Distributed Policy-Based Management Framework for Wireless Sensor Networks

By

Nidal Qwasmi

A thesis submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

In

Electrical and Computer Engineering
Faculty of Engineering and Applied Science

At

University of Ontario Institute of Technology (UOIT)
Oshawa, Canada
June 2014

CERTIFICATE OF APPROVAL

ABSTRACT

Policy-Based Management Systems (PBMS) are becoming a critical component of any information technology environment, due to their ability to abstract hardware complexity from their users. Policy-based systems exist in such areas as data center management, security, privacy, and computer network management. The Wireless Sensor Network (WSN) is no exception, although implementation of policy-based management in a WSN is still in its infancy. Wireless Sensor Networks (WSNs) are particularly challenging due to many characteristics, such as a working environment that makes maintenance and support a challenge; a deployment scale of hundreds, if not thousands, of nodes; and constrained hardware resources. Memory, processing, and battery power are limited, making WSNs capable of handling only applications with limited resource requirements. Consequently, the implementation of policy-based management applications on WSNs has to tackle these characteristics of WSNs and take these limitations into consideration during the design phase. Therefore, due to hardware resource constraints, policy-based management applications on WSNs can store only a limited number of policies in the local memory of a sensor node and must recycle them when additional policies are required. This recycling process creates communication overhead on the network and requires a policy deployment mechanism. The communication overhead will logically reduce the lifetime of the sensor's batteries, and the policy's deployment mechanism dictates system limitations and capabilities. To tackle these challenges, a new distributed policy-based management framework named TinyPolicy has been devised, which can store, locate, access, and execute any policy in the WSN. This new framework uses a newly created policy deployment mechanism named PolicyP2P, which is designed to make the distributed policy-based management system more robust against node failure, eliminate the threat of single points of failure, and improve policy availability. More importantly, it will increase the total number of policies that can be deployed in the WSN, which will result in more manageable constraints or tasks.

Keywords: Distributed systems, Policy management, Distributed policy management, Wireless Sensor Network (WSN), PolicyP2P, TinyPolicy.

DEDICATION

I dedicate my thesis work to my beloved family, who have believed in me and supported me throughout the process.

ACKNOWLEDGMENTS

A special thanks to Dr. Ramiro Liscano, my advisor, for his patience and support throughout the entire process. I also wish to thank my committee members, Dr. Khalil El-Khatib and Dr. Shahram Heydari, for agreeing to serve on my thesis committee and for their valuable expertise and precious time. Finally, I would like to acknowledge and thank the University of Ontario Institute of Technology (UOIT) for allowing me to conduct my research and providing remarkable support.

TABLE OF CONTENTS

LIST OF FIGURES ix

LIST OF TABLES xi

LIST OF EQUATIONS xii

ACRONYMS AND ABBREVIATIONS xiii

SYMBOLS xv

Chapter 1 Background 1

1.1 Introduction 1

1.2 Motivational challenges 2

1.3 Thesis objectives 6

1.4 Thesis contributions 7

1.5 Outline 8

Chapter 2 Related work 9

2.1 Distributed policy-based management 15

2.2 Policy-based management for WSN 16

2.3 Policy structure and protocol 23

2.4 P2P algorithms in WSN 24

Chapter 3 TinyPolicy: A Distributed Policy Framework 27

3.1 Software components 29

3.2 Data repositories 30

3.3 Modified policy structure 32

3.4 Multiple policies 33

Chapter 4 Policy management in TinyPolicy 35

4.1 Policy creation algorithm 36

4.2 Policy modification and deletion 38

4.3 Policy execution 39

4.4 Policy retention algorithm 41

4.5 Multiple policies 42
Chapter 5 Bloom filter 43
5.1 Bloom filter implementation 44
5.2 Bloom filter analysis 45
5.3 Hashing algorithms 46
5.4 Bloom filter evaluation 49
Chapter 6 PolicyP2P – A Policy Overlay Network 53
6.1 Determining policy key 55
6.2 Distributed Policy Addressing 58
6.3 PolicyP2P algorithm 59
6.4 Policy lookup 61
6.5 Network formation 61
6.6 Node joining the network 65
6.7 Node leaving the network 66
6.8 Network structure maintenance 68
Chapter 7 Complexity analysis of TinyPolicy 72
7.1 Network formation messages 74
7.2 Overhead messages 75
7.3 Policy administration message 77
7.4 Data analysis 83
Chapter 8 Validation of TinyPolicy through implementation in TinyOS 89
8.1 Policy management tool (Policy IDE) 99
8.2 Thesis validation 101
8.3 Increase the ability to support more policies in a WSN 102
8.4 Improve robustness of the distributed policy framework for WSNs 108
8.5 Streamline the policy distribution processes 115
8.6 Framework limitations and constraints 119
Chapter 9 Conclusions and Future Work 125
References 129
Appendixes 135
Appendix A Policy management tool (Policy IDE) interface 136

Appendix B Mathematical Model Data 147
Appendix C Publications.. 160

LIST OF FIGURES

FIGURE 1 OBLIGATION POLICY 10
FIGURE 2 AUTHORIZATION POLICY 11
FIGURE 3 POLICY-BASED MANAGEMENT SYSTEM ARCHITECTURE 18
FIGURE 4 LAYERED FUNCTIONAL ARCHITECTURE OF WSNS 20
FIGURE 5 A SELF-MANAGED CELL WITH SELF-HEALING SERVICES (GRAY BOXES) 20
FIGURE 6 FINGER2 ARCHITECTURE 22
FIGURE 7 DISTRIBUTED POLICY FRAMEWORK 29
FIGURE 8 FINGER2 POLICY STRUCTURE 32
FIGURE 9 TINYPOLICY POLICY STRUCTURE 33
FIGURE 10 COMPLEX POLICY STRUCTURE 34
FIGURE 11 POLICY CHAIN 34
FIGURE 12 POLICY CREATION PROCESS 37
FIGURE 13 POLICY CREATION GUI 37
FIGURE 14 POLICY MODIFICATION AND DELETION PROCESS 39
FIGURE 15 POLICY EXECUTION PROCESS 41
FIGURE 16 POLICY RETENTION PROCESS 42
FIGURE 17 BLOOM FILTER ANALYSIS FOR A SAMPLE SIZE OF 1,024 MEMBERS 50
FIGURE 18 BLOOM FILTER ANALYSIS FOR A SAMPLE SIZE OF 18,000 MEMBERS 51
FIGURE 19 POLICY KEY 57
FIGURE 20 POLICY STORAGE EXAMPLES 59
FIGURE 21 POLICYP2P 60
FIGURE 22 OVERLAY NETWORK FOR POLICY-BASED SYSTEMS 62
FIGURE 23 TREE STRUCTURE FOR THE OVERLAY NETWORK 63
FIGURE 24 MESSAGE SEQUENCE FOR A NEW NODE JOINING THE WSN 64
FIGURE 25 MESSAGE SEQUENCE FOR A JOIN REQUEST TO AN EXISTING WSN 66
FIGURE 26 MAINTENANCE REQUEST PROCESS 68
FIGURE 27 MAINTENANCE REQUEST ACTIVITIES 70
FIGURE 28 MONITORING POLICY REQUEST FLOWCHART 71
FIGURE 29 RE-JOIN RESPONCE MESSAGE 76
FIGURE 30 TOTAL NUMBER OF MAINTENANCE MESSAGES 77
FIGURE 31 TOTAL NUMBER OF ADMINISTRATIVE MESSAGES 79
FIGURE 32 PERCENTAGE OF TOTAL FORMATION TIME 84
FIGURE 33 POLICY LOADING PERFORMANCE 85
FIGURE 34 BLOOM FILTER PERFORMANCE WITH MISSING POLICIES RATE OF 30% 86
FIGURE 35 CENTRAL AND DISTRIBUTED POLICY REPOSITORY PERFORMANCE 88
FIGURE 36 DEVELOPMENT ENVIRONMENT 91
FIGURE 37 PHYSICAL SETUP 92

FIGURE 38 MODULE DIAGRAM 96
FIGURE 39 COMPONENTS DIAGRAM 1/2 97
FIGURE 40 COMPONENTS DIAGRAM 2/2 98
FIGURE 41 IDE ARCHITECTURE 100
FIGURE 42 POLICY EXECUTION STEP 1 103
FIGURE 43 POLICY EXECUTION STEP 2 105
FIGURE 44 POLICY EXECUTION STEP 3 106
FIGURE 45 POLICY EXECUTION STEP 4 107
FIGURE 46 OVERLAY NETWORK FORMATION STEP 1 109
FIGURE 47 OVERLAY NETWORK FORMATION STEP 2 111
FIGURE 48 OVERLAY NETWORK FORMATION STEP 3 112
FIGURE 49 NODE FAILURE CASE 114
FIGURE 50 LOADING FIRST POLICY 108F 117
FIGURE 51 LOADING SECOND POLICY 208F 118
FIGURE 52 NETWORK TOPOLOGY LIMITATION 120
FIGURE 53 COMPILATION ERROR WHEN POLICY REPOSITORY SIZE REACHED 1928 120
FIGURE 54 COMPILATION EXPERIMENT WITH DIFFERENT POLICY REPOSITORY SIZE 122
FIGURE 55 COMPILATION ERROR WHEN NODE REPOSITORY SIZE REACHED 10922 122
FIGURE 56 COMPILATION EXPERIMENT WITH DIFFERENT NODE REPOSITORY SIZE 123

LIST OF TABLES

TABLE 1 POLICY TYPES AND EXAMPLES 13
TABLE 2 CODE SIZE BREAKDOWN OF SIMAPP 23
TABLE 3 PROCESSING DELAYS 23
TABLE 4 CENTRALIZED VERSUS DISTRIBUTED POLICY REPOSITORY 27
TABLE 5 POLICY MANAGEMENT MESSAGES 35
TABLE 6 ADVANTAGES OF USING BLOOM FILTER 43
TABLE 7 HASHING ALGORITHMS COMPARISON 48
TABLE 8 EXPERIMENTAL RESULTS FOR 1,024 MEMBERS 51
TABLE 9 EXPERIMENTAL RESULTS FOR 18,000 MEMBERS 52
TABLE 10 NETWORK MESSAGES 54
TABLE 11 OVERLAY NETWORK VERSUS PHYSICAL NETWORK 55
TABLE 12 OVERLAY NETWORK SIZE 63
TABLE 13 NETWORK MESSAGE SIZES 72
TABLE 14 NETWORK PERFORMANCE FOR WSN WITH 20 NODES 80
TABLE 15 NETWORK PERFORMANCE FOR HETEROGENEOUS WSN WITH 20 NODES 82
TABLE 16 PROGRAM SIZE IN BYTES 94
TABLE 17 POLICY REPOSITORY MAXIMUM LIMIT EXPERIMENT 121
TABLE 18 NODE REPOSITORY MAXIMUM LIMIT EXPERIMENT 123
TABLE 19 CONTRAST BETWEEN TINYPOLICY AND FINGER2 127

LIST OF EQUATIONS

(1)… 31
(2) … 31
(3) … 46
(4) … 46
(5) … 46
(6) … 47
(7) … 56
(8) … 74
(9) … 74
(10) … 74
(11) … 75
(12) … 75
(13) … .... ……77
(14) … 78
(15) … 78
(16) … 79
(17) … 79
(18) … 80
(19) … 80

ACRONYMS AND ABBREVIATIONS

- CNMS: Cognitive Network Management System
- DMTF: Distributed Management Task Force
- DHT: Distributed Hash Table
- Finger/Finger2: Policy-based platform for Wireless Sensor Network
- GUI: Graphical User Interface
- IETF: Internet Engineering Task Force
- IoT: Internet of Things
- LPDP : Local Policy Decision Point
- MANET: Mobile Ad hoc Network
- NGN: Next Generation Networks
- P2P: Peer-to-Peer
- PBM: Policy-Based Management
- PBMS: Policy-Based Management System
- PCIM: Policy Core Information Model
- PDP: Policy Decision Point
- PDA: Personal Digital Assistant
- PEP: Policy Enforcement Point
- Policy Key (Policy-ID): Identification data of a policy in PBM systems. It is imperative to locate any particular policy within the network.
- PolicyP2P: The PolicyP2P software component consists of all algorithms that are required by the overlay network to operate. PolicyP2P is an algorithm created by this research to find the closest matching node ID to the policy key.
- QoS: Quality of Service
- QoE: Quality of Experience
- SensorML: Sensor Model Language
- TinyPolicy: A framework for distributed policy-based management, presented in this thesis
- VM: Virtual Machine

- WBAN: Wireless Body Area Network
- WSN: Wireless Sensor Network

SYMBOLS

- $\|$: concatenation
- %: modulo operator; same as “mod”
- $\nexists$: not existence qualifier (there does not exist); same as “$\neg\exists$”
- $\in$: element of
- $\notin$: not an element of
- $\approx$: almost equal to
- $\wedge$: logical *and*
- $\sum$: summation
- P : total number of policies in the local policy repository
- P_i : total number of policies in the local policy repository of node number *i*
- p_{ji}: policy number *j* on node number *i*
- a_i : overlay address of node number *i*
- a'_{ji} : overlay address of leaf node number *j* of parent node number *i*
- T : transmission speed
- L : length in bytes of the overlay address
- v : overlay tree level number
- b : a specific number of bits which represents nodes in one level v of the overlay tree structure
- f : total number of leaf nodes (children) per parent node
- $O(\,)$: big O notation, to express the time complexity of an algorithm

Chapter 1 Background

In this chapter, the motivational challenges and thesis objectives are discussed, followed by the contributions of this thesis and an outline of the chapters.

