, which only takes the pointer and value as parameters. The send* methods calls the abstracted method send, which performs the broadcasting of the signal.

The class also provides abstracted methods lookAtBin and lookAtBackScatter, which returns a binary string from analyzing PingID backscatters and ScrollID backscatters respectively, "nothing" if there are no responses, or "invalid" if the backscatter response cannot be properly demodulated.

5.6.3 ClassOneTag class

The ClassOneTag class extends the Tag class, and implements the Class 1 tag state machine in the lifeCycle method. The class provides an abstracted method called backScatterReply, which performs the broadcasting of the backscatter response.

5.6.4 ClassOneTagReceive class

The ClassOneTagReceive class extends the TagReceive class, and implements a loop in the lifeCycle method that performs preemptive signal handling to identify the reader command. The ClassOneTagReceive instance does not wake up the ClassOneTag instance even if the received signal is invalid.

The class provides a public method getResponse, which returns the demodulated binary string of the reader command, as well as a public method getT0, which is the signal length of one modulated bit "0".

5.6.5 ClassOneModel class

The `ClassOneModel` class extends the `RFIDModel` class, and implements the abstract methods that instantiate and initialize `ClassOneTag` and `ClassOneReader` instances.

5.6.6 ClassOneSigLib class

The `ClassOneSigLib` class extends the `SigLib` class, and provides modulation and demodulation methods for Class 1 reader commands and tag backscatter responses. The class also stores variable parameters that affect the modulation and demodulate of the Class 1 signals.

5.6.7 ClassOneLogLib class

The `ClassOneLogLib` class provides genetic bit manipulation and number / bit conversion methods that facilitate the construction of complete commands. The class also provides abstracted methods that construct the binary string commands for Class 1 readers and tags. The class also stores 8-bit binary string constants that defines the identity of different reader commands.

5.7 Other objects used by *RFIDSim*

This section describes the functionality presented by the Java classes not part of the simulation, but contribute to the setup of each simulation.

5.7.1 Experiment class

The `Experiment` class is part of the DESMO-J simulation framework package. The class must be instantiated in order to set up a DESMO-J simulation. The class terminates the simulation when the method check of the `StopCondition` instance returns true.

5.7.2 Environment class

The `Environment` class stores the dimensions of the simulation environment, as well as the coordinates of the readers and tags in the environment. The resulting instance is one of the inputs for the constructor method of the `RFIDModel` class, and used to instantiate readers and tags with the specified coordinates.

5.7.3 StopCondition class

The `StopCondition` class extends the `Condition` class, which is part of the DESMO-J simulation framework package. The class implements the method `check`, which returns true only if all readers in the environment have terminated their respective discovery phase.

5.8 User Interface

This section describes the functionality presented by the Java classes that construct the user interface of *RFIDSim*.

5.8.1 EnvironmentPanel class

The `EnvironmentPanel` class is used to construct the environment settings panel of the `RFIDWindow` instance. The class displays four button / commands - New, Load, Save, and Modify. An `Environment` instance is created after a successful construction of a new environment, or a successful parse of a environment settings file.

5.8.2 ClassZeroPanel

The `ClassZeroPanel` class is used to display text fields for parameters that can be modified for a Class 0 simulation.

5.8.3 ClassOnePanel

The `ClassOnePanel` class is used to display text fields for parameters that can be modified for a Class 1 simulation.

5.8.4 RFIDWindow class

The `RFIDWindow` class is used to construct the main window of *RFIDSim*, which contains a EnvironmentPanel instance, as well as instances of the `ClassZeroPanel` and `ClassOnePanel`.

5.8.5 ResultsWindow class

The `ResultsWindow` class is instantiated after the termination of the simulation experiment, and constructs a window that displays all the logged messages in a table format.

5.8.6 MapDialog class

The `MapDialog` class is used to create the window that allows users to add new readers and tags with a mouse. The class also displays a map that shows the location of all readers and tags in the environment so far.

5.8.7 SimLogTableModel class

The `SimLogTableModel` class extends the `AbstractTableModel` class, and wraps a `SimLog` instance into a format that can be used to create a `JTable` instance.

5.8.8 SignalPanel class

The `SignalPanel` class takes a `Signal` instance for instantiation, and creates a canvas that displays the amplitude levels of a signal graphically.

5.8.9 RAndFPanel class

The `RAndFPanel` class takes a `RiseAndFall` instance for instantiation, and creates a canvas that displays the rising and falling edges of a signal graphically.

Chapter 6

Usage

6.1 Building with source code

RFIDSim is released under the GNU General Public License - the source code of the simulator is available when requested, and any user is free to modify *RFIDSim*, provided that the user releases any changes and improvements made.

To compile *RFIDSim* from source, the user should have the following applications installed on the build machine:

Java 2 Platform - The Java 2 Platform can be obtained for free from Sun Microsystem's Java website at:

http://java.sun.com

DESMO-J simulation framework - The DESMO-J simulation framework can be obtained for free from DESMO-J's homepage at:

http://asi-www.informatik.uni-hamburg.de/themen/ ..

.. sim/forschung/Simulation/Desmo-J/

The DESMO-J simulation framework should either be placed in the same directory as *RFIDSim* source files, or the path of the DESMO-J framework .jar file should be part of the Java classpath.

To ensure a clean compile of *RFIDSim*, the .class files for *RFIDSim* should be deleted from the directory first.

The user should execute the following command to properly compile *RFIDSim*:

```
javac RFIDSim.java
```

6.2 Execution requirements

If *RFIDSim* has compiled successfully, the user can type the following command to execute *RFIDSim*:

```
java RFIDSim
```

Because of the large amount of memory resources *RFIDSim* uses, the user might encounter an "out of memory error" during the execution of the simulation when there are a large number of tags or readers in the simulated environment. The user can type the following to allocate more memory for the java virtual machine:

```
java -Xms128m -Xmx1024m RFIDSim
```

The `-Xms` parameter specifies the initial heap size of the Java virtual machine, and the texttt-Xmx parameter specifies the maximum heap size of the Java virtual machine. In the above example, the Java virtual machine would allocate 128 megabytes for its initial heap size, and the heap size will grow to a maximum of 1 gigabytes.

6.3 Using *RFIDSim*

This section explains each component of *RFIDSim*'s user interface, as well as any features offered by *RFIDSim* that would facilitate the analysis of the simulation log.

Figure 6-1 is the main window of the *RFIDSim*, and what the user first sees when *RFIDSim* executes. The dropbox displaying "Class Zero Simulation" can be clicked, and the user can choose to run the Class One Simulation instead (figure 6-2). However, in order to run any simulation, the environment settings must be configured correctly. (The environment settings are independent of simulation profile. The environment settings are preserved even if the user switches to a different simulation profile.)

The top panel provides several options specific to configuring the environment settings. New environment settings can be created by clicking "New..", or it can be loaded from a previously saved environment settings file by clicking "Load..".

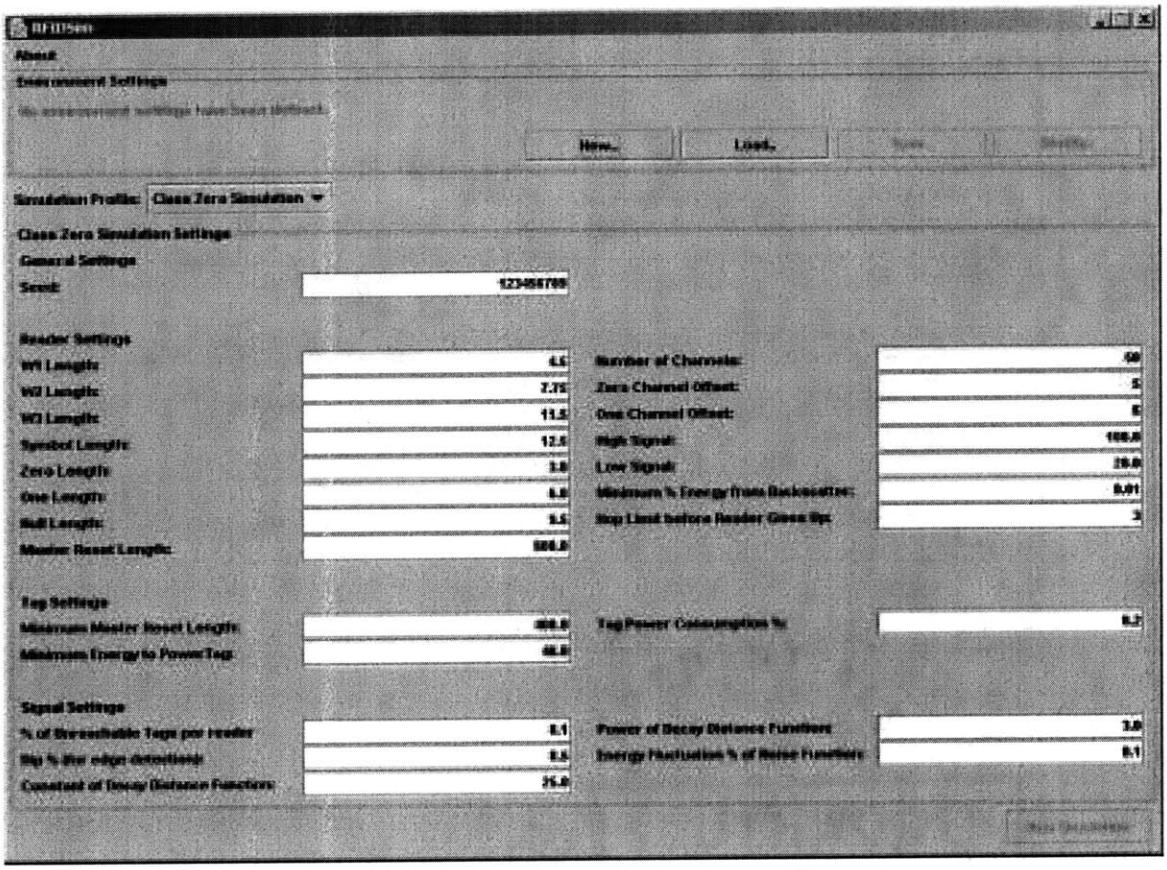

Figure 6-1: The main window of *RFIDSim* (screenshot)