1.1 Introduction

Sensors are becoming part of our daily life, finding their way into such fields as environmental, medical, and military. Many examples of such applications are presented in Gutiérrez et al. [1] Wireless sensor networks (WSNs) collect sensing data from the surrounding environment. Each WSN contains a number of sensors, each of which is responsible for monitoring one or more events. Therefore, it is likely that a WSN will contain different types of sensors from various manufacturers. As a result, a WSN usually works in a heterogeneous environment where sensors are incompatible with different hardware and software standards and from different manufacturers. Even though certain types of sensors may overcome some of these problems, this usually proves complex and costly[2]. To overcome some of these challenges and to conceal the complexity of the underlying network devices from the human operator, researchers have considered Policy-Based Management (PBM) platforms a viable solution [3], [4].

WSNs pose particular challenges due to such characteristics as the working environment (such as in animal habitats, underwater, on volcanoes, and inside the human body) which complicates maintenance and support, and limited hardware resources, particularly memory, processing and battery power, which require software with minimum power and memory usage [5]. Consequently, the administration of WSNs is becoming a challenge [5], due to the working environment and heterogeneous sensors on different systems. These characteristics naturally constrain the capabilities of the applications that run on the WSN. Policy-Based Management (PBM) as an implementation on WSNs is no different, and these limitations should be taken into consideration when designing any solution for WSNs. Due to these limitations, as shown by Zhu et al. [6], devices in a WSN with the Finger platform installed can store a limited number of policies in their memory and recycle them when required. The number of policies in the WSN is directly related to the number of constraints that can be created on the WSN, which logically equal the number of governing functions that can be performed. Therefore, the more policies the WSN can accommodate, the more governing functions (constraints) the users of the WSN can create.

1.2 Motivational challenges

The sensors' harsh and unrestricted work environment requires sensor nodes to be small and inexpensive, with limited sensing, computation and wireless transmission capabilities [7]. A typical sensor device (such as Iris Mote, Mica, MicaZ, TelosB, TMote Sky, and Sentilla JCreate) is equipped with an 8- or 16-bit CPU running at 4-8 MHz, 2-10 kB RAM, 30-128 kB flash memory [1], [8], and a radio transmission rate up to 250 kbps with a range of a few hundred meters [9]. Further improvements in operating conditions may come from the use of energy efficient 32-bit CPUs and from research efforts to invent a renewable energy sensor by harvesting energy or to create an energy-free sensor by using ambient RF as the only source of power [10]. Still, to keep cost and power consumption as low as possible, sensor nodes remain resource constrained compared to a smartphone or tablet. The resource-constrained nature of the sensor devices and their heterogeneous working environments suggest that resource sharing and policy-based management would be an ideal solution for such environments.

Prior research and real world experience support our claim that resource sharing and policy-based management are an ideal solution for WSNs. In the health care domain [11], the Wireless Body Area Network (WBAN), a type of WSN, can provide an affordable and proactive health care system to monitor patient health conditions. This solution can save lives, improve the quality of life, and reduce health care costs by reducing hospital stays. Major challenges for WBANs listed by Movassaghi et al. [11] in an extensive survey of the state of the art in WBANs include the following:

- Today, sensor nodes are still constrained by limited resources, due to several factors. The sensor nodes are small in size, which limits hardware enhancement. The WBAN area (the human body) is, of course, limited in size too, which has a huge influence on the acceptable size of the sensor node. Economic forces are another factor; nodes must cost as little as possible.

- Accessing implanted settings and replacing implanted nodes can be quite difficult. The difficulties of replacing nodes and altering their behaviors make it important to find alternatives to physical access to the implanted nodes.

- Network physical area size is limited to human body size, for which large size devices are unsuited and which rule out the use of larger size sensor nodes with greater capabilities.

- The size of each sensor node has to be as small as possible, due to the limited physical size of the WBAN.

Our work overcomes most of the previously listed challenges in WBANs, because TinyPolicy is based on two main concepts. The first concept is resource sharing, which overcomes the sensor's resource limitations and the need for larger size sensors. The second concept is controlling the sensor behavior by policy rather than by reprogramming the node, which avoids physical replacement of the node.

In the agriculture domain, Gutiérrez et al. [1] developed an automated irrigation system to reduce the waste of water used for agriculture crops. The system consists of a distributed WSN to monitor soil moisture and temperature, actuators to control the irrigation system, and a gateway unit to handle sensor information. The system monitors such environmental parameters as soil moisture and temperature by using sensors deployed in plant root zones. Researchers in [1] resolve the energy constraints by using photovoltaic panels to recharge AA 2000-mAh Ni-MH CycleEnergy batteries, and resolve the need to reprogram sensor nodes due to changes in thresholds by frequently sending the sensing data to a centralized unit which has more capabilities. The TinyPolicy framework can assist in this case by enabling the control of thresholds directly on the sensor node by using policies to avoid unnecessary transmission of data to a central node, which may reduce energy consumption. In addition, the TinyPolicy framework can enhance system reliability by using a distributed approach rather than a central node, which creates a single point of failure in the system.

In the natural science domain, scientists rely on WSNs to help address previously insoluble scientific questions. For example, Naumowicz et al. [12] successfully designed and deployed a WSN to monitor seabirds on Skomer Island, a UK National Nature Reserve. The complexity of reprogramming the WSN software proved to be a big disadvantage; they had to rely on a computer science team to do this work for them, which resulted in delays and distracted the natural scientists from their core goals. (A new approach to programming the WSN is currently being investigated.) A policy-based system, such as TinyPolicy, would be a good alternative way to handle such cases, as the behavior of the WSN would be controlled by policies rather than by reprogramming the sensor's firmware.

In the civil engineering domain, Kim et al. [13] designed, implemented, deployed and tested a WSN for Structural Health Monitoring (SHM) on the 4200 ft long main span and the south tower of the Golden Gate Bridge (GGB), the largest WSN deployment for SHM to date. Limited RAM on each sensor node proved to be an obstacle to resolving the packet size issue. The TinyPolicy framework can help deal with memory limitation by sharing memory resources with other capable sensors in the WSN. The trade-off here is between freeing more local memory and increasing transmission activities, but the actual trade-off numbers need to be investigated.

Due to memory capacity limitations, a sensor device may hold a limited number of policies at any given time, which may not always be sufficient. These limitations may severely restrict the management capabilities and number of tasks that can be performed on the device and on the WSN as a whole. Therefore, dynamic deployment of policies is necessary to utilize node resources efficiently and to execute the required policies accurately.

The architectures of many existing and proposed policy-based WSN platforms rely on local policy repositories on the nodes to access any required policy. (Some of these architectures are discussed in more detail in Chapter 2 Related work) This type of architecture raises many serious issues, particularly the issue of network dynamism and robustness, since it creates node silos, which can communicate with the network gateway but do not communicate or share resources with other nodes in the network. In addition, it may cause longer disruptions to node service, because a defective node will need to be replaced with an exact replica in order to resume service.

Moreover, this architecture creates serious administrative overhead during the deployment of new policies or the replacement of a defective node, because the administrator needs to create an exact replica of the defective node with all applicable policies stored on it. Furthermore, the administrator has to make sure that the new policies have been deployed successfully on the targeted node, which also adds extra overhead to the task.

WSN implementation dictates the required number of nodes and policies. Kim et al. [13] studied the Golden Gate Bridge (GGB) where 64 nodes are distributed over 4200ft bridge body. Each sensor monitors ambient vibrations and strong motion. Major requirements of this system as mentioned in [13] are signals quality (such as noise floor of the system, installation error, and temperature variation), sampling rate, time synchronization, multi-hop routing, and reliably dissemination (such as data lost and blockage of hopping). Hence, the total number of parameters is eight parameters each of which required five policies in average (such as authorization, installation, max, min, and acceptable range policy). Therefore, each node needs at least 40 policies (exceeding the local policy repository capacity in Finger2 platform). Hence, the total number of policies for this system is 2,560 policies (40 policies per node multiplied by 64 nodes).

1.3 Thesis objectives

The goal of this thesis is to specify a fully distributed policy-based framework for WSNs. This new framework will meet the following objectives when compared to a conventional non-distributed policy platform:

Increase the ability to support more policies in a WSN.

Due to the nature of limited resources on the sensor node memory as discussed by Zhu et al. [6], it is quite possible for a policy-based WSN network to have more policies than the sensor node capacity. The number of policies in the WSN is directly connected to the number of constraints that can be created on the WSN, which logically equals the number of functions that can be performed on the WSN. Therefore, the more policies the WSN can accommodate, the more management functions (constraints) the users of the WSN can perform.

Improve the robustness of the distributed policy framework for a WSN.

The existing architecture creates node silos, which can only communicate with the network gateway but do not communicate with other nodes in the network. Our framework creates a structured peer-to-peer (P2P) overlay network, in which all nodes can share resources and which has a maintenance mechanism to maintain the network structure.

Streamline the policy distribution processes.

As shown in [6], [14], [15], the architectures of many existing or proposed policy-based WSN platforms rely on a local policy repository on each node to access any required policy. This type of architecture creates serious administrative overhead during the deployment of new policies or replacement of a defective node, because the network operator needs to push all applicable policies to the targeted node before deploying it in the WSN. In our framework, the new node will pull all required policies from other nodes in the network after they are deployed into the WSN; no human intervention will be needed.

1.4 Thesis contributions

The primary contributions of this thesis are the following:

1. Designing a novel framework for a fully distributed policy-based system. Details are discussed in Chapter 3 TinyPolicy: A Distributed Policy Framework.

   1.1 Developing a new distribution technique for policies in a WSN by creating a new policy-centric P2P algorithm named PolicyP2P. Details are discussed in section 6.3.

   1.2 Introducing and analyzing a new implementation for a Bloom filter in the areas of WSN and policy-based systems. Details are discussed in Chapter 5 Bloom filter.

   1.3 Introducing a new approach for constructing a policy key by using a sensor's data rather than by using arbitrary numbers as in other existing systems and platforms. Details are discussed in section 6.1.

2. Creating a new tool for policy debugging and testing, named Policy IDE. This new tool allows the users to test and debug the newly created policy in a simulation environment through a simple GUI. Details are discussed in Chapter 8 Validation of TinyPolicy through implementation in TinyOS and Appendix A Policy management tool (Policy IDE) interface.

3. As a contribution to the WSN research community, our work was used as a basis for other open source projects, such as [16] and [17], which inspire other researchers abroad.

4. Publications related to our work are listed in Appendix C Publications.

1.5 Outline

This thesis consists of nine chapters and is organized in the following way: Chapter 1 Background covers the thesis background, motivations, objectives, and contributions. Chapter 2 Related work discusses different knowledge areas and related work. This chapter is divided into four sections, each dealing with a separate knowledge area. Chapter 3 TinyPolicy: A Distributed Policy Framework discusses the TinyPolicy framework and architecture design. Chapter 4 Policy management in TinyPolicy discusses policy management algorithms in TinyPolicy; these algorithms deal with policy creation, modification, deletion, execution, retention, and the handling of multiple policies. Chapter 5 Bloom filter describes the Bloom filter analysis, implementation, and evaluation, and its value for the framework. Chapter 6 PolicyP2P – A Policy Overlay Network discusses the PolicyP2P software component, which consists of all algorithms that are required by the overlay network to operate. Chapter 7 Complexity analysis of TinyPolicydiscusses the results of the complexity analysis of the overlay network. Chapter 8 Validation of TinyPolicy through implementation in TinyOS discusses the implementation and evaluation of the framework. It also introduces the Policy Management Tool, which provides great assistance in managing the policy-based environment (create, delete, enable and disable a policy, and trigger an event), and in debugging and testing policy execution. Chapter 9 Conclusions and Future Work briefly summarizes this research and proposes future work and improvements.

Chapter 2 Related work

Various knowledge areas were studied in this research, such as distributed policy-based management, policy-based management for WSNs, policy structure, and protocol and P2P algorithms for WSNs.

Many of the existing or proposed policy-based WSN platforms rely on a local policy repository on the sensor node to access any required policy [6], [14], [15]. This type of architecture raises many serious issues; particularly the issue of network dynamism and robustness, since it creates node silos that can only communicate with the network gateway but do not communicate or share resources with other nodes in the network. Moreover, this architecture creates administrative overhead during the deployment of new policies or the replacement of a defective node, because the administrator needs to know exactly which policies apply to which nodes, the address of the targeted node, and how to create an exact replica of the defective node. Moreover, the administrator of the existing architecture has to make sure that the new policies have been deployed successfully to the targeted node, which also adds extra overhead to the task. Our new framework can avoid this additional overhead by deploying the new policy to a hosted node that has been mathematically selected, rather than deploying it directly on the targeted node. The targeted node can access the new policy from the hosted node when it is required or from the Root if the deployment of the new policy on the hosted node was not successful.

Policy is defined as a constraint on the system behaviors, which can be expressed using natural language or mathematical notation. However, neither of these two approaches is ideal for computer systems [18]. Natural language is commonly used to write real-life policies, but it typically lacks clarity and precision [18]. Mathematical notation, on the other hand, has extreme clarity and precision, though it suffers from limited ability to express constraints and is difficult to understand [4]. Policy-based systems try to strike a balance between these two approaches by creating a policy language that can fulfill the requirements of the targeted system. Hence, policy languages are declarative and not procedural; they express constraints on system behaviors but do not specify how these constraints ought to be enforced [19].

Policy-based systems use many existing expressive languages for specifying policies. Policy languages include XACML (eXtensible Access Control Markup Language) from OASIS [20], Ponder2 from Imperial College in London [21], PDL (Policy Description Language) from Bell [22], CQL (CIM Query Language) from DMTF [23], and CIM-SPL (Simple Policy Language CIM) from DMTF. However, they are not appropriate for WSNs due to resource constraints in the sensor node. Some of these constraints are memory, computational power, and limited wireless signal range. In fact, frequently changing network topology, limited wireless signal range, and limited resources are considered the most challenging issues in designing a policy system for WSNs [24].

The most notable initiative in dealing with this issue of policy language was Finger2, an embedded policy system for wireless sensor nodes, which was a simplified and scaled-down version of Ponder2 [15]. Finger2 uses the PonderTalk [21] object-oriented policy language because of its simplicity, and it can efficiently exchange messages between objects. PonderTalk is a slightly modified version of Smalltalk [25] that was created at the Department of Computing in Imperial College, London [26]. PonderTalk has two types of policies, Obligation policies and Authorization policies. Obligation policies monitor events, apply conditions, and trigger actions [26]. Figure 1 shows the syntax of the obligation policy.

```
Policy := root/factory/ecapolicy create.
Policy event:  myEvent;
condition: [:arg | bool-expression];
action:  [:arg | statements]
```

Figure 1 Obligation Policy

As shown in Figure 1, the obligation policy structure consists of the following parts:

- ***Policy ID (policy name)***: A unique identification number or string that identifies the policy.

- ***Policy Event***: An identification number or string that identifies the unique event associated with the policy.

- ***Condition***: An expression that the policy engine evaluates to trigger the associated action.

- ***Action***: The task to be performed if the condition in the policy is positive.

The second type of policy is the authorization policy, which is used to authorize access to secured resources [26]. Figure 2 shows the syntax of the authorization policy.

```
Policy := root/factory/authpolicy
subject: root/personnel/nurse/ward1
action: “getrecord”
target:  root/patient/ward1
focus: “t”
```

Figure 2 Authorization Policy

As shown in Figure 2, the authorization policy structure consists of the following parts:

- ***Policy ID (policy name)***: A unique identification number or string that identifies the policy.

- ***Subject***: An object that has the permission. In this example, it is the nurse in Ward1.

- ***Action***: The transaction type (task to be performed). In this example, it is get patient medical record.

- ***Target***: An object that the permission is given about. In this example, it is the patient in ward 1.

- ***Focus***: This field shows which object the policy is intended to protect. In this example, it is the target (patient in ward 1).

The WSN environment is constrained due to limited resources, such as energy, memory, and processing power. Such limitations affect the number of applicable languages that can efficiently operate with it. In addition to the limitations of the operating environment, the selection of language is further limited by the fact that the language needs to work efficiently to exchange messages between managed objects and be able to handle the policy structure and operations.

In policy-based management systems, alternatives to policy languages to transform a policy into a physical implementation include the following:

- ***Transformation using static rule***: A system expert creates a static mapping between the high-level policy and low-level implementation. For example, suppose a service provider has a policy to provide a specific level of service based on the user's company. This policy could be translated to: if user from subnet 10.10.3.0/24 then reserve 20 Mbps and use encryption 128 bits [18].
- ***Transformation using policy table lookup***: The system stores a table of policies used by the system [27]; the administrator queries the table with a set of configuration parameters to obtain a set of goals that can be achieved for those parameters [18].
- ***Transformation using Case-Based Reasoning***: A use case database or history of the system behavior [28] is employed to transform high-level policies or goals into low-level configuration parameters and vice versa [18].

Agrawal et al. [18] provided a convincing classification of different policy types that links the definitions to the system's various states and behaviors. The classifications consist of the following:

- ***Configuration constraint policy:*** This type defines configuration constraints, such as allowable, minimum, and maximum values for configuration attributes. Examples:
    - Maximum number of threats for application server is 50
    - Virtual memory size should be less than two times the size of physical memory

- ***Metric (Goal) constraint policy:*** This type defines metric constraints, such as upper or lower bound on a metric. Examples:
    - Keep CPU utilization below 50%
    - Directory lookup should be completed in less than one second
- ***Action policy:*** This type requires the system to perform certain actions when a particular event or change in system status has occurred. Examples:
    - If CPU utilization exceeds 70% then allocate additional server
    - If system temperature exceeds 95° C then shut down the system
- ***Alert policy:*** This type is similar to the action type, except that in this type, the action is a notification message sent to another entity. Examples:
    - If users did not access their email accounts in more than 6 months, notify them by email
    - If the system goes down, notify the administrator

This classification was for wired network environments. Nevertheless, it can be valid for the wireless sensor network environment as well. Table 1 presents a mapping of Agrawal's policy type classifications to WSNs.

Table 1 Policy Types and Examples

Policy Type	**Description**	**Policy Examples**	
		Wire Network	**DPBM-WSN**
Configuration	Define configuration constraints, such as allowable, minimum, maximum values for configuration	➢ Maximum number of threats for application server is 50 ➢ Virtual memory size should be	➢ Increase/decrease the timing event frequency ➢ Increase/decrease the sensing rate

	attribute	less than two times the size of physical memory	
Metric	Define metric constraints, such as upper or lower bound on a metric	➢ Keep CPU utilization below 50% ➢ Directory lookup should be completed in less than one second	➢ Increase the sensing rate by 10% if the difference between the last two readings is 20% ➢ Decrease the transmission rate by 20% if battery level is less than 10%
Action	Require the system to perform certain actions when a particular event or change in system status has occurred	➢ If CPU utilization exceeds 70% then allocate additional server ➢ If system temperature exceeds 95° C then shut down the system	➢ If the sensing data storage exceeds 90% utilization then switch to another storage node ➢ If parent node is not accessible then try to join another parent node
Alert	Similar to the action type except that in this type, the action is a notification message sent to another entity	➢ If users did not access their email accounts for more than 6 months, notify them by email ➢ If the system goes down, notify the administrator	➢ If battery level is below 10%, notify the administrator ➢ if policy storage is 90% utilized, notify the administrator

2.1 Distributed policy-based management

Distributed mechanisms have been used to resolve resource constraints in many knowledge areas, such as distributed computing, distributed file systems, distributed learning, and distributed manufacturing. There has been a great deal of research on distributed policy-based management of types of networks other than WSNs. These initiatives include the following.

The Madeira project [29] is a research project to develop solutions to Next Generation Networks (NGN) challenges. This project uses a fully distributed policy-based network management framework, which exploits the peer-to-peer paradigm. Researchers justify the use of policy-based and peer-to-peer approaches in the Madeira project as compensation for the lack of flexibility, dynamism, and autonomy that the NGN paradigm requires. Madeira achieves these objectives by developing an overlay mesh network of distributed management elements. Each management element will be responsible for managing a subset of the network independently from other subsets of the network. The approach adapted by the Madeira project is similar to that in this thesis, in that both use the policy-based management concept supported by an overlay network structure.

Galani et al. [30] researched a policy-based framework as a feasible solution for the Future Internet. Authors defined the Future Internet (FI) as a powerful network with heterogeneous technologies, low expectation of Quality of Service (QoS)/Quality of Experience (QoE), and evolving business models. All these characteristics combine to create a highly complex network and service management environment based on business objectives, which cannot be handled by traditional network management and thus creates a need for autonomic management behavior. A policy management framework was specified to overcome the challenges of the highly diverse, decentralized, and dynamic Future Internet.

VanderHorn et al. [31] introduced the Cognitive Network Management System (CNMS). CNMS is a research initiative for complex Mobile Ad hoc Networks (MANETs). It provides a real-time policy-based management framework that aims to mitigate the need for centralized network management, provide automated management by providing reasoning and enforcing mechanisms for network resources, reduce human intervention, and increase network reliability. The authors achieve these objectives by utilizing a lightweight policy-based framework, which is able to adapt at runtime to unpredictable network conditions by creating and enforcing new learned policies. A learned policy is a new policy created by a cognitive node to mitigate unpredictable network conditions. Learned policies can be distributed to other nodes to manage similar network conditions.

2.2 Policy-based management for WSN

Sensor nodes are designed to work in harsh and unrestricted environments for an extended period. Therefore, the cost of these sensors has to be low, which may restrict such capabilities as memory and computational power. Hence, sensors need to be updated from time to time due to resource constraints or changes in the operational environment. The conventional way to reprogram the sensors is to take the sensors from the field and reprogram them [1], [11], [12]. This approach has proven hectic and problematic. Another approach is to reprogram the sensors over the air by sending the new code through a transmission protocol. This approach has the disadvantage of depleting the sensor node energy. Finally, researchers have investigated policy-based management as an alternative way to reprogram and manage sensors.

Lee et al. [32] investigate different approaches to sensor node reprogramming. The two known methods for reprogramming are manual and over-the-air. In manual reprogramming, the sensor node code is updated through physical access to the node. This has proven to be tedious and time-consuming. In over-the-air reprogramming, the code is disseminated over the air to all sensor nodes in the WSN. The drawbacks of this method are network congestion and energy depletion. The large number of transmission activities creates network congestion, while energy depletion results from nodes receiving a large amount of network traffic to update their code.

Lee et al. [32] proposed a novel approach to managing the process of over-the-air reprogramming by categorizing the different possible cases of node reprogramming based on the node's execution characteristics. The proposed approach creates a profile (policies) for each case to reduce the negative impact on the WSN. The simulation results show impressive improvement over other reprogramming techniques, but this approach did not eliminate the negative impact of over-the-air reprogramming on the WSN, nor did it reduce energy depletion or the need for node reprogramming. Our work eliminates the negative impact of over-the-air reprogramming by reducing the need for this process through controlling the node behavior by policy programming, which requires significantly less transmission of data compared to full code reprogramming. Our work also reduces energy depletion by significantly reducing the transmitted data size.

Jacquot et al. [33] proposed a new approach to WSN management named LiveNCM, which stands for "LiveNode Noninvasive Context-aware and modular Management." It is a new approach to WSN management systems in which a configurable modular architecture is enabled to fit to an application and provide traditional administrative functionalities. In addition, it introduces two new concepts to WSN management. The first concept is noninvasive context awareness to deduce the network node status from current processing messages, which consequently reduces network traffic and energy consumption. The second concept is the estimator model, which is the possibility of computing some predictable values. Therefore, nodes can only send data outside the predicted range. In this way, the node will preserve energy and reduce the amount of transmitted data, as is demonstrated by some impressive simulation results in this work.

Zhang et al. [5] proposed a network management architecture as depicted in Figure 3. The proposed architecture is based on fault, configuration, accounting, performance, and security management components. The basic idea behind the proposed architecture is to form hierarchical clusters, which communicate with their cluster nodes and another superior sink node. Each node in the network is capable of performing cluster head as well as cluster child functionalities.

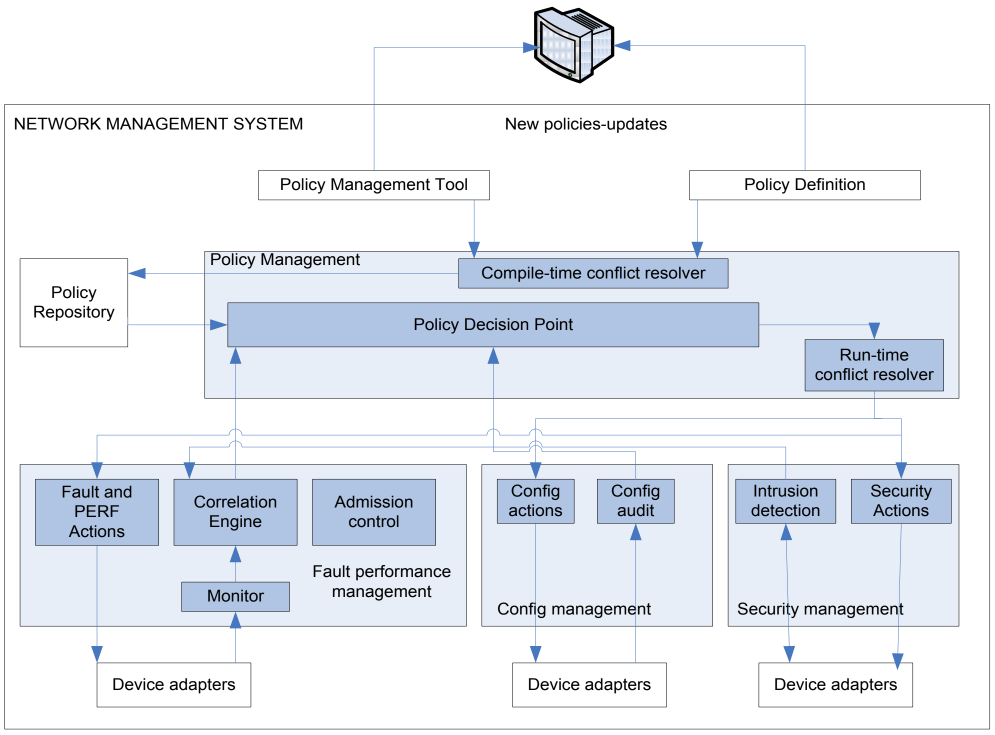


Figure 3 Policy-based management system architecture

There are many challenges associated with this architecture. First, forming and maintaining the cluster structure would pose a significant communication overhead on the network, due to the amount of information that must be exchanged between the cluster head and its children.