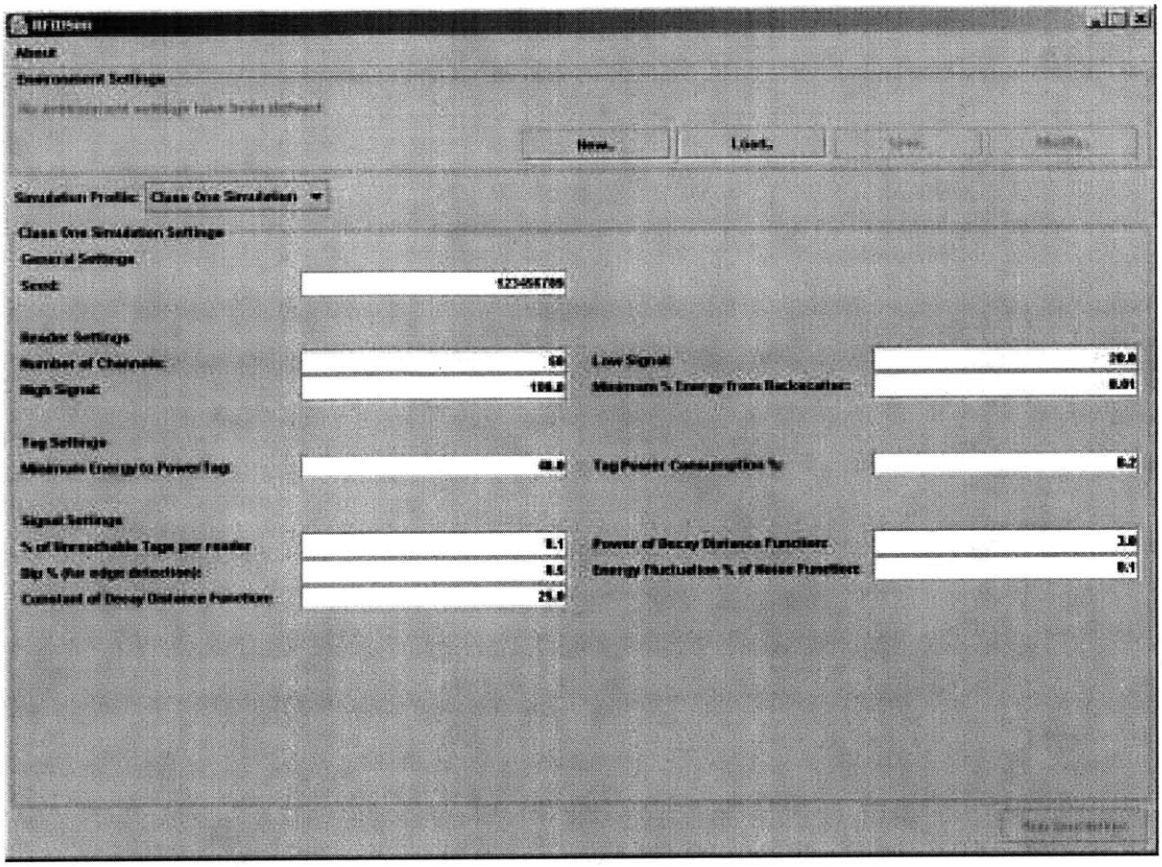

Figure 6-2: Choosing a different profile in *RFIDSim* (screenshot)

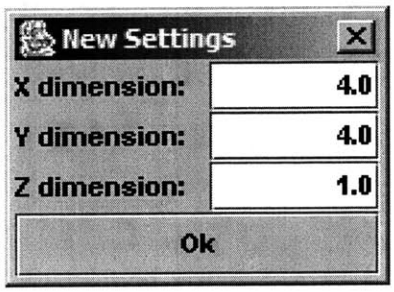

Figure 6-3: Dimensions dialog (screenshot)

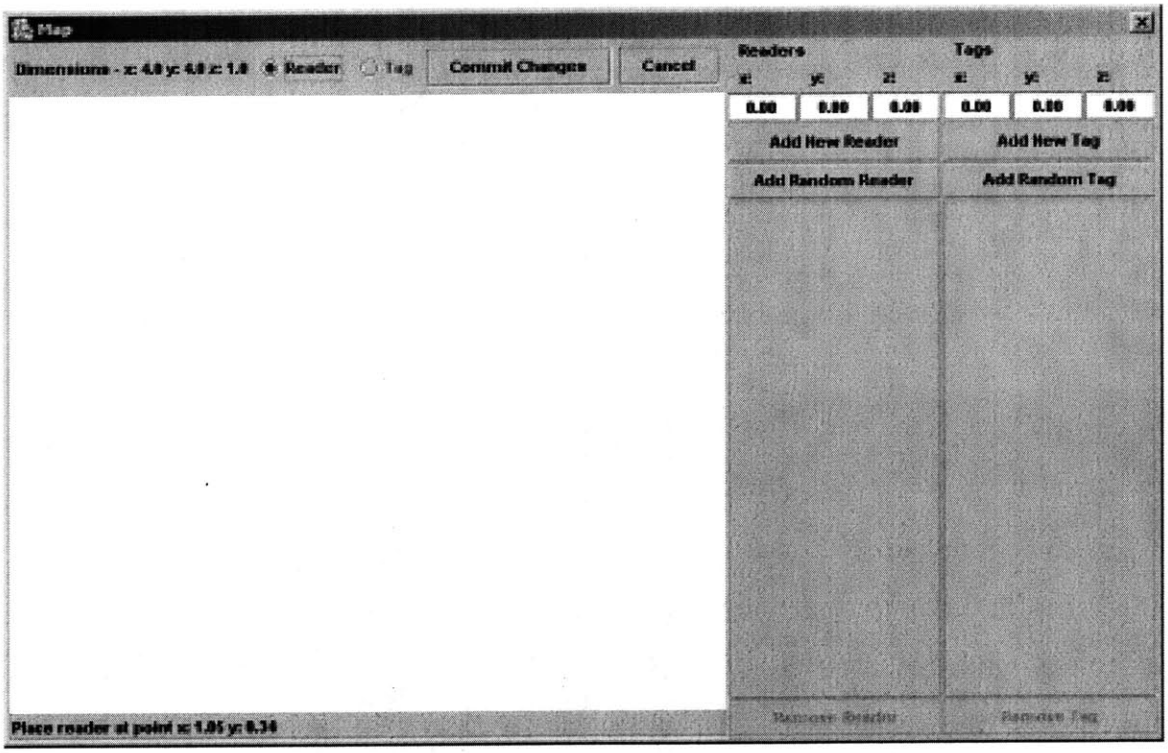

Figure 6-4: Map dialog with new environment (screenshot)

The dimensions dialog appears after the user clicks "New.." (shown in figure 6-3. The dialog prompts for the specific dimensions of the simulation environment.

After the user defines the dimensions and clicks "Ok", the Map window (shown in Figure 6-4 is opened with an empty environment. The user can choose to add readers and tags in the environment by clicking on the environment itself, or the user can choose to enter the coordinates manually in the upper right text fields, and clicking the "Add New Reader" or "Add New Tag" button. Finally the user can also click "Add Random Reader" or "Add Random Tag", and *RFIDSim* will add the respective reader or tag with random coordinates.

If the user adds a reader or tag by clicking on the environment, a dialog (figure 6-5) pops up to prompt the user to enter the coordinate for the Z dimension.

Readers and tags can also be removed from the environment. To remove a tag or reader from the environment, the user should choose the row of the tag or reader to be removed, and click "Remove Reader" or "Remove Tag". In order to allow the user

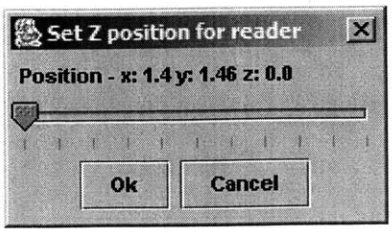

Figure 6-5: Dialog after clicking on Environment (screenshot)

to easily identify the reader or tag the row refers to, the reader or tag is highlighted in the environment when a row is clicked.

The environment settings for a simulation is defined when the user click "Commit Changes", and the user returns to the main window. If the environment is valid (the only requirement being that there is at least one reader and one tag), the user is then allowed to begin the simulation or to save the environment settings to a file. Clicking "Modify.." returns the user to the Map window, where the user can modify the environment by adding or removing tags.

The bottom pane lists specific settings for the protocol. The settings are categorized into General Settings (which is simply the seed used to drive the simulation), Reader Settings, Tag Settings and Signal Settings. Once the settings are approved by the user, the user can click "Run Simulation" and wait for the results window to open.

Depending on the complexity of the environment and the processing speed of the machine, it might take *RFIDSim* a few minutes to complete the simulation and open the Results in a new window. Clicking the "Run Simulation" repeatedly will only initiate concurrent simulations that will deterministically present the same result, as well as slowing down the simulation even more.

The results window (figure 6-6) is composed of a table that displays the simulation log, as well as several smaller tables that allow the user to reduce the verbosity of the simulation log by filtering out message types, or all messages from a selected reader or tag. The user should first check and uncheck what log entries would like to be seen or hidden, and can either refresh the table by clicking the button "Update current

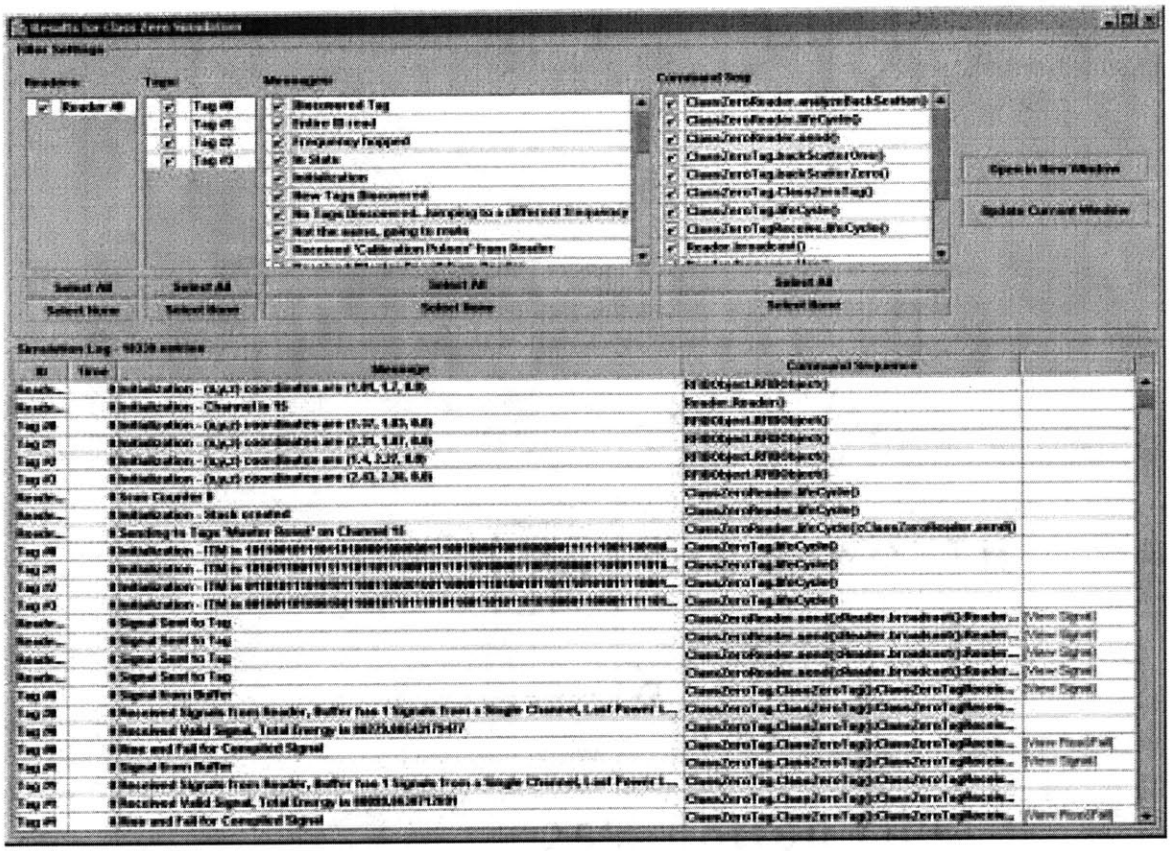

Figure 6-6: Results from simulation (screenshot)

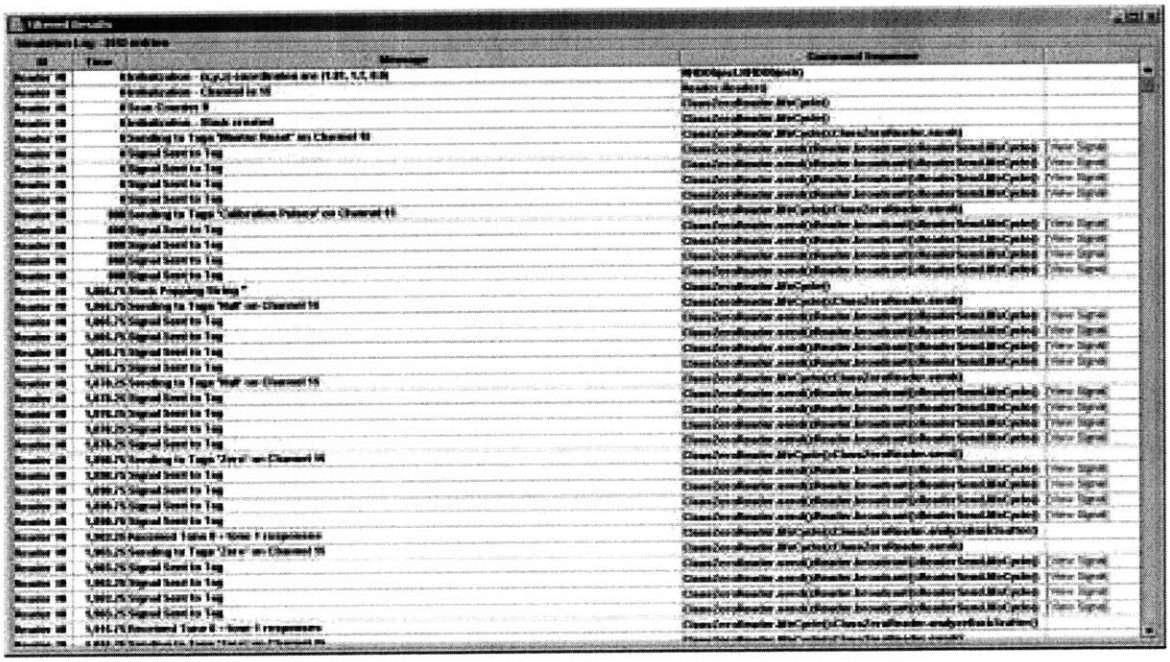

Figure 6-7: Filtered results in a new window (screenshot)

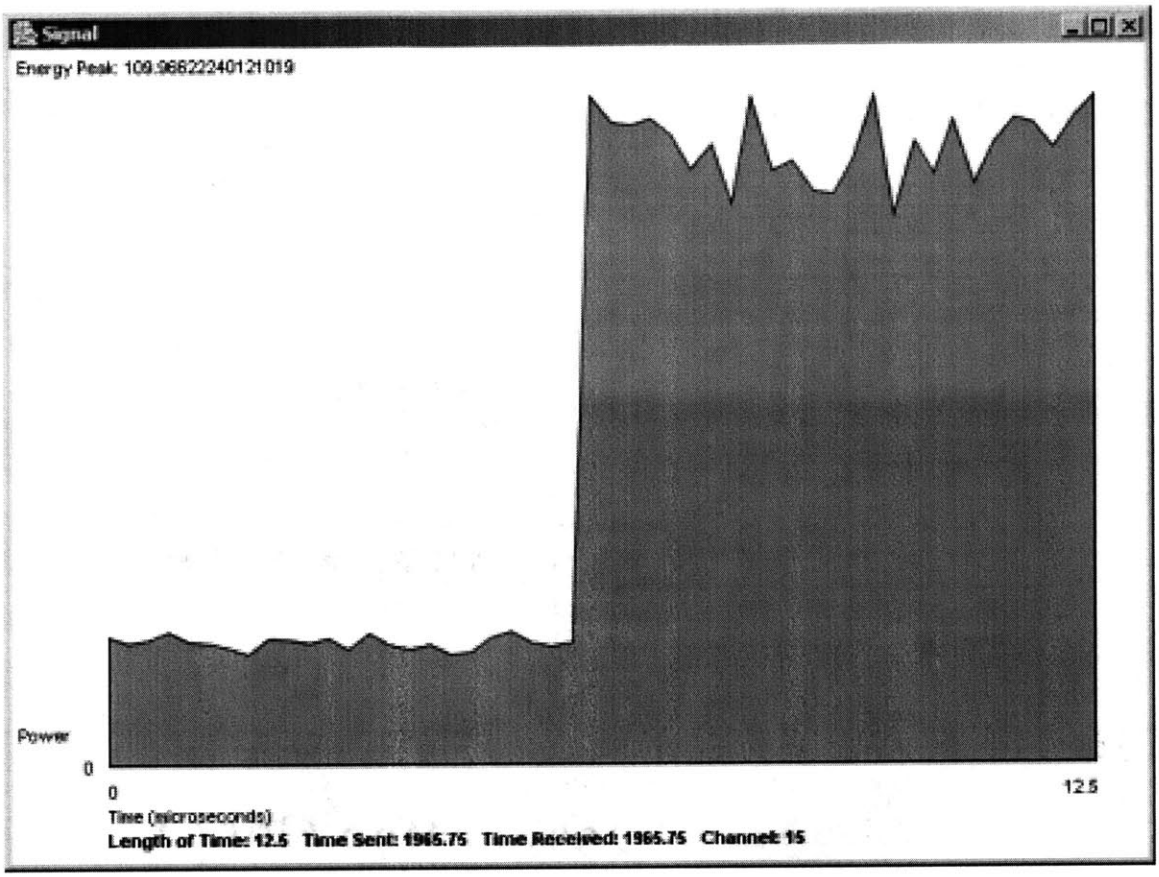

Figure 6-8: Graphical representation of a Signal instance (screenshot)

window", or open up the filtered results in a new window (figure 6-7).