Second is the size of the software that the architecture is proposed to have on each sensor node, which is expected to be larger than the average sensor's memory capacity. As shown in Figure 3, the architecture is proposed to have the following software components, which are enormously larger than any other existing policy-based framework for WSNs:

1. Policy management component (Policy Decision Point (PDP), Compile-time conflict resolve, and Runtime conflict resolve)
2. Fault and Performance management
3. Configuration management

4. Security management

5. Quality of Service (QoS) management

Third is the policy repository. Given the predicted large size of the software that needs to reside on each sensor node, it is unlikely that a lot of memory would be left to store the policies, and thus our proposal for a dynamic and distributed repository becomes a necessity for such an architecture.

The fourth challenge is multiple policies execution. Zhang et al. [5] did not discuss this topic and provides no information on how the system would handle such an issue. Multiple policies execution is necessary in some cases where an event requires triggering multiple policies in sequence.

Fifth, the setup and administration of such an architecture would be a significant task and would require a highly skilled professional to set up and manage.

Bourdenas et al. [15] proposed a self-managed cell (SMC) framework for a WSN. The authors argued the need for self-managed architecture, which is due to the complexity of sensor network applications and the fact that users are not expected to have high technical skills. The authors came to this conclusion from the cases they investigated in their research, which ranged from health care to environmental monitoring applications.

Typically, sensor networks are structured in three distinct layers as shown in Figure 4. The bottom layer is sensing, where actual sensing events are captured; the middle layer is analysis, where sensing events are processed for making decisions; the upper layer is dissemination, where collaboration with other network resources takes place. The other part shown in Figure 4 is the self-healing extension proposed by the authors.

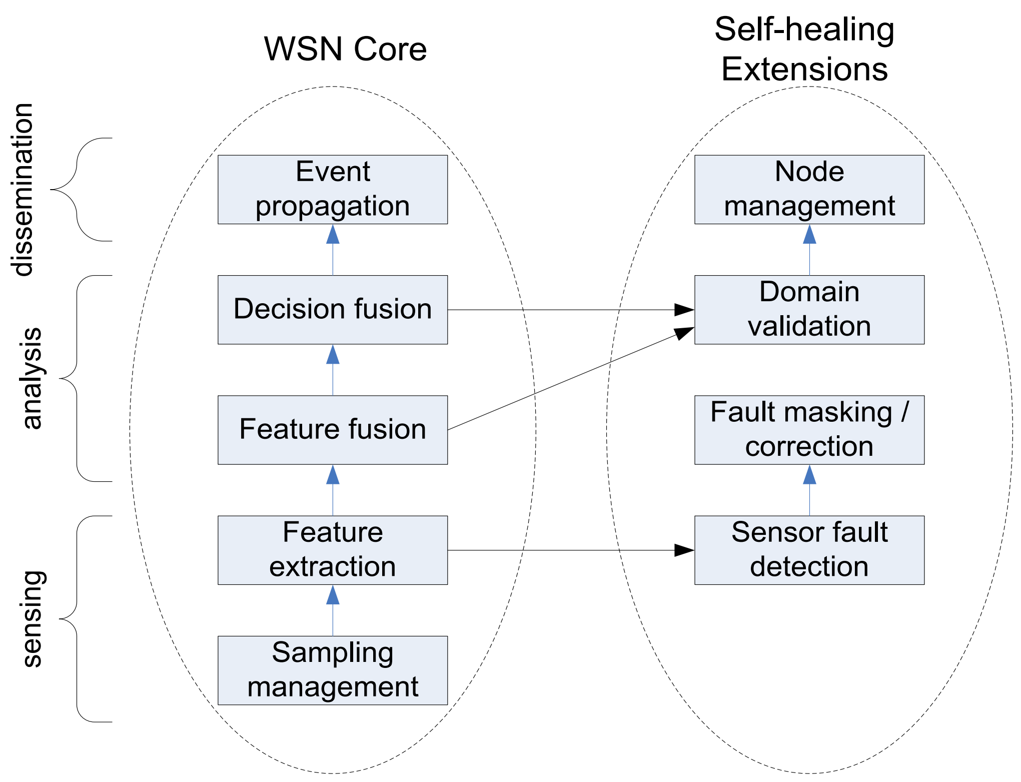


Figure 4 Layered functional architecture of WSNs

Figure 5 depicts the proposed SMC architecture, with the gray boxes representing self-healing services and the white boxes representing the core SMC services.

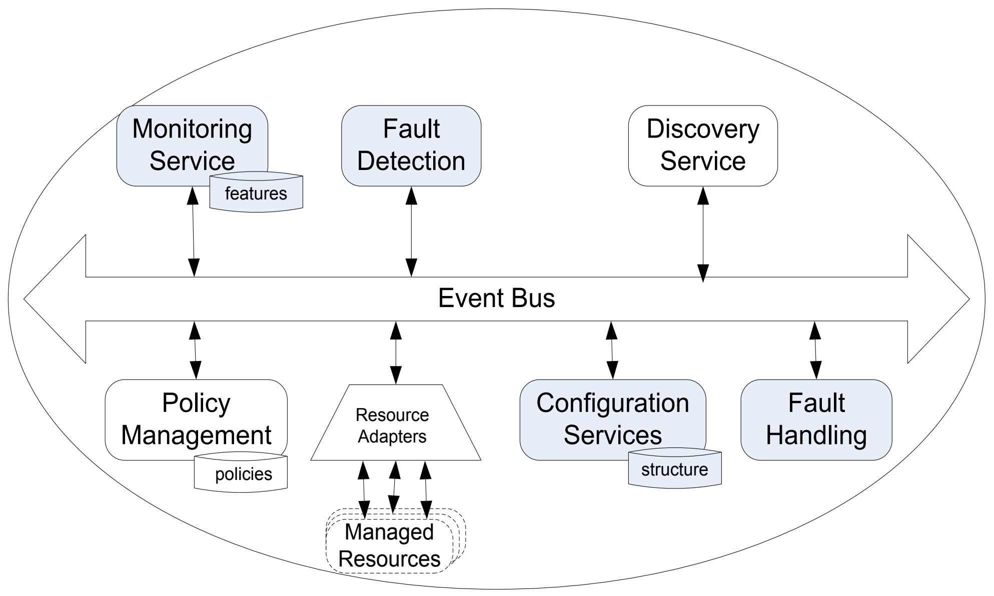


Figure 5 A Self-managed cell with self-healing services (gray boxes)

Policies are the means to control the behavior of the node. Bourdenas et al. proposed two types of policies. First is obligation policy: Event-Condition-Action (ECA) rules, which can express system behavior in an event-driven model. Second is authorization policy: controlling resource access or services by other nodes.

As shown in Figure 5, managed objects (nodes) are generating events, which can then communicate with the Policy Service through the Event Bus. Actions, on the other hand, are operations executed by managed objects, which also communicate through the Event Bus. To implement the proposed architecture, Bourdenas et al. developed the Starfish framework, which consists of the following components:

- ***Finger2***: An embedded policy system for sensor nodes.
- ***SML***: A module library to simplify the programming of sensor nodes. It provides basic functions and tools used in sensing applications. These include sensor sampling, feature extraction facilities, timers for scheduling of events, and network primitives for exchange of messages among nodes.
- ***Starfish editor***: A client-side graphical user interface for managing policies, missions, and roles on sensor nodes.

Figure 6 shows how the Finger2 architecture handles events as well as actions. The Authorization Manager checks the Event first to authenticate the source. After authenticating the source, the event is passed to the Obligation Manager/Event Manager, which searches the local repository for applicable policies. Applicable policies are then forwarded to the embedded Virtual Machine (VM) for execution. In some cases, the VM consults with the Authorization Manager to permit remote events triggered by the requested action.

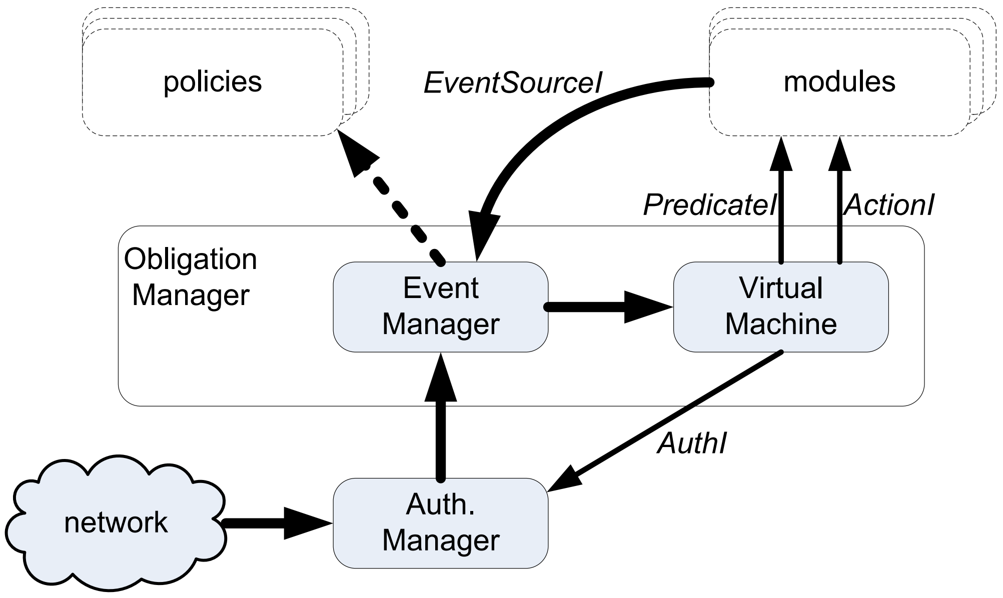


Figure 6 Finger2 architecture

Finger2 is the only policy engine for WSNs in the academic domain. Finger2 has been a basis for our work.

Zhu et al. [6] developed a simple TinyOS application, SimApp, making use of Finger. This application implements an event source of acceleration, and two actions, which toggle the red light and the green light. The application components consist of one obligation policy, which is the green light toggled when the acceleration is larger than a given threshold, and one authorization policy, which is controlling access to the red light action. The authors present their experimental results in [14]. Table 2 shows the experimental results for code size and Table 3 shows the processing delays of the experiment. These results are used as a benchmark for our work. The work done by Zhu et al. in [6] was studied as a guide to building our new framework environment, and its experimental results are contrasted with theirs.

Table 2 Code Size Breakdown of SimApp

Component	ROM (KB)	RAM (KB)
Finger(with authentication)	20.65	2.35
Finger (without authentication)	4.99	0.53
Comm.	8.08	0.49
Basics	2.55	0.04
Total (w/o)	15.62	1.06
Total (w)	31.28	2.88

Table 3 Processing Delays

Operation	Delay
Obligation Interp.	62 µs
Authorization Interp.	81 µs
Public Encrypt.	9530 ms
Public Decrypt.	5281 ms
Symmetric Encrypt	150 µs
Symmetric Decrypt	90 µs

2.3 Policy structure and protocol

A policy-based management system has to have a viable policy structure that can facilitate the management of sensors. Researchers have investigated the policy structure from various perspectives. Some researchers have studied the policy structure as a data entity, and others have investigated the need for a dedicated protocol to transport policies.

Ayari et al. [34] proposed a novel approach for Distributed Policy-Based Management in Mobile Ad-hoc Network (MANET). The proposed approach consists of three main parts: policy structure, policy-based framework, and Distributed Policy Management Protocol (DPMP). Policy structure contains the following segments:

Name (policy identification), Time (policy enforcement time), Group (one of four predefined policy groups), Role (an attribute that is used to select one or more policies), Scope (the policy target), On (trigger field for policy execution), If (policy condition of type Boolean), and Then/Do (task to execute).

The proposed protocol is vulnerable to deadlock and infinite circulation of messages in the network, as it is missing a time to live flag, which can be used to avoid such situations. The number of hops, which can be used to avoid sending messages to unwanted domains, is another piece of information that is missing from the proposed protocol.

In the policy structure, Ayari et al. did not discuss a case in which multiple policies need to be executed due to an event. In addition, it would be useful if the architecture had a field for the policy priority or execution sequence. Another issue concerns the purpose of the "enforcement time" field. It is not clear what they mean by policy enforcement time, since in practice it would be impossible to predict when the event would occur. Moreover, the length of the actual policy is too large to be applicable to wireless sensor networks or even to ad hoc networks. Ayari et al. also restricted the role of the Local Policy Decision Point (LPDP) to make local decisions, communicate with monitors, and interact with other LPDPs to distribute policies for non-configured nodes. This thesis expands the role of the LPDP to process and acquire the requested policies from remote nodes. (See Chapter 3.) Finally, Ayari et al. did not discuss the process of creating and administering the policies, which might be challenging and require human intervention. Their research was in a different domain than WSN, but it can be modified for the domain of WSN.