The columns for the table showing the simulation log entries are ID (indicating which reader or tag logged the message), Time (which indicates the time during the simulation when the message was logged), Message (which displays the actual message logged), Command Sequence (which displays the instance and method the message was logged from), and the signal / rise and fall column (where the presence of text indicates that a graphical representation of a Signal instance or RiseAndFall instance is viewable from clicking that cell)

Clicking on a "[View Signal]" cell would open a new window that plots the Signal instance in its signal form (figure 6-8). Clicking on a "[View Rise&Fall]" would plot the RiseAndFall instance using the rising and falling times (figure 6-9). The signal window shows a graphical representation of the amplitude levels of the logged Signal

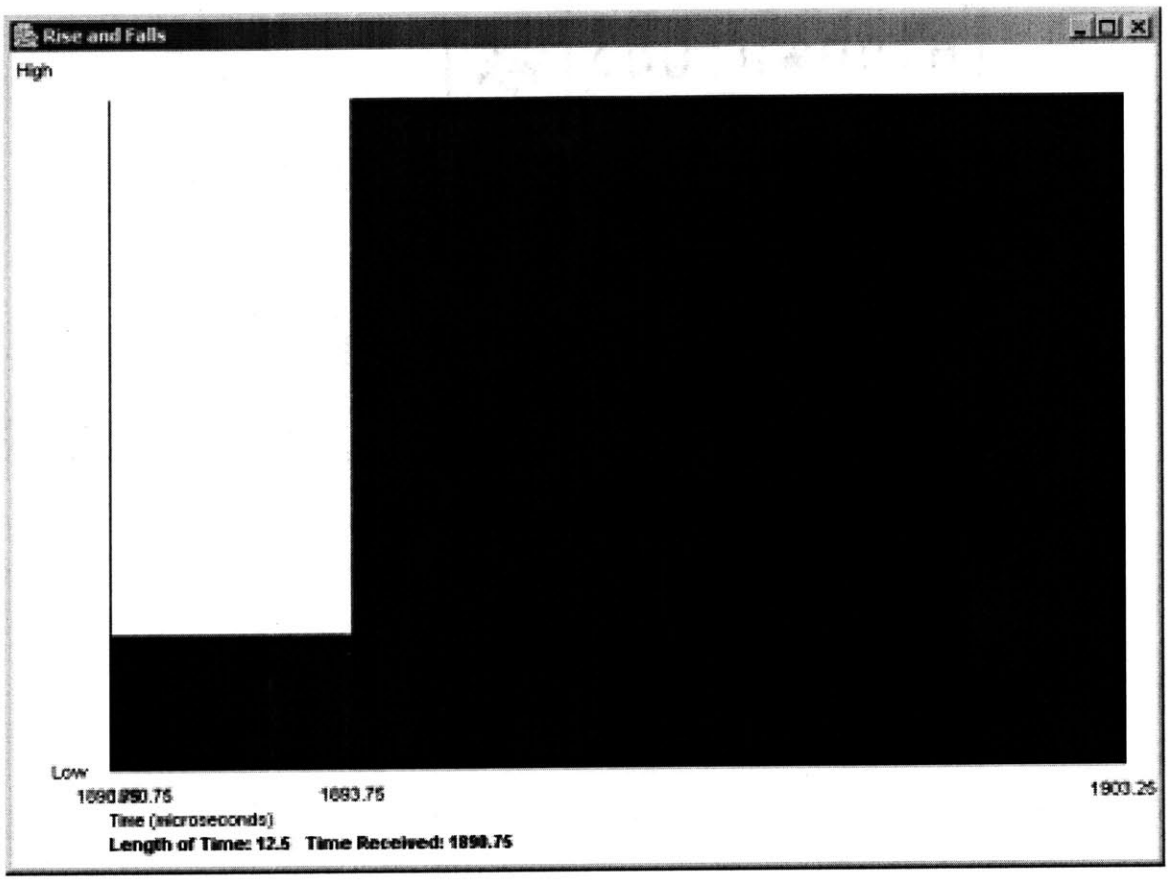

Figure 6-9: Graphical representation of a RiseAndFall instance (screenshot)

instance. Both types of windows can be resized dynamically to see more details.

Chapter 7

Conclusion

7.1 Summary

RFIDSim is currently implemented to provide a high level of abstraction between logic and signal modeling components, which allow developers to focus on improving the logical algorithms of the RFID protocols without having to worry about the details of the signal modeling components, or focus on improving the signal modeling components without breaking the communication logic of the supported protocols. The multiple levels of abstracted method calls in the logical components of the protocol improves code clarity in the reader's discovery algorithm and tag's state machine.

RFIDSim is also designed to be a solid and extensible framework that allows future protocols to be easily implemented. The unique asymmetric nature of communication between RFID readers and tags are modeled in the core components of the simulator, and common RFID functions, such as edge detection, are shared and inherited by all protocols. The collection of common RFID functions is expected to grow as different protocols are implemented in the simulator, which would further minimize the amount of work needed to add support for other protocols. The large amount of shared functionality between different protocols also allows protocol comparisons to be more consistent.

Finally, the analysis and debugging tools implemented in the simulator allows the user to analyze the communications between readers and tags on any level of detail,

from the logical communication level, down to the waveform of a modulated signal.

7.2 Future work

RFIDSim can be improved and extended in several areas - simulation performance, signal modeling assumptions, support for powered tags, and coordination between readers during the discovery process.

7.2.1 Simulator performance

The data structures of some core components, such as the Signal class, can be further optimized, as the current design is less than ideal. Inefficient data structures can be expensive in terms of processor cycles, as there might be a lot of unnecessary and costly casting between data types. The inefficiency can also be expensive in terms of memory management, as bloated data structures will limit the number of readers and tags that can be simulated in a given environment.

7.2.2 Refinement in the signal modeling assumptions

The current noise, distance and energy models are currently quite high level and simple, and the accuracy of the simulation results can be increased if they are replaced with more sophisticated models.

7.2.3 Support for powered tags

Currently the simulator only supports powerless passive tags, and while these low-cost tags will be the preferred choice for customers wishing to deploy RFID solutions on a large-scale, there should be minimal work involved in getting the RFID simulator to support powered tags as well.

7.2.4 Coordination between readers

In most applications of RFID scenarios, the size of the environment will require using more than one single reader. In the current implementation of the simulator, there is no coordination, or even communication allowed between readers in the same environment. Implementing support for communication between readers would allow for more interesting and efficient discovery algorithms.

Appendix A

Class 0

A.1 ITM

The ITM of a Class 0 tag contains a 64-bit or 96-bit EPC (Electronic Product Code) that identifies a physical object, and a 16-bit CRC checksum of the EPC at the end. The CRC is used for verification purposes when a reader discovers a tag, in order to conclude with high probability that the EPC discovered is without errors.

For the discovery algorithm, the class 0 protocol has defined three modes of identification codes to be used for discovery. Our discovery process focuses on the ID2 format.

A.1.1 ID0

ID0 uses a complete random identification code for the discovery process. Instead of employing CRC methods for verification, the format embeds an odd parity bit after every 4 bits. Because the identification code is completely random for every discovery process, the ID0 interrogated from the tag will be different for every discovery process, and the actual EPC must be read by the reader at the end of a successful singulation. The identification code structure of ID0 is shown in figure A-1.

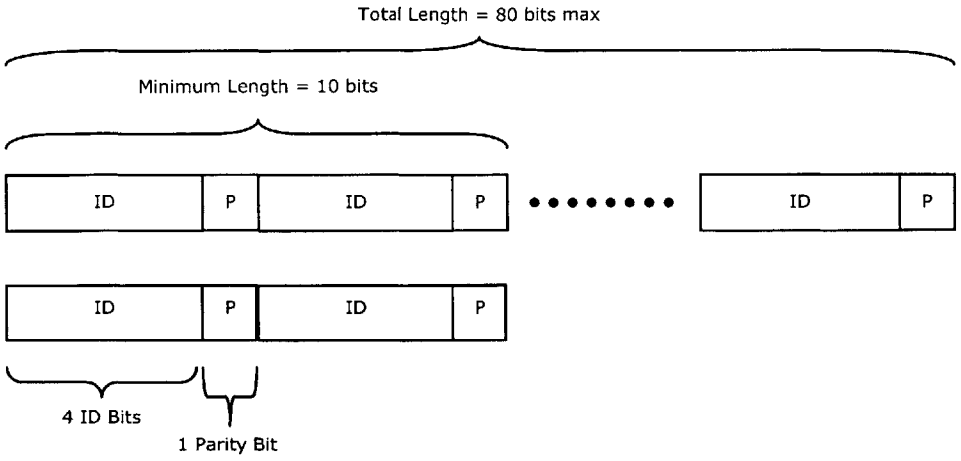

Figure A-1: Identification code of the ID0 format for the Class 0 protocol.

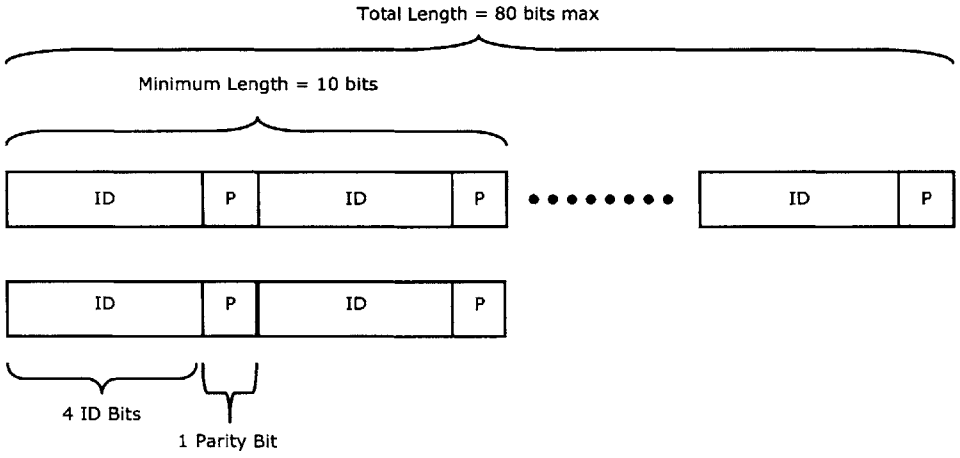

Figure A-2: Identification code of the ID1 format for the Class 0 protocol.

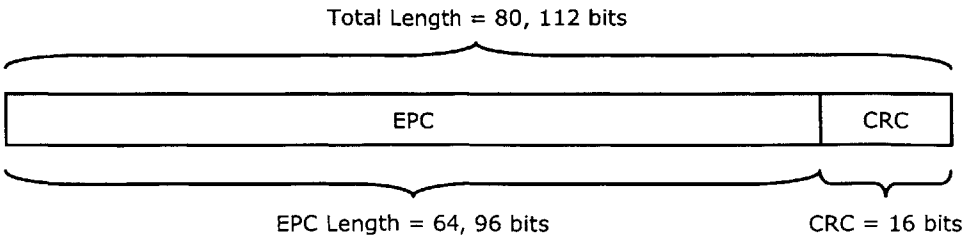

Figure A-3: Identification code of the ID2 format for the Class 0 protocol.

A.1.2 ID1

ID1 uses a pseudo-random identification code for the discovery process. Instead of employing CRC methods for verification, the format embeds an odd parity bit after every 4 bits. Because the identification code is not completely random, the EPC can be discovered by looking up a one-to-one mapping of ID1-to-EPC database. The identification code structure of ID1 is shown in figure A-2.

A.1.3 ID2

ID2 uses the EPC code and CRC as the identification code for the discovery process. At the end of a successful singulation, the CRC is verified and compared in order to conclude with high probability that the EPC discovered is without errors. The identification code structure of ID2 is show in figure A-3.

A.2 Reader Commands

Class 0 reader commands are modulated using ASK and sent continuously in one stream. These are the following commands a Class 0 reader can issue:

A.2.1 Master reset

A master reset signal is characterized by a 800 microsecond carrier-wave signal.

Figure A-4 shows the signal form of a master reset command.

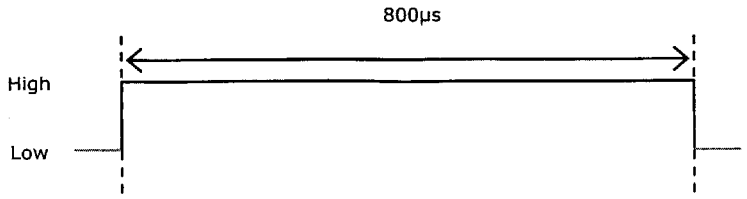

Figure A-4: Signal of a master reset command for the Class 0 protocol.

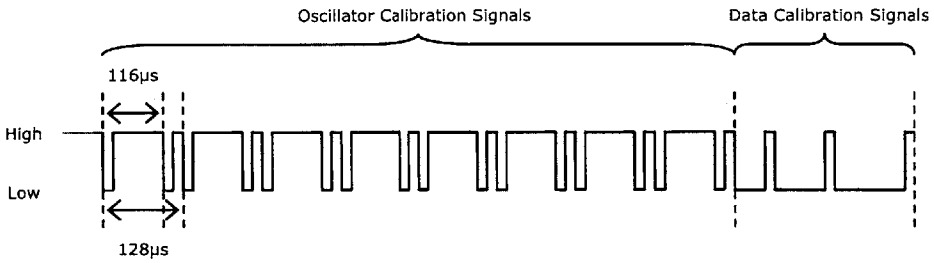

Figure A-5: Signal of a master reset command for the Class 0 protocol.

When a tag receives a master reset signal, the tag resets all internal states, and initializes itself at the beginning of the discovery process.

A.2.2 Calibration

A calibration signal immediately follows a master reset signal. It's composed of the oscillator calibration signals, followed by the data symbol calibration signals.

The oscillator calibration signals are used to synchronize the clock speed of the tags. The data symbol calibration signals are used to define the timings and lengths of the data symbols.

Figure A-5 shows the signal form of a calibration command.

When a tag receives a calibration signal, the tag synchronizes itself to the clock given in the oscillator portion, and remembers the timings defined by the data symbol calibration signals.

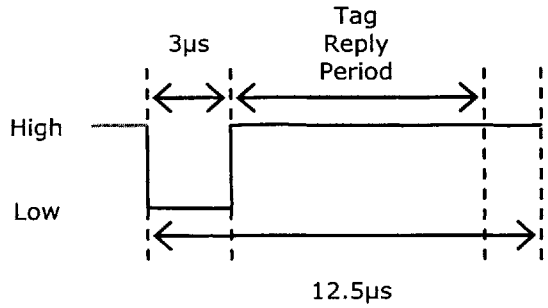

Figure A-6: Signal of a zero data symbol for the Class 0 protocol.

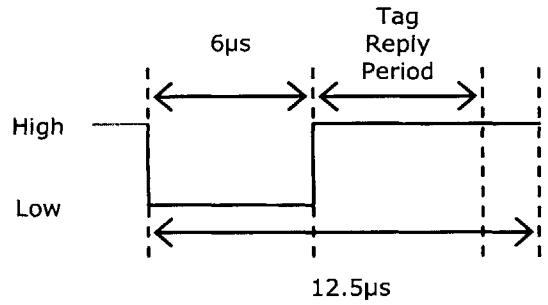

Figure A-7: Signal of a one data symbol for the Class 0 protocol.

A.2.3 Zero / One / Null

Zero / one / null data symbols are sent to transition tags from state to state. The falling edge from a pervious signal determines the beginning of the symbol, and length of time the rising edge appears determines the identify of the data symbol.

Figure A-6 shows the signal form of a zero data symbol, figure A-7 shows the signal form of a one data symbol, and figure A-8 shows the signal form of a null data

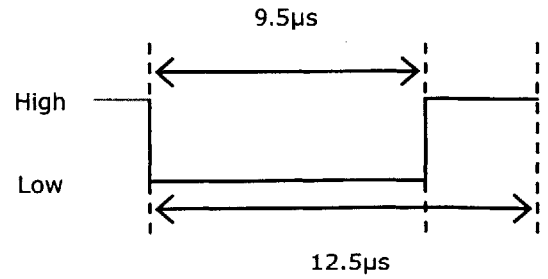

Figure A-8: Signal of a null data symbol for the Class 0 protocol.

symbol.

When a tag receives a data symbol, the tag identifies the data symbol with the calibration signal sent during initialization. After the data symbol has been identified correctly, the tag performs a state transition depending on the data symbol.

A.3 Tag States

Figure A-9 shows the state transition diagram of a Class 0 tag.

A.3.1 Dormant State

A Class 0 tag in this state is inactive, and waiting for a master reset signal from a reader.

A.3.2 Calibration State

A Class 0 tag in this state has just received a master reset signal, and waiting for a calibration signal from the reader.

A.3.3 Global Command Start State

A Class 0 tag in this state has just received a calibration signal, and waiting for various data symbols to transition to other states.

A.3.4 Global Command State

A Class 0 tag in this state is awaiting various global commands that can be issued by the reader. This state is not used in *RFIDSim*'s implementation of the discovery process.