2.4 P2P algorithms in WSN

A fully distributed policy-based management approach was used to implement our framework. The use of hashing and P2P algorithms was fundamental. This section presents some prior research on P2P algorithms.

Thanh et al. [35] surveyed routing using distributed hash tables (DHTs), identified various algorithms, and compared them for energy efficiency, scalability, and data storage/lookup efficiency. Algorithms that could be used in our new proposed framework are Geographic hash table (GHT) [36], Chord for sensor networks (CSN) [37], Virtual Ring Routing (VRR) [38], Topology-based Distributed Hash Table (T-DHT) [39], Cell Hash Routing (CHR) [40], and ScatterPastry [41]. The authors concluded that ScatterPastry scored highest in all categories: scalability, energy efficiency, and data storage/lookup efficiency. The GHT, CSN, and VRR algorithms were on a par, followed by T-DHT and finally CHR.

Al Sukkar et al. [42] researched P2P systems in the domain of data-centric storage in a WSN. The authors proposed an algorithm for efficient data-centric storage in a WSN without the support of any physical location information system. The proposed algorithm supplies a unique temporary node address for every node in the WSN, based on its current relative location in the WSN. The node address will have a tree structure, where each node may have a parent and children.

The other part of their research was the routing algorithm, which works similarly to Pastry [43]. The routing algorithm requires each node to have information about the first hop neighbors and forwarding requests based on the longest node address matching the data object hash number. The work by Al Sukkar et al. [42] inspired our work in many ways, but it differs in several aspects as well. The first aspect is the problem that they were trying to solve. Al Sukkar et al. proposed a solution to resolve WSN content management, while our work tries to solve WSN network management. The second aspect involves their incorporating information about the relative (not physical) location of the sensor node in the address allocation, while our work incorporates a sensor's local information, such as the overlay address and Event ID number. The third aspect involves the routing algorithm. Al Sukkar et al. dictated a specific routing algorithm, while our work does not.

Gutierrez et al. [44] proposed to use a P2P network with a WSN to create a programming abstraction to ease the development of WSN applications. The abstraction relies on the feedback loop as a way to design the components of the abstraction and define their self-managing behavior. Feedback loops allow one to model different types of systems, especially self-managing systems. This type of system consists of the following four components:

- Subsystem: The main software component
- Monitor: A software agent that monitors the Subsystem
- Correcting agent: A software agent that receives information from the Monitor and decides on appropriate corrective actions
- Enforcement agent: A software agent that applies the corrective actions to the Subsystem

This research has demonstrated other benefits of using a P2P overlay network that simplifies software development for a WSN by abstracting the underlying network complexity. Some of the limitations in the existing works are: addressing a specific type of WSN as in [42], using arbitrary numbers for node or policy identification, limited the number of available policies to the node local repository capacity, relying on a human intervention in administrating policies in the system. On the other hand, this research addresses the WSN management in general and overcome all limitations mentioned earlier.

Chapter 3 TinyPolicy: A Distributed Policy Framework

In conducting this work, the existing policy-based management platform named Finger/Finger2 [6], [14], [15] was studied and used as a basis on which to build a new framework supporting distributed policy management. A fully distributed policy-based framework for WSNs was designed and built.

A framework for WSNs can be designed either with a central policy repository approach in which all nodes look up a policy in a Root node in the network, or with a fully distributed approach in which there are multiple repositories and copies of a policy in the WSN. The contrast between the two designs is summarized in Table 4.

Table 4 Centralized versus Distributed Policy Repository

Centralized policy repository	Fully distributed policy repository
Reliability: Less reliable; a node cannot get a policy from any other node	**Reliability**: More reliable; a node can get a policy from multiple sources (two to three sources)
Load Distribution: Policies are concentrated in the Root node. The more policies exist in the WSN, the more overhead the Root node will incur.	**Load Distribution**: Policies are uniformly distributed among all WSN nodes. Policy management overhead is distributed among different nodes.
Resilient: The loss of the Root node will disrupt system operation.	**Resilient**: The system will keep operating even with the loss of hosted nodes.
Performance: Unpredictable; all nodes have to get the policies from one particular node, no matter how far it is from the requesting nodes.	**Performance**: Predictable; through hashing function selection and adjustment, policy distribution can be controlled to store policies closer to their targeted node.

Node alive inspection: Not supported	**Node alive inspection:** Embedded ability to inspect for node alive status. (Responsiveness)

The architecture of the system was inspired by other work, notably Ayari et al. [34] This system architecture consists of four main components:

- Local policy repository for storing policies locally on the node
- LPDP (Local Policy Decision Point) for logical evaluation of the policies
- PEP (Policy Enforcement Point) for locally executing policies
- Monitor for tracking local and neighboring node information

Ayari restricted the role of the LPDP to making local decisions, communicating with the monitor, and interacting with other LPDPs to distribute policies for non-configured nodes. In our work, the architecture capabilities are expanded by using such mechanisms as Peer-to-Peer (P2P) communication, overlay network, tree-structure network, shared resources, and autonomic behavior.

Our framework consists of four main software components as shown in Figure 7. The main four software components are: Monitor, Local Policy Decision Point (LPDP), Policy Enforcement Point (PEP), and PolicyP2P. Moreover, the framework includes five data repositories (see section 3.2) to support system operations.

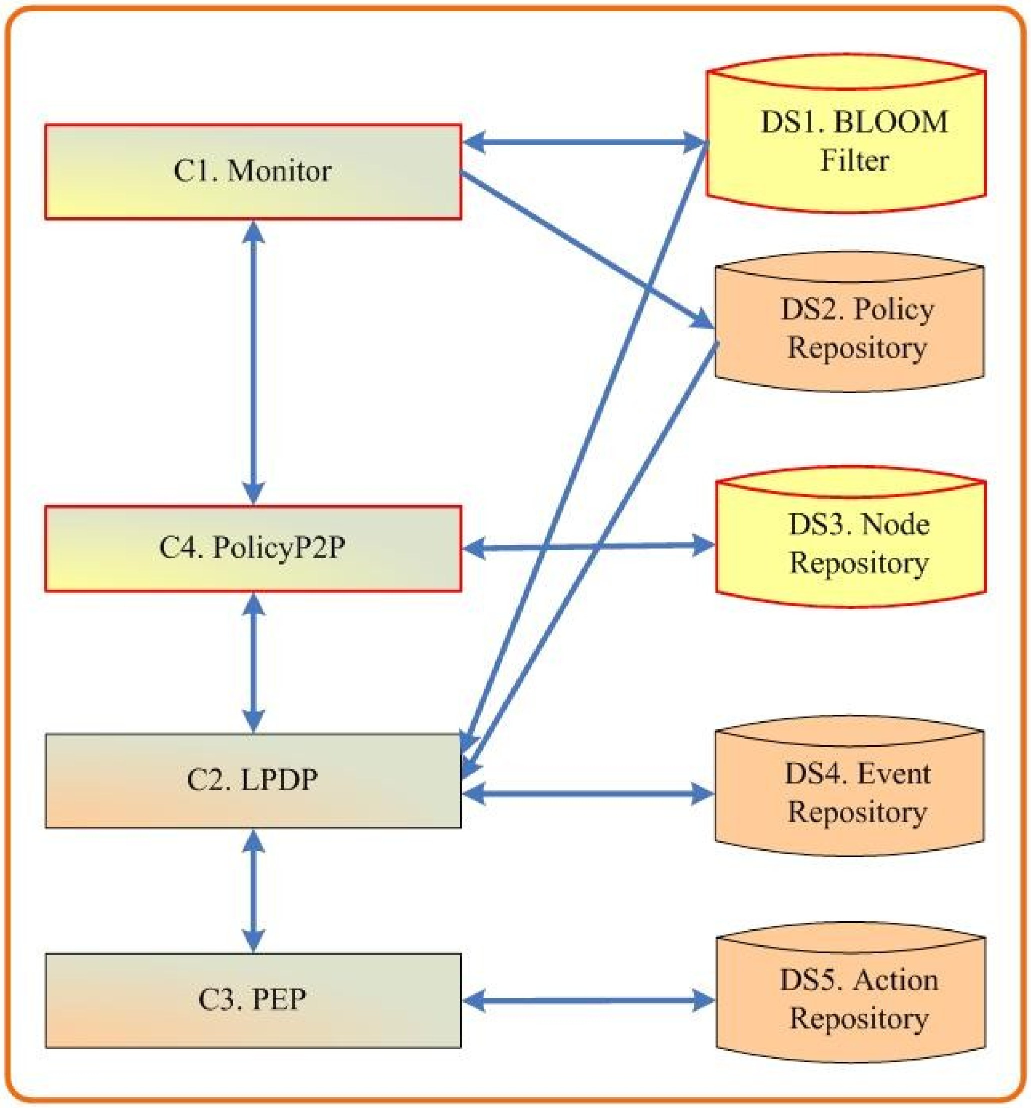


Figure 7 Distributed policy framework

3.1 Software components

As shown in Figure 7, the main software components of our framework are the following:

- ***C1. Monitor:*** Responsible for monitoring and updating Bloom filter values on the sensor network as well as on the local sensor node. The Monitor is also responsible for acquiring any necessary policy from any other remote sensor node, based on a request from PolicyP2P. The Monitor will also watch the most frequently used policies in the local sensor node and store them in the Local Policy Repository.

- ***C2. Local Policy Decision Point (LPDP):*** Responsible for making local decisions based on applicable policies, which are to be enforced by the Policy Enforcement Point (PEP). The decision made by the LPDP is based on policies stored in the local policy repository or acquired by the PolicyP2P component. LPDP will first try to get the policy from the local policy repository. If the policy does not exist there, LPDP will check the Bloom filter to validate the existence of the policy within the sensor network. LPDP will then decide whether to pass the request to PolicyP2P or declare the policy does not exist.

- ***C3. Policy Enforcement Point (PEP):*** Responsible for enforcing the policy decision (Action) provided by LPDP.

- ***C4. PolicyP2P:*** Responsible for maintaining the location of different policies within the sensor network. When a particular policy does not exist in the local repository, the PolicyP2P will issue a request to the Monitor to acquire the targeted policy from a remote node.

3.2 Data repositories

Our framework includes five data repositories to support system operations, as shown in Figure 7. The data repositories are the following:

DS1. Bloom Filter: The main objective of the Bloom filter is to inquire whether an element is a member of a given set. The purpose of the Bloom filter is to provide assurance on whether a policy exists on the sensor network. This process prevents any unnecessary policy inquiry transactions on the sensor network, which results in faster decision processing and preservation of sensor node energy.

DS2. Policy Repository: A data structure to store policy content. The policy repository will have limited capacity and will be able to hold a predetermined number of policies. The Monitor will update the Policy Repository based on the discretion of PolicyP2P or by monitoring policy usage. The capacity of the repository is a design choice that can be changed during development, but it can be mathematically calculated as in (1), by dividing the available memory size after uploading the program to the sensor's node memory by the actual size of the policy.

$$available\ memory = sensor's\ total\ memory\ size - program\ size$$

$$maximum\ policy\ repository\ capacity = \frac{available\ memory}{policy\ size} \tag{1}$$

To illustrate the previous equation, a Mica or IRIS sensor is used in this example. The sensor device has a memory size of 128 kB, the policy size in this thesis framework is 29 bytes, and the TinyPolicy program size is 30 kB. Hence, the theoretical maximum repository capacity would be calculated as in (2). However, not all the available memory can be used for the policy repository; part of the available memory should be reserved for the storage of program and operating system variables.

$$128\ kB - 30\ kB = 98\ kB ==> \frac{98\ kB * 1024}{29\ B} = 3460\ Policies \tag{2}$$

DS3. Node repository: A data structure used by PolicyP2P to store nearby node overlay addresses. The PolicyP2P algorithm uses this table to forward the request within the sensor network.

DS4. Event List: A data structure to store all possible events for the local sensor node. It can be populated at compile time or at runtime.

DS5. Action List: A data structure to store all possible actions for the local sensor node. It can be populated at compile time or at runtime.

3.3 Modified policy structure

Our work employed the policy structure and protocol used by Finger/Finger2, but with some modifications. Our new framework requires some modifications, mainly to the policy structure: the policy key and some other fields required by the new framework, as shown in Figure 9. Figure 8 shows the policy structure used in Finger2, while Figure 9 shows the modified policy structure.