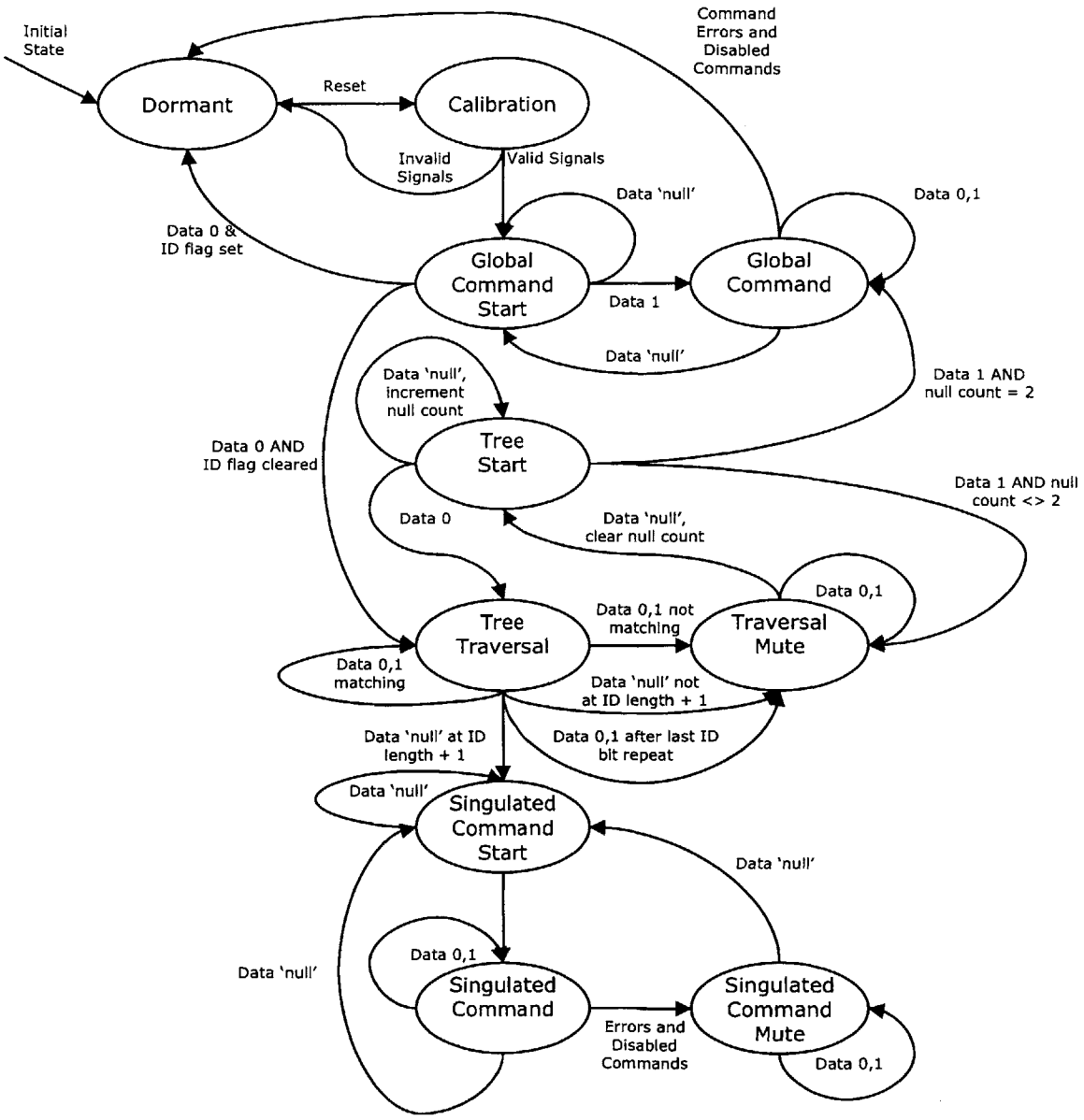

Figure A-9: State Machine defined for Class 0 tags

A.3.5 Tree Start State

A Class 0 tag in this state is awaiting to begin the discovery process, and waiting for a zero data symbol to begin the tree-walking algorithm.

A.3.6 Tree Traversal State

A Class 0 tag in this state is operating in the tree-walking algorithm, and waiting for the next data symbol.

A.3.7 Traversal Mute State

A Class 0 tag in this state has received a data symbol that does not match its own identification code, and has temporarily muted itself until it receives a null data symbol to restart the tree-walking algorithm.

A.3.8 Singulated Command Start State

A Class 0 tag in this state has successfully been singulated, and only one tag should be in this state at any given time. The tag is waiting for a data symbol to return to the dormant state.

A.3.9 Singulated Command State

A Class 0 tag in this state has successfully been singulated, and awaiting various commands that can be issued by the reader. This state is not used in *RFIDSim*'s implementation of the discovery process.

A.3.10 Singulated Command Mute State

A Class 0 tag in this state has successfully been singulated, and entered a temporary mute condition because of an error in the reader's command, or because of a disabled command issued by the reader. The tag is waiting for a null data symbol to

return to the singulated command start state. This state is not used in *RFIDSim*'s implementation of the discovery process.

A.4 Tag Backscatter Responses

The following are backscatter responses a Class 0 tag can issue. The varying frequencies at which the backscatter response is modulated at puts the backscatter response in a different channel.

A.4.1 Zero

A zero backscatter response is defined by a modulated and alternativing backscatter at 2.2 Mhz. The backscatter begins on the rising edge of the receiving data symbol, and ends at a point defined by the data symbol calibration symbol.

A.4.2 One

A one backscatter response is defined by a modulated and alternativing backscatter at 3.3 Mhz. The backscatter begins on the rising edge of the receiving data symbol, and ends at a point defined by the data symbol calibration symbol.

A.5 Discovery Process

The Class 0 discovery algorithm implemented in *RFIDSim* uses a variation of the tree-walking algorithm that walks down one level at a time.

A.5.1 Algorithmic Overview

1. After initialization, the reader broadcasts a zero symbol to announce the beginning of the tree-walk. All tags receiving the zero symbol would send out their MSB bit in the identification code.

2. If there is no backscatter response from both channels that the reader listens to, the reader realizes that either there are no tags present, or any present tags have fallen out of the synchronization. The reader will attempt to restart the discovery process.

3. If there is a backscatter response from only the channel indicating bit "0", the reader would broadcast a zero data symbol as the next signal. Tags that receive the zero data symbol would understand that it is walking down the "0" branch, and the tags with a matching bit in the identification code would backscatter their next bit. Tags that do not match would stay mute until the restart of the discovery process.

4. Similarly, if there is a backscatter response from only the channel indicating bit "1", the reader would broadcast a one data symbol as the next signal. Tags would interpret the one data symbol as the reader moving down the "1" branch, and the tags with a matching bit in the identification code would backscatter their next bit. Tags that do not match would stay mute until the restart of the discovery process.

5. If there are backscatter responses coming from both channels, the reader pushes the string for the "1" branch into a stack, as this indicates there are tags along the "1" branch as well, and the reader will revisit that path afterwards.

6. The process repeats by proceeding to Step 2, and completes a successful singulation when a tag has responded all the way to the lth level of the tree, where l is the length of the identification code.

7. The tag that has successfully backscattered all l bits of its identification code would automatically stay dormant for the rest of the discovery process, until the reader instructs the tag otherwise or when the tag's internal power depletes.

8. The reader, having identified and singulated one tag, pops the next string out of the stack and restarts the discovery process, prompting all the muted tags that fell out of the path to restart the tree-walking algorithm. If the stack is empty, depending on the rigorousness of the discovery algorithm, the reader either assumes all tags are discovered and terminates, or the reader repeats at Step 1, in order to discover any tags that it might have missed.

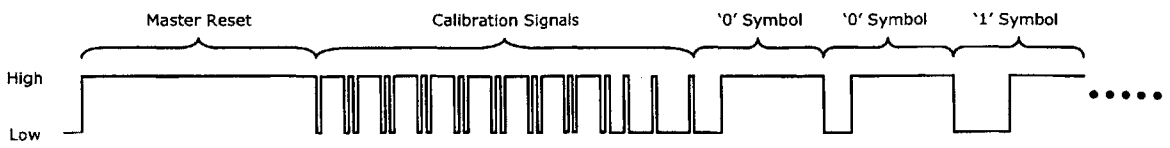

Figure A-10: Signal form of the commands sent by the reader at the beginning of the discovery process for the Class 0 protocol.

A.5.2 Signal Overview

The reader begins the discovery process by sending out a master reset and calibration signal, and the rest of the discovery process is a continuous string of data symbols. Figure A-10 shows the signals sent by the reader at the beginning of the discovery process.

The success of the algorithm is directly affected by tag's ability to receive every single signal for the duration of the discovery process. Any tag that has received an invalid state transition or an invalid data symbol would automatically exit into dormant state, and awaits the next master reset signal to begin the discovery process again.

A.5.3 Analysis

Because no information is modulated in the backscatter response, but rather in the decision to backscatter in either the zero or the one channel, the reader has effectively four bits of information after each data symbol sent in the tree-walking phase:

No response in zero channel + No response in one channel -> no tags, or tags are out of synchronization.

No response in zero channel + Response in one channel -> there are tags following the algorithm in the "1" branch.

Response in zero channel + No response in one channel -> there are tags following the algorithm in the "0" branch.

Response in zero channel + Response in one channel -> there are tags following the algorithm in both the "0" branch and the "1" branch.

The tag's large state transition machine, together with the tag's ability to store a pointer to the level of the binary tree the reader is walking on, requires tags to track a large amount of internal state during the discovery process.

The relatively small amount of information disclosed in the tags' response for each reader signal implies that a large number of reader signals would need to be sent in order to collect enough information from the tag. Therefore, in order to maintain acceptable performance levels, the length for each reader signal must be relatively short.

However, because each reader signal is relatively short, there is less information that can be encoded into each signal. Therefore, Class 0 tags are designed to track a large amount of internal state during the discovery process, including the position of the reader in the tree-walking algorithm.

Appendix B

Class 1

B.1 ITM

The ITM of a Class 1 tag contains a 64-bit or 96-bit EPC (Electronic Product Code) that identifies a physical object, and a 16-bit CRC checksum of the EPC at the beginning of the ITM.

There is also an 8-bit password at the end of the ITM that is used for reprogramming the tag. This feature is not used in *RFIDSim*'s implementation of the discovery process.

Figure B-1 shows the identification code structure of a Class 1 ITM.

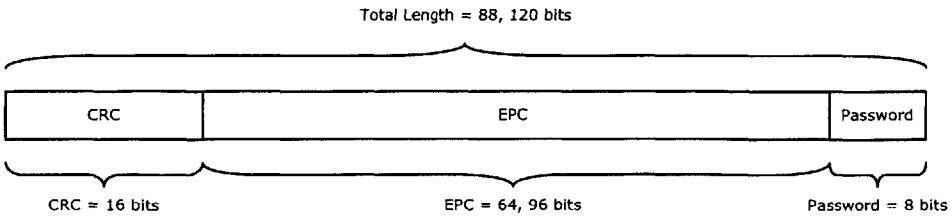

Figure B-1: Identification code structure of a Class 1 ITM.

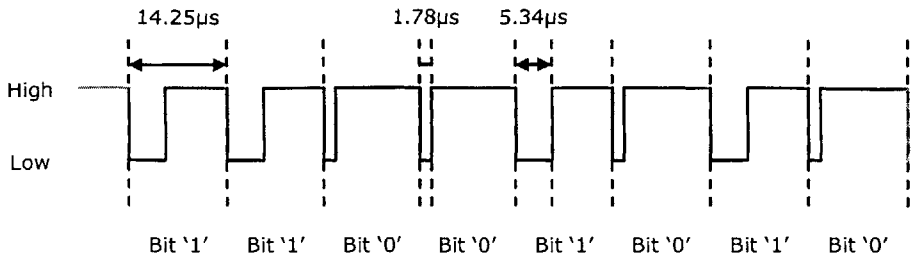

Figure B-2: Modulated signal form of the binary string "11001010".

B.2 Reader Command Format

Class 1 reader commands are packet-based, and each command follows a strict binary packet format. The binary data is modulated using ASK. The signals are not sent continuously in one single stream; as a result, each reader command contains a calibration period to synchronize the clock speed. Figure B-2 shows the modulated signal form of the binary string "11001010".

A reader command packet is defined with the following fields:

B.2.1 PREAMBL

The preamble is the prefix of a reader command. There is a period with no signals sent, followed by a period of carrier-wave signal.

B.2.2 CLKSYNC

The clock synchronization field is composed of 20 binary bits of "0". This period is used to allow the tags to synchronize their clocks to the command.

B.2.3 SOF

The Start of Frame field is composed of 1 binary bit of "1". This field indicates the beginning of the actual command.

B.2.4 CMD

The command field is composed of 8 binary bits. This field specifies the command being sent.

B.2.5 P1

The P1 field is composed of 1 binary bit. This field is the odd parity bit of the command field.

B.2.6 PTR

The pointer field is composed of a multiple of 8 binary bits. This field specifies a numeric pointer location that the command is referring to. If the value of the field is 255 or more, the 8 binary bits is represented by eight binary "1" bits (which equals 255), and the remaining value (subtracted by 255) is presented in the next 8 binary bits. The process is repeated as long as the remaining value is 255 or more. Therefore the last 8 bits of the pointer field always has a value of 254 or less.

B.2.7 P2

The P2 field is composed of 1 binary bit. This field is the odd parity bit of the pointer field.

B.2.8 LEN

The length field is composed of a multiple of 8 binary bits. This field specifies the numeric number of bits in the value field. If the value of the field is 255 or more, the 8 binary bits is represented by eight binary "1" bits (which equals 255), and the remaining value (subtracted by 255) is presented in the next 8 binary bits. The process is repeated as long as the remaining value is 255 or more. Therefore the last 8 bits of the pointer field always has a value of 254 or less.

B.2.9 P3

The P3 field is composed of 1 binary bit. This field is the odd parity bit of the length field.

B.2.10 VALUE

The value field is composed of a variable number of binary bits, specified by the length field. This field specifies the binary data that the tag will attempt to match against its own identifying code. The field terminates after l bits, where l is the numeric value of the length field.

B.2.11 P4

The P4 field is composed of 1 binary bit. This field is the odd parity bit of the length field.

B.2.12 P5

The P4 field is composed of 1 binary bit. This field is the odd parity bit of all the parity fields (P1, P2, P3, P4).

B.2.13 EOF

The End of Frame field is composed of 1 binary bit of "1". This field indicates the end of the actual command.

B.3 Reader Commands

These are the following commands that a Class 1 reader can issue:

B.3.1 ScrollAllID

A ScrollAllID command is characterized by the binary string "00110100" in the command field. All tags that receives the ScrollAllID command will backscatter a ScrollID Reply of their CRC, followed by the EPC code.

B.3.2 ScrollID

A ScrollID command is characterized by the binary string "00000001" in the command field. Tags matching the binary string in the value field beginning at the location defined by the pointer field will backscatter a ScrollID Reply of their CRC, followed by the EPC code.

B.3.3 PingID

A PingID command is characterized by the binary string "00001000" in the command field. Tags matching the binary string in the value field beginning at the location defined by the pointer field will backscatter a PingID Reply.

B.3.4 Quiet

A Quiet command is characterized by the binary string "00000010" in the command field. Tags matching the binary string in the value field beginning at the location defined by the pointer field will no longer respond to any future reader commands, until a Talk command is received.

B.3.5 Talk

A Talk command is characterized by the binary string "00010000" in the command field. Tags matching the binary string in the value field beginning at the location defined by the pointer field will respond to any future reader commands, until a Quiet command is received. This reader command is not used in *RFIDSim*'s implementation of the discovery process.

B.3.6 Kill

A Kill command is characterized by the binary string "00000100" in the command field. Tags matching the binary string (which contains the entire ITM - CRC, EPC and password) in the value field beginning at the location 0 will be permanently deactivated, and will no longer respond to any future reader commands. This reader command is not used in *RFIDSim*'s implementation of the discovery process.

B.3.7 ProgramID

A ProgramID command is characterized by the binary string "00110001" in the command field. Tags will reprogram the ITM of the tag, beginning at the location defined by the pointer field, with the binary string in the value field. The binary string in the value field must be exactly 16 bits, and the pointer must be a multiple of 16. A pointer to the MSB of the password field require the last 8 binary bits of the value field to be "0". This reader command is for reprogramming the ITM of a tag, and is not a required for the discovery phase. This reader command is not used in *RFIDSim*'s implementation of the discovery process.

B.3.8 VerifyID

A VerifyID command is characterized by the binary string "00110000" in the command field. All tags that receives the VerifyID command will backscatter a ScrollID Reply of their CRC, followed by the EPC code, and followed by the password. The command is ignored by a tag that has previous received a LockID command. This reader command is for reprogramming the ITM of a tag, and is not a required for the discovery phase. This reader command is not used in *RFIDSim*'s implementation of the discovery process.

B.3.9 LockID

A LockID command is characterized by the binary string "00110001" in the command field. LockID is a specific version of the ProgramID command that locks up the reader

to not respond to the VerifyID command. The value of the pointer field must point to the MSB of the password, the value of the length field must be 16, and the last eight bits of the value field must be the binary string "10100101". This reader command is for reprogramming the ITM of a tag, and is not a required for the discovery phase. This reader command is not used in *RFIDSim*'s implementation of the discovery process.

B.3.10 EraseID

A EraseID command is characterized by the binary string "00110010" in the command field. EraseID sets all bits of the CRC, EPC and password field to "0". The command is ignored by a tag that has previous received a LockID command. This reader command is for reprogramming the ITM of a tag, and is not a required for the discovery phase. This reader command is not used in *RFIDSim*'s implementation of the discovery process.

B.4 Reader Command Response Period

For Class 1 reader commands that awaits a backscatter response from a reader, there is a response period after the End of Frame field of the reader command, where tags are given the opportunity to backscatter their responses. The modulation of the response period differs depending on the command sent.

There are the following types of reply windows that a Class 1 reader might send after a command:

B.4.1 ScrollID Response Period

A ScrollID response period is characterized by a tag setup window (the length of 8 modulated binary bits), followed by a carrier-wave signal. The carrier-wave signal must not be longer than 20ms before the next reader command begins. This response period is sent immediately after the End of Frame field of the reader commands

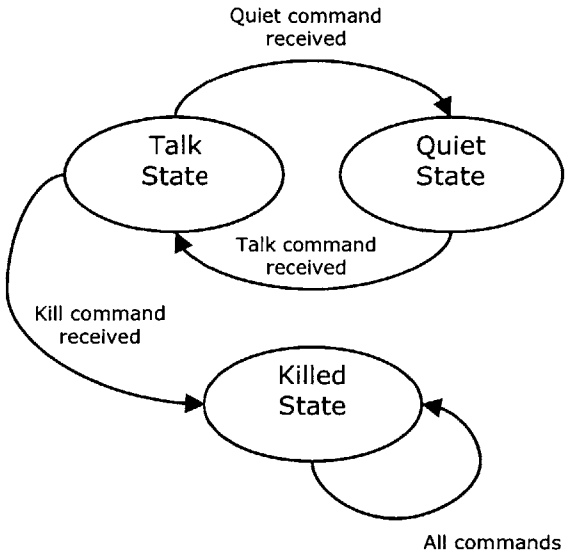

Figure B-3: State Machine defined for Class 0 tags

ScrollAllID, ScrollID, or VerifyID.