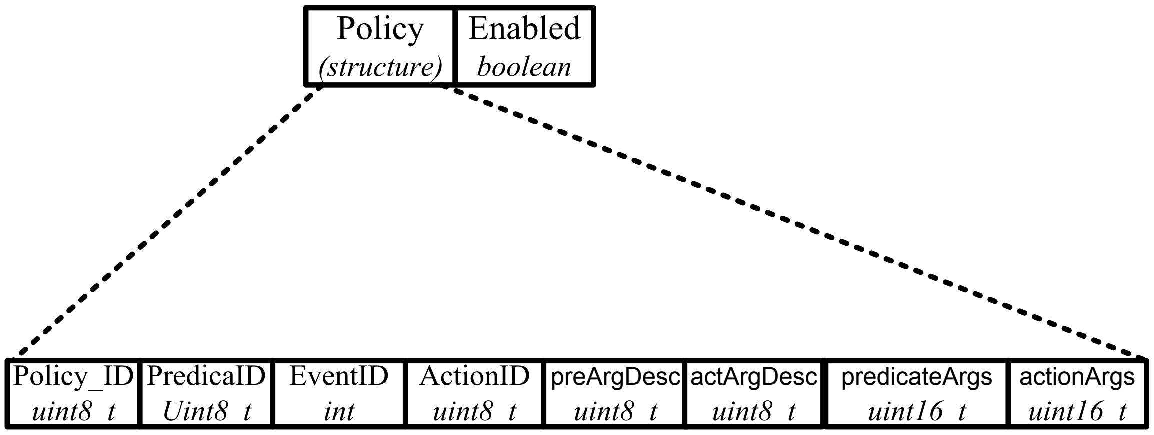


Figure 8 Finger2 policy structure

The main modifications to the policy structure involved the Type, Frequency, and Policy ID fields, as shown in Figure 9. The first modification to the structure added two new fields, Type and Frequency. These two new fields are very important for the policy retention algorithm, since it tracks the policy type and its frequency of use. (A more detailed discussion of the policy retention algorithm is in Chapter 4 Policy management in TinyPolicy.) The policy retention algorithm will use the Type field to distinguish between local policies (policies needed by a local sensor) and hosted policies (policies required by remote nodes). The second modification was the doubling in size of the Policy key (policy ID) field. This change was necessary for the PolicyP2P algorithm to work, as it needs the Policy key to be in the same number space as the Node ID. This is because Node ID is of type int16_t; hence, the Policy key has to be of the same type and size for the PolicyP2P algorithm to work. The PolicyP2P algorithm is discussed in detail in Chapter 6 PolicyP2P – A Policy Overlay Network.

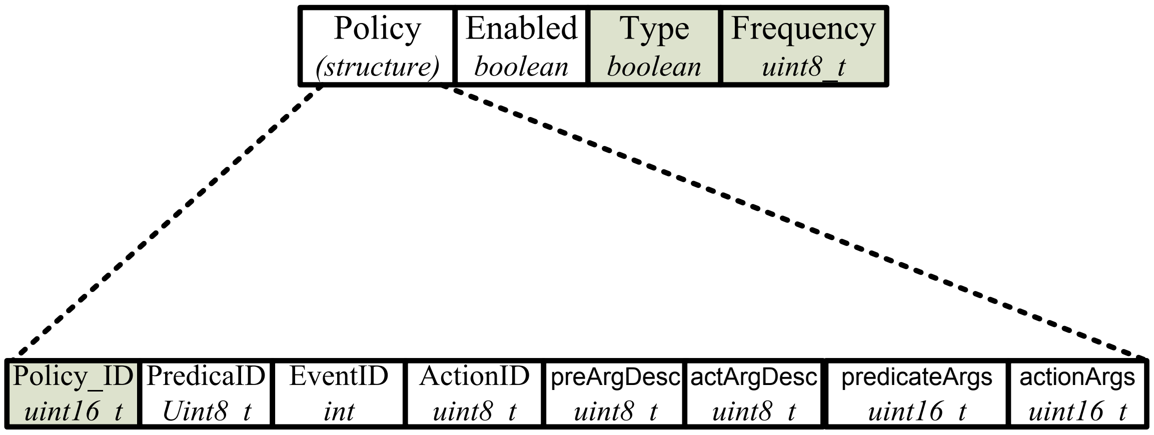
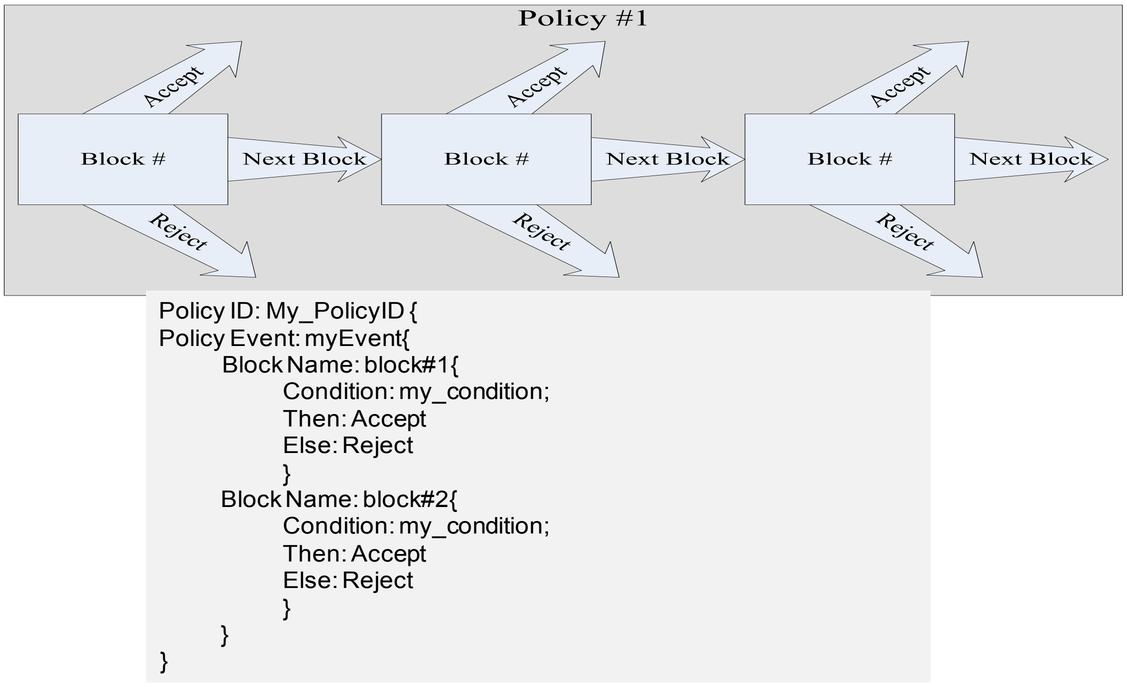


Figure 9 TinyPolicy policy structure

3.4 Multiple policies

The need to execute multiple policies per event is a major issue and can be resolved in different ways. The most common approaches to resolve the multiple policies issue employ a complex policy structure or policy chain. The difference between the two approaches is that the complex policy structure uses a compound policy structure to accommodate all required policies. In this approach, the multiple policies structure is actually a repetitive structure of a single policy structure but with different labels. On the other hand, the policy chain is a daisy chain of single policy structures, each with an extra field for the address of the next policy in the chain.

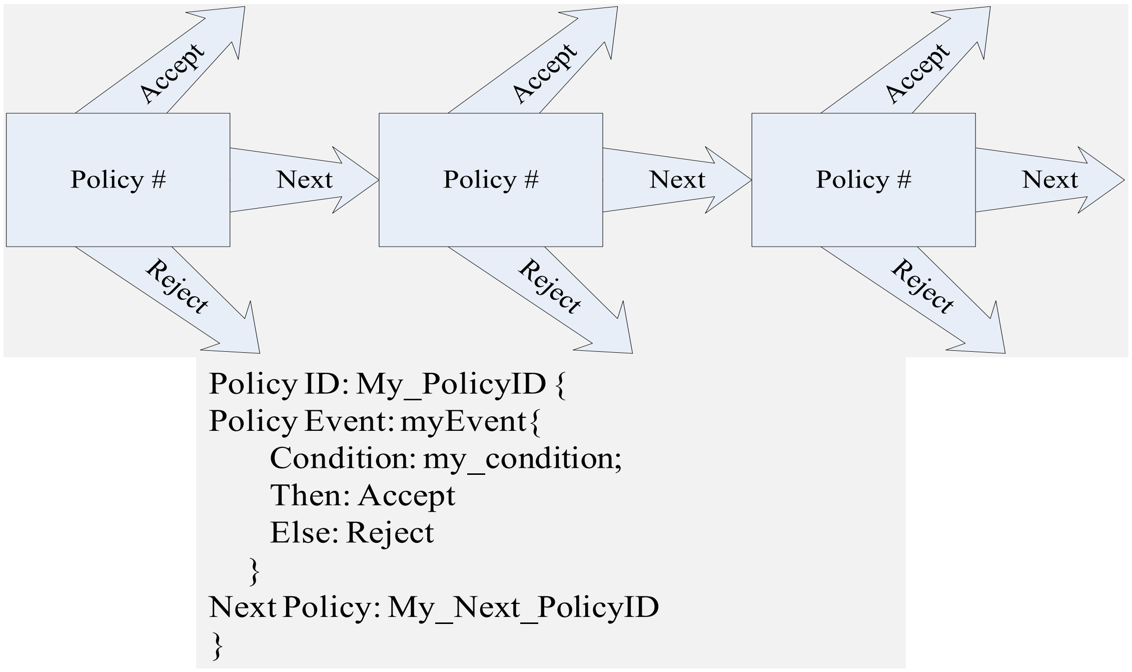
Complex policy structure: In this approach, the policy structure consists of more than one simple term (policy condition) as shown in Figure 10. The policy framework needs to accommodate this change by modifying its process execution accordingly.



```
Policy ID: My_PolicyID {
Policy Event: myEvent{
        Block Name: block#1{
                Condition: my_condition;
                Then: Accept
                Else: Reject
                }
        Block Name: block#2{
                Condition: my_condition;
                Then: Accept
                Else: Reject
                }
        }
}
```

Figure 10 Complex policy structure

Policy chain: In this approach, multiple policies are connected together in a daisy chain called a policy chain, as shown in Figure 11.



```
Policy ID: My_PolicyID {
Policy Event: myEvent{
        Condition: my_condition;
        Then: Accept
        Else: Reject
    }
Next Policy: My_Next_PolicyID
}
```

Figure 11 Policy chain

Chapter 4 Policy management in TinyPolicy

The following algorithms were created to support policy management in this thesis framework:

- **Policy creation:** Defines the steps for new policy creation and storage. The flowchart for policy creation is shown in Figure 12.
- **Policy modification and deletion:** Defines the steps for modification or deletion of a policy. The flowchart for policy modification and deletion is shown in Figure 14.
- **Policy execution:** Defines the steps for policy execution. The flowchart for policy execution is shown in Figure 15.
- **Policy retention:** Defines the steps required to retain or recycle the unwanted policies in the node repository. The flowchart for policy retention is shown in Figure 16.
- **Multiple policies:** Defines the steps required to execute multiple policies for a single event.

In order to manage policy operations, this thesis framework uses network message number 0x28. This message has a parameter specifying the policy's transaction type. Table 5 lists the possible values for this parameter.

Table 5 Policy Management Messages

Message Name	Description
LOAD_POLICY	***Load policy***: Issued by the Root to load a policy
REMOVE_POLICY	***Remove policy***: Issued by the Root to remove a policy

ENABLE_POLICY	***Enable policy***: Issued by the Root to enable a policy
DISABLE_POLICY	***Disable policy***: Issued by the Root to disable a policy
GET_POLICY	***Get policy***: Issued by any node to request a policy
SEND_POLICY	***Send policy***: Issued by any node to send the requested policy
TRIGGER_EVENT	***Trigger event***: Issued by any node to trigger an event on any other node
RELOAD_POLICY	***Reload policy***: Issued by any parent node to forward a policy to one of its predecessors

The remaining sections of this chapter will discuss these algorithms in more detail.

4.1 Policy creation algorithm

The new policy creation process starts by using the policy management tool, Policy IDE, on a computer that is connected to the Root node. The user creates a policy through the GUI of Policy IDE, as shown in Figure 13 and discussed in detail in Appendix A Policy management tool (Policy IDE) interface. The steps for policy creation are illustrated in Figure 12. After the policy is created using Policy IDE, the node (Root) updates the local Bloom filter array and broadcasts the array to the rest of the WSN nodes. To store the newly created policy in the WSN, the Root uses the PolicyP2P software component to hash the policy ID and compute the remote target node address for the node that will host the new policy.