B.4.2 PingID Response Period

A PingAllID response period is characterized by a tag setup window (the length of 8 modulated binary bits), followed by 8 bin response windows (each the length of 8 modulated binary bits). This response period is sent immediately after the End of Frame field of the reader command PingID.

B.5 Tag States

Each Class 1 reader command sends all the required information for the tag to perform a pattern matching on its identification code. As a result, a Class 1 tag state machine only has three states, one of which is a permanent irreversible state transition. Figure B-3 shows the state transition diagram of a Class 1 tag.

B.5.1 Talk State

A Class 1 tag in this state evaluates all supported reader commands. The tag makes a state transition to the quiet state when it successfully evaluates a Quiet command. The tag makes a state transition to the killed state when it successfully evaluates a Kill command.

B.5.2 Quiet State

A Class 1 tag in this state ignores all supported reader commands, except for the Talk command. The tag only makes a state transition to the talk state when it successfully evaluates a Talk command.

B.5.3 Killed State

A Class 1 tag in this state ignores all supported reader commands, and is permanently deactivated. The tag does not make any more state transitions once it has entered the killed state, even after powering down.

B.6 Tag Backscatter Responses

The backscatter response includes binary data modulated using ASK. The data rate of a backscatter response is twice the data rate of the reader command data rate.

The following are backscatter responses a Class 1 tag can issue. The backscatter response channel is the same channel as the reader command is received in.

B.6.1 ScrollID Reply

The format of a ScrollID backscatter response consists of an 8-bit preamble string "11111110", followed by the CRC and the entire EPC code. The ScrollID backscatter response is sent when the tag receives a ScrollAllID command, or when the tag receives a ScrollID command, and the binary string in the value field beginning at the location defined by the pointer field matches the ITM.

The backscatter response begins in a predefined delay period after the tag setup window ends.

B.6.2 PingID Reply

The format of a PingID backscatter response consists of an 8-bit binary string. The PingID backscatter response is sent when the tag receives a PingID command, and the binary string in the value field beginning at the location defined by the pointer field matches the ITM. The tag backscatters the 8-bit string that follows the matching string in the value field (the beginning position of the 8-bit response is <value in the pointer field> + <length of the value field>).

The backscatter response begins in a predefined delay period after the tag setup window ends, plus n delay periods (each period the length of 8 modulated binary bits), where n is the numeric value of the first three bits of the 8-bit response.

For example, if the first three bits of the PingID Reply is "101" (which would translate to bin number 5), after the tag setup window ends, the backscatter response begins in:

<predefined delay period> + <5 * (length of 8 modulated binary bits)>

B.6.3 VerifyID Reply

The format of a VerifyID backscatter response is almost identical to the ScrollID backscatter response. The format consists of an 8-bit preamble string, followed by the CRC, the entire EPC code, and the password. The VerifyID backscatter response is sent when the tag correctly evaluates a VerifyID command from the reader. This backscatter response and its corresponding reader command is not used in *RFIDSim*'s implementation of the discovery process.

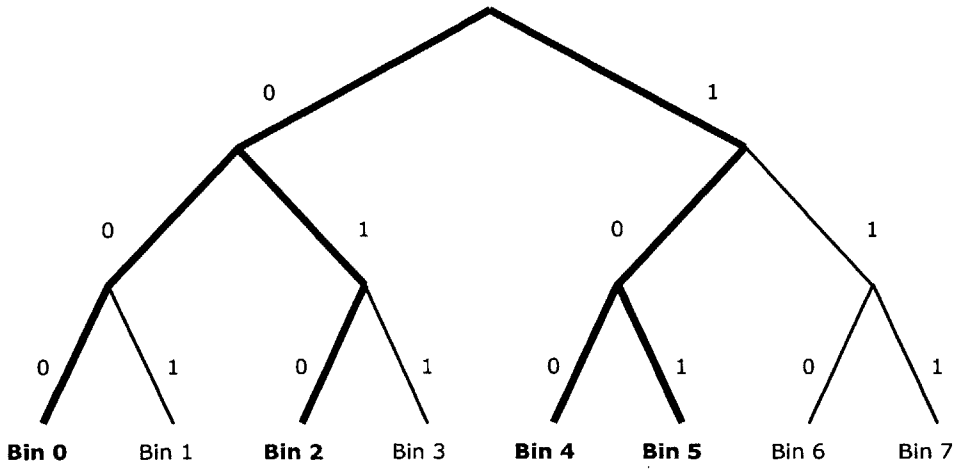

Figure B-4: Tree representation of walkable paths using bin modulation

B.7 Discovery Process

The Class 1 discovery algorithm implemented in *RFIDSim* uses a variation of the tree-walking algorithm that walks down three levels at a time.

B.7.1 Bin Modulation

The Class 1 discovery algorithm uses bin modulation of the PingID backscatter response to walk down three levels at a time.

As defined under section B.6.2, the PingID backscatter response from the tag begins after n bin response window periods, where n is the numeric value of the first three bits of the 8-bit response. By partitioning the PingID response window into 8 separate bin response windows, the reader can infer 3 bits of information based on which bin response window the PingID reply arrives in.

Figure B-4 shows the walkable paths in a binary tree form using the bin-modulated PingID replies from binary strings "10010101", "10100000", "01001010", "10000111", and "00001110". from the same PingID replies, based on only the first three bits of the backscatter responses.

B.7.2 Algorithmic Overview

1. The carrier-wave initialization and calibration phases are embedded in each reader command, so the tree-walking algorithm begins when the discovery algorithm begins. Because the reader command format requires at least one bit for the binary string of the value field, the reader pushes the binary "1" into the stack, and sends a PingID command with the pointer 0 and the binary string "0".

2. If there are no backscatter responses in any of the bin response windows, the reader broadcasts a PingID command with the next string in the stack. If the stack is empty, the reader goes to Step 8.

3. If there is k bin response windows with a backscatter response, where k is more than 1, the reader pushes k-1 binary strings into the stack, with each string composed of the identified string so far, together with the 3-bit binary value of the bin response window.

4. If there is only one bin response window with a backscatter response, the reader broadcasts a PingID command with the identified string, together with the 3-bit binary value of the bin response window.

5. After two consecutive PingID command on the same identified string, if the reader detects only one bin response window has a backscatter response for both commands, the reader concludes with a high confidence that there might be only one tag walking down that path. The reader will then attempt to perform a ScrollID command with the identified string. If the ScrollID reply can be demodulated correctly, and the CRC is verified, the reader proceeds to Step 8.

6. The process repeats at Step 2, and completes a successful singulation when a tag has responded all the way to the lth level of the tree, where l is the length of the identification code.

7. The reader broadcasts a Quiet command with the matching identification code, and the singulated tag enters a quiet state.

8. The reader, having identified and singulated one tag, pops the next string out of the stack and broadcasts the PingID command with the string. The process then

repeats by proceeding to Step 2. If the stack is empty, the reader proceeds to Step 9.

9. To ensure with a reasonable probability that the stack did not miss any tags that have not been singulated, the reader broadcasts a ScrollAllID command. If the reader receives any ScrollID reply in ScrollID response window, the reader restarts the algorithm by proceeding to Step 2. If the reader receives no backscatter responses from any tag, the reader assumes that all discovered tags are in quiet mode, and terminates.

B.7.3 Signal Overview

For the entire duration of the discovery process, the reader sends out commands consecutively, with a short period of no wireless signal in between each command. The success of the algorithm, unlike Class 0, does not require a tag to receive every single signal for the duration of the discovery process, as each reader command encodes sufficient information for a tag to locate the pointer and the identified string it is trying to match. As a result, even if a tag is unable to demodulate a reader command, the tag will still understand the next command sent by the reader.

B.7.4 Analysis

The length of each reader command is fairly long, because of the large amount of information modulated in each command, as well as the large amount of information modulated in backscatter responses. As a result, each reader command sent out is rather costly in terms of speed. The algorithm attempts to increase the performance of the discovery process, by using bin modulation to walk down three levels at a time (therefore using sending out less reader commands), and also by using ScrollID to singulate an entire tag when it has a high confidence of singulation.

However, because Class 1 tags are given all information needed to match identifying strings for each reader command, the tags are only required to track the transitions between Quiet state and Talk state, which leads to a simpler logic design.

Bibliography

[1] Daniel W. Engels, Tom A. Scharfeld, Sanjay E. Sarma. *Review of RFID Technologies.* Submitted for publication in the IEEE Sensors Journal, 2001.

[2] Tom A. Scharfeld. *An Analysis of the Fundamental Constraints on Low Cost Passive Radio Frequency Identification System Design.* Masters Thesis submitted to the Department of Mechanical Engineering at Massachusetts Institute of Technology, August 2001.

[3] Klaus Finkenzeller. *RFID Handbook: Radio-Frequency Identification Fundamentals and Applications.* John Wiley & Sons Ltd, 1999.

[4] RFID Journal. *Gillette to Buy 500 Million EPC Tags.* RFID Journal, November 2002.

[5] Auto-ID Center. *860MHz-930MHz Class 0 Radio Frequency Identification Tag Radio Frequency & Logical Communication Interface Specification Candidate Recommendation, Version 1.0.0.* Auto-ID Center, August 2003.

[6] Auto-ID Center. *860MHz-930MHz Class 1 Radio Frequency Identification Tag Radio Frequency & Logical Communication Interface Specification Recommended Standard, Version 1.0.0.* Auto-ID Center, August 2003.