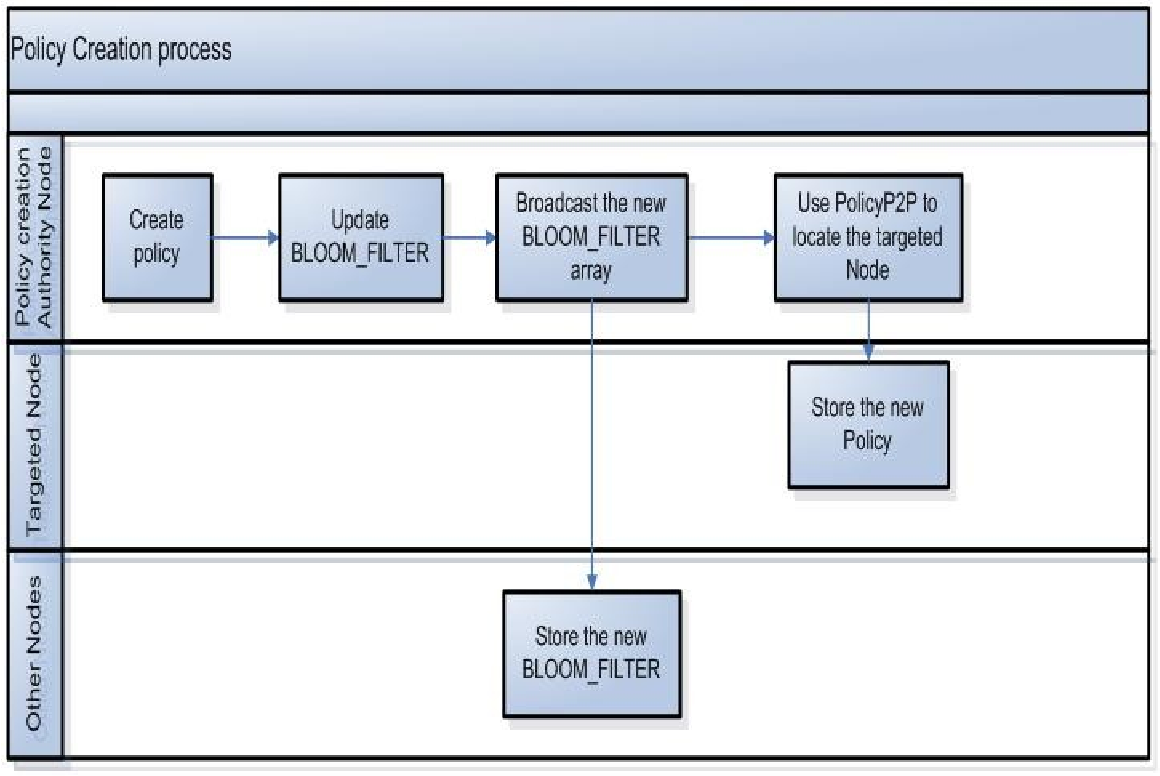
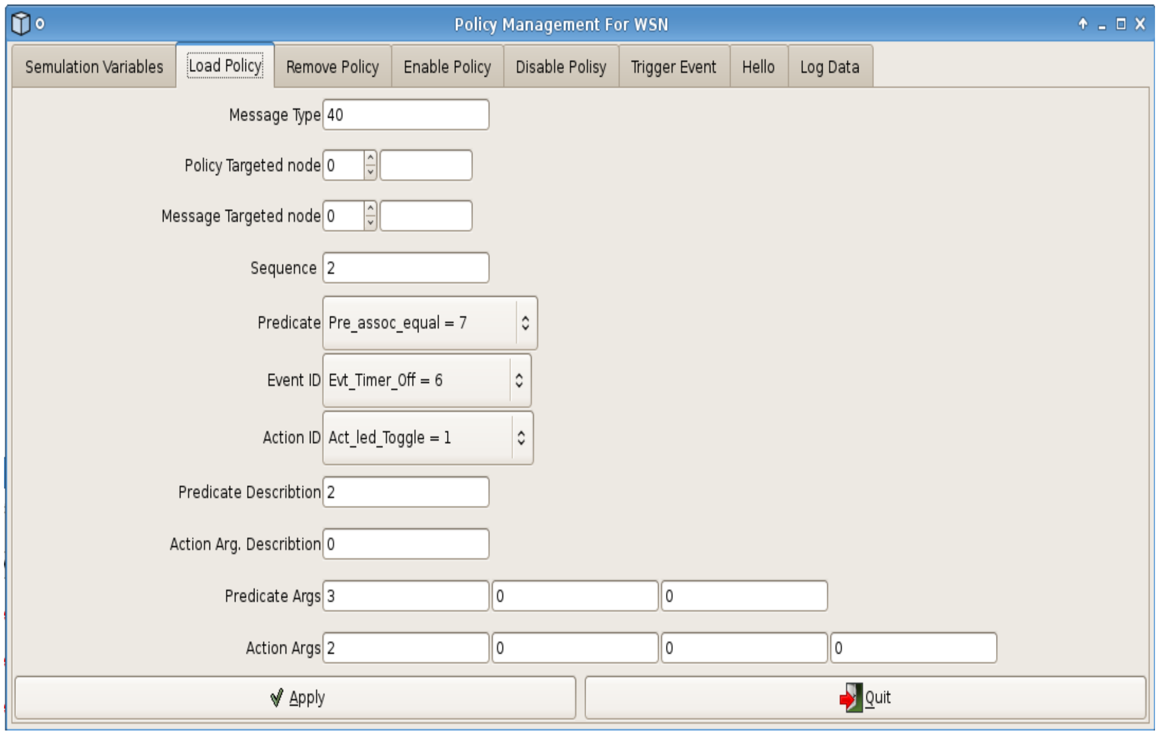

Policy Creation process
Policy creation Authority Node
Targeted Node
Other Nodes
Create policy
Update BLOOM_FILTER
Broadcast the new BLOOM_FILTER array
Use PolicyP2P to locate the targeted Node
Store the new Policy
Store the new BLOOM_FILTER

Figure 12 Policy Creation process



Message Type	40			
Policy Targeted node	0			
Message Targeted node	0			
Sequence	2			
Predicate	Pre_assoc_equal = 7			
Event ID	Evt_Timer_Off = 6			
Action ID	Act_led_Toggle = 1			
Predicate Describtion	2			
Action Arg. Describtion	0			
Predicate Args	3	0	0	
Action Args	2	0	0	0

Figure 13 Policy creation GUI

4.2 Policy modification and deletion

The policy modification and deletion process is illustrated in Figure 14. The process starts by checking if the policy exists in the local repository. If the policy does not exist in the local repository, the process is directed to the policy creation process as described in section 4.1. If the policy is an existing policy, the process checks the operation type. The operation type is either deletion or modification. If the operation type is deletion, the Root deletes the policy from the local repository and broadcasts the deletion request to the rest of the WSN; the other nodes then remove the targeted policy from their local repositories. The next step in policy deletion is to re-create the BLOOM_FILTER array based on the Root local policies remaining in the local policy repository. Finally, the Root broadcasts the new BLOOM_FILTER to the rest of the WSN nodes, which replace the old BLOOM_FILTER array on the other nodes.

For the policy modification process, there is no need to perform any changes on the BLOOM_FILTER array as this process intends to change only the policy content. Therefore, the policy creation authority (Root) retrieves the targeted policy from the Root's local repository, and the user can use a GUI similar to the one depicted in Figure 13 to modify the targeted policy. After the policy modification operation is completed, the Root broadcasts a deletion request to the other nodes, which remove the targeted policy from their local repositories. The purpose of broadcasting the deletion request is to make sure that only one version of the modified policy exists in the WSN. Finally, the Root sends the modified policy to the targeted node using PolicyP2P.

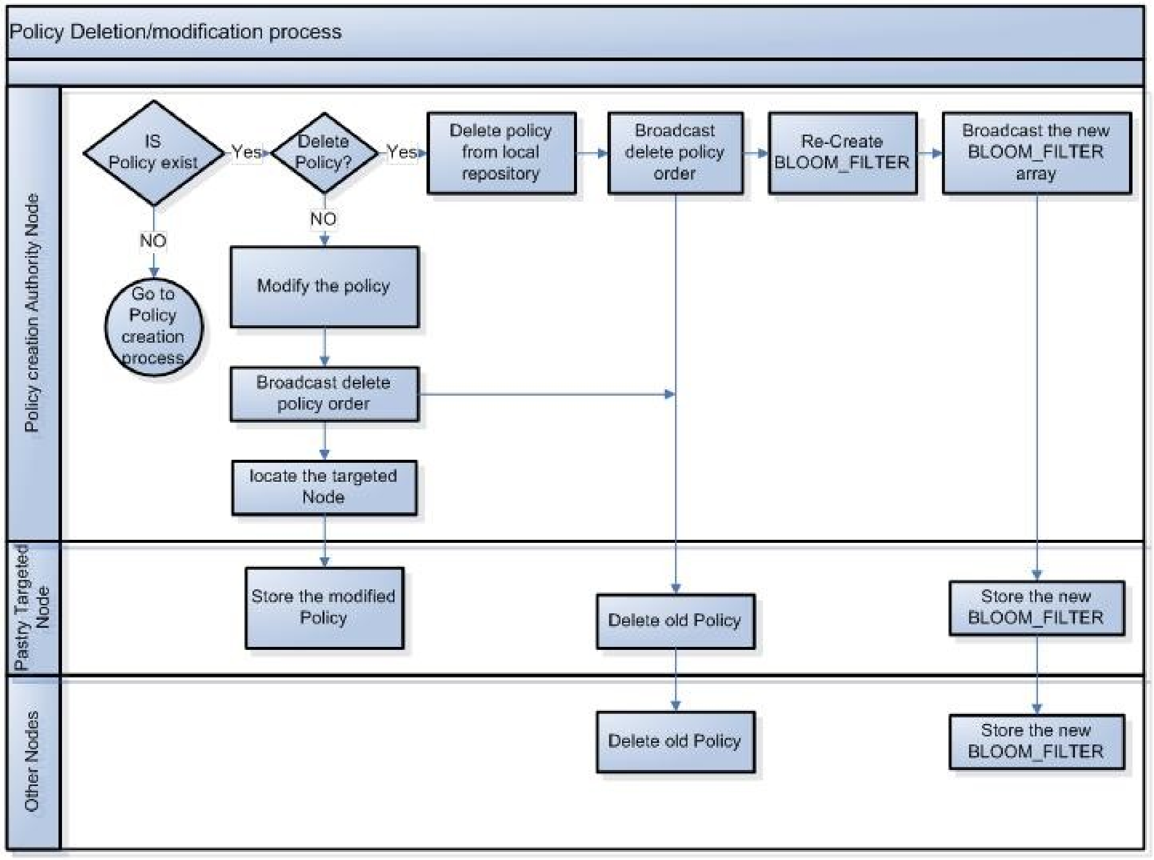

Policy Deletion/modification process
Policy creation Authority Node
Pastry Targeted Node
Other Nodes
IS Policy exist
Yes
NO
Go to Policy creation process
Delete Policy?
Yes
NO
Delete policy from local repository
Broadcast delete policy order
Re-Create BLOOM_FILTER
Broadcast the new BLOOM_FILTER array
Modify the policy
Broadcast delete policy order
locate the targeted Node
Store the modified Policy
Delete old Policy
Store the new BLOOM_FILTER
Delete old Policy
Store the new BLOOM_FILTER

Figure 14 Policy modification and deletion process

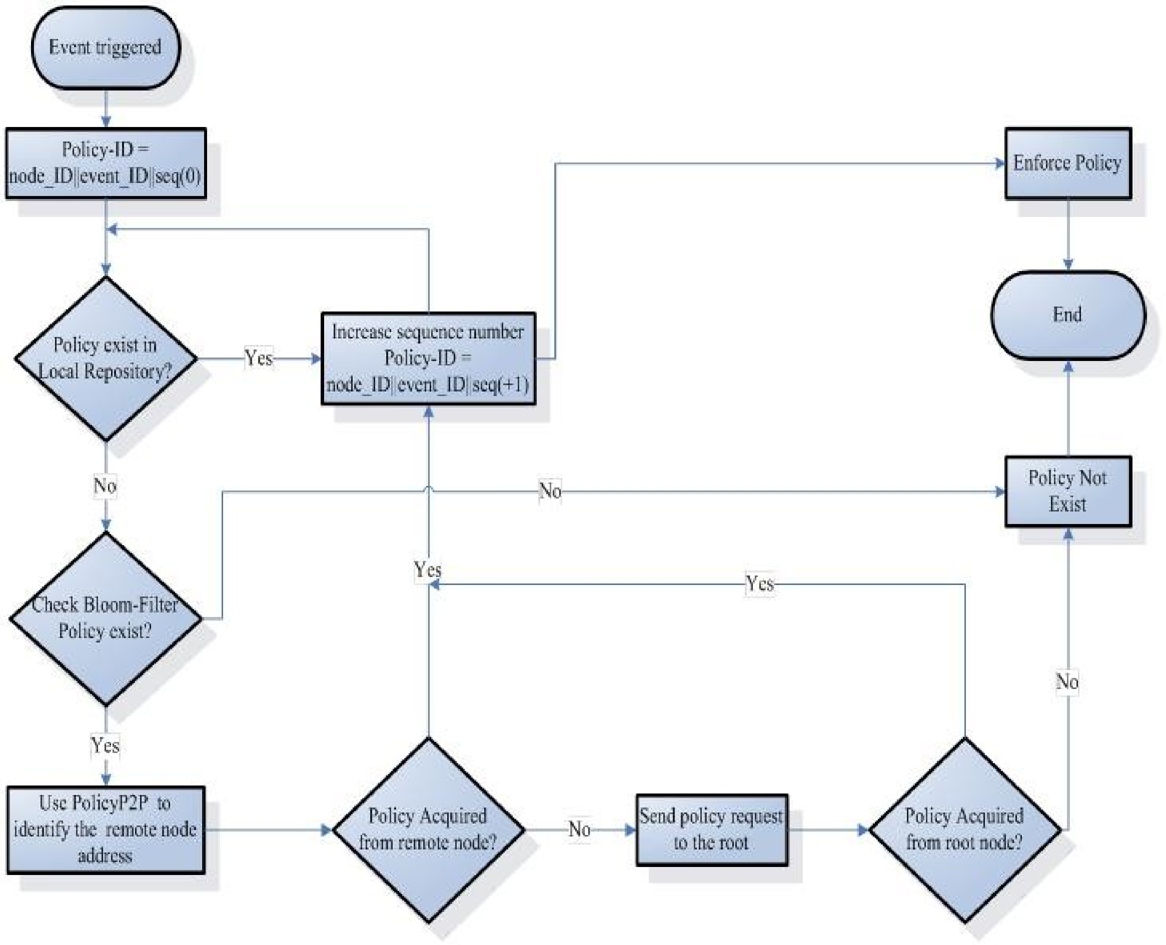
4.3 Policy execution

The policy execution process is shown in Figure 15. Each policy is associated with an event on the sensor node. The policy execution process of the associated policy starts when the sensor node triggers the associated event. First, the policy execution process constructs the policy key, which is the concatenation of Node ID, Event ID, and sequence number (sequence starts with 0) as shown in (7). The value of this concatenated data is then hashed using a proper hashing function. The generated hash value is the new policy key, which will be used throughout the rest of the algorithm. The algorithm then moves to check if the policy exists in the local policy repository. If the policy exists then two tasks are executed. The first task determines if there is more than one policy (multiple policies/chain of policies) associated with this event. The algorithm examines that by incrementing the sequence number and submitting a new task for policy execution with the new policy key. The second task enforces the policy by evaluating the condition in the policy and applying the required action if it is a valid policy.

If the policy is not found in the local policy repository, the process will check the BLOOM_FILTER to validate the existence of the policy within the WSN. If the BLOOM_FILTER test is negative then no further action is required and the execution is stopped. However, if the BLOOM_FILTER is positive then PolicyP2P calculates the remote node address, after which the Policy Execution Process sends a policy request to obtain the policy from the targeted node. If the targeted node provides the required policy then the process posts a new task for policy lookup with an increment to the sequence number to verify whether it is a single policy or multiple policies. After that, the algorithm enforces the acquired policy.

The targeted node could fail to provide the required policy for many different reasons: energy depletion, hardware error, communication error, or software error, just to name a few. In this case, the local node sends the request to the Root. If the Root provides the required policy then the same previous two tasks are executed. However, if the Root does not provide the required policy then the local node stops the execution and ends the process, because the policy does not exist.

As discussed previously, the local node might receive policies from remote nodes. In such cases, the local node would store the policies in the local node policy repository for future uses, based on the discretion of the policy-retention algorithm.



Event triggered
Policy-ID = node_ID||event_ID||seq(0)
Policy exist in Local Repository?
Yes
Increase sequence number Policy-ID = node_ID||event_ID||seq(+1)
Enforce Policy
End
No
Check Bloom-Filter Policy exist?
No
Policy Not Exist
Yes
Yes
No
Yes
Use PolicyP2P to identify the remote node address
Policy Acquired from remote node?
No
Send policy request to the root
Policy Acquired from root node?

Figure 15 Policy execution process

4.4 Policy retention algorithm

The purpose of this algorithm is to keep the frequently used policies in the local policy repository. Every time the node receives a request to load a policy, this algorithm is triggered to check if the repository is full. If the repository is not full then no action is necessary. However, if the policy repository is full then the algorithm searches for a foreign policy that has the lowest frequently used rate. (Foreign policy is defined as a policy that has been hosted in the current node based on the discretion of the PolicyP2P algorithm.) The targeted policy is then replaced with the new policy. Figure 16 shows the detailed steps of the algorithm.

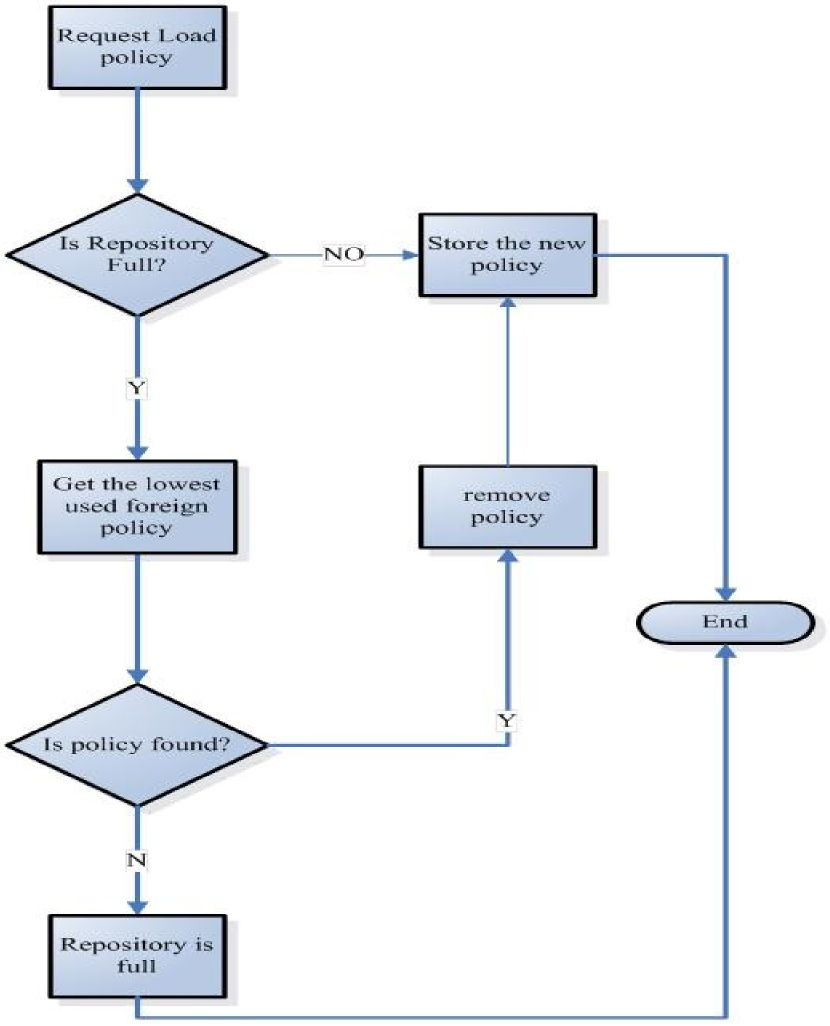


Figure 16 Policy retention process

4.5 Multiple policies

For this research, to resolve the need for multiple policies, the policy chain approach was chosen instead of the complex structure approach. As discussed in Chapter 6 PolicyP2P – A Policy Overlay Network, the policy key (policy ID) consists of three pieces of data as shown in Figure 19. For each triggered event, the node starts the policy execution with sequence number equal to zero; then it increments it by one until the BLOOM_FILTER test is negative, as shown in Figure 15. The node checks each new policy against its local policy repository. If the policy exists, the node executes it; otherwise, the node performs the BLOOM_FILTER test to save time and energy.

Chapter 5 Bloom filter

The policy structure may vary depending on the system and application requirements, but the most important part of any policy structure is the policy key (ID). The policy key plays a crucial role in any policy-based system, because it is used throughout the network to locate the targeted policy. For this reason, Chapter 6 PolicyP2P – A Policy Overlay Networkdiscusses a policy key that is based on the sensor's local data. This effective policy key is used by the Bloom filter to inquire about the existence of any policy within the network before wasting sensor node energy looking up a policy that may not exist in the network. It is possible to design the system without the Bloom filter. However, the contrast between the two designs is summarized in Table 6.

Table 6 Advantages of Using Bloom Filter

With Bloom filter	Without Bloom filter
Assurance: Provides assurance of policy existence	**Assurance**: Provides no assurance of policy existence
Lookup time: Policy is guaranteed to be found, so lookup time is not wasted	**Lookup time:** Policy is not guaranteed to be found, so lookup time may be wasted
Alert tool: A supported tool to alert the administrator about defective nodes	**Alert tool**: Cannot provide information about defective nodes.
Transmission time: Saving around 0.002 second (2000 μs) of transmission time per missing policy (more simulation data is in Appendix B Mathematical Model Data).	**Transmission time**: Wasting 0.002 second (2000 μs) of transmission time per missing policy (more simulation data is in Appendix B Mathematical Model Data).
Overhead time: Overhead time is computation time of 0.000126 s (126 μs),	**Overhead time**: No computation time overhead.

from the evaluation data shown in section 5.4.	

5.1 Bloom filter implementation

Bloom [45] introduced for the first time the concept of using a hashing function technique to trade-off between space and time with some allowable error. The Bloom Filter, as it was named later, is an elegant data structure that validates the existence of an object in the domain space with no false negatives and an acceptable rate of false positives. It has been widely used to resolve resource constraints in various knowledge areas, including distributed computing, distributed file systems, distributed learning, and distributed manufacturing. There are some implementations of Bloom filters in WSNs in content-based routing [46][47]. In addition, the Bloom filter has many other implementations in databases, computer networks, social networks, and cryptography. Our work implements the Bloom filter technique to inquire about the existence of any policy within the network before expending sensor node energy on looking up a policy that may not exist in the network. No changes to the Bloom filter algorithm were made. However, a significant analysis was performed to choose balanced parameters for the algorithm that are appropriate for the WSN environment.

Adam Kirsch et al. [48] researched the benefits of using fewer hashing functions to build the Bloom filter array. The authors proved formally that only two hashing functions are necessary to use the Bloom filter array without any loss in the asymptotic false positive probability. Their proposed method uses two hashing functions $h_1(x)$ and $h_2(x)$ to generate k number of new hashing functions in the form of

$g_i(x) = h_1(x) + ih_2(x)$, where i is between 0 and k - 1.

Due to resource constraints in the sensor node, the proposed method in [48] should prove valuable in WSNs.

Prosenjit Bose et al. [49] studied the false-positive rate in the Bloom filter analysis provided by Bloom [45]. The authors claim that Bloom's analysis is inaccurate, because it underestimates the false-positive rate. They provided a new analysis, but the difference in rates between the two analyses is negligible and applies only to certain specific cases.

5.2 Bloom filter analysis

In order to use the Bloom filter, it is necessary to determine the values of many inputs, such as the optimal filter array size, the ideal number of hashing functions, hashing function algorithms, and the acceptable maximum rate of false positives. The following analysis should answer these questions satisfactorily.

Consider a set $S = \{s_1, s_2, \cdots, s_n\}$ of *n* members, and an array $A = \{a_1, a_2, \cdots, a_m\}$ of m members (bits) with an initial value of zero for all members (bits). H is a set of independent hash functions $H = \{h_1, h_2, \cdots, h_k\}$, each with output range between 1 and m. For optimal results, *k* has to be calculated by the following formula [48]:

$$k = ln\, 2^{(m/n)}$$

To add member index *a* to the set *A*, each bit at positions $h_1(a), h_2(a), \dots, h_k(a)$ in array *A* is set to 1. Any bit may be set to 1 many times. To check for membership of item $b \in S$, all bits at positions $h_1(b), h_2(b), \dots, h_k(b)$ in array *A* have to be equal to 1. It is still possible that the conclusion is wrong (called a false positive), but the probability of the false positive can be controlled by selecting an optimal number of hashing functions as well as the size of the Bloom filter array. Thus, it is certainly true that $b \notin S$ if any bit of $h_1(b), h_2(b), \dots, h_k(b)$ in array *A* is equal to zero. This observation is true, because for the member to be a valid member, it has to set all applicable bits in the array to 1. If any bit is zero then it is not a valid member.

The Bloom filter promises to be an effective algorithm; however, it raises many questions, including the following:

- What is the optimal filter array size?
- What is the performance of the membership test?
- What is the acceptable maximum rate of false positives?

- What are the trade-offs in servicing the filter?
- What is the acceptable trade-off between the actual member lookup test and the membership test?

To decide on the optimal filter size, assume *n* keys have been added to the filter *F* with size m (bits) using k number of hash functions. Then the probability that a particular bit still has the value of zero is $(1-\frac{1}{m})^{kn}$. The probability of a false positive in this case is given in (3) [48].

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx (1-e^{kn/m})^{k} \tag{3}$$

Prosenjit Bose et al. [49] claimed that (3) is inaccurate and underestimates the false-positive rate, but the difference in rates between the two analyses is negligible and applies only to certain specific cases. (3) can be simplified to (4) as explained in [48].

$$k = ln\,2^{(m/n)} \tag{4}$$

It can be inferred that the optimal number of hash functions is $k = ln\,2^{(m/n)}$. Thus, the filter size m (bits) can be obtained using (5).

$$m = \frac{n\,k}{ln\,2} \tag{5}$$

5.3 Hashing algorithms

Due to hardware resource limitations, hashing algorithms in the sensor node need to be lightweight (code size and computation), independent, uniformly distributed, and to require minimal computational power. Our work adapts the proposed method in [48] which is based on selecting two hashing functions $h_1(x)$ and $h_2(x)$ as a base to generate k more new hashing functions in the form of (6), where i is between 0 and k - 1.

$$g_i(x) = h_1(x) + ih_2(x) \tag{6}$$

Moreover, section 5.4 shows that intersection of the false-positive probability curve with the hashing function line is between 1 and 2 for both test samples of sizes 1,024 and 18,000 members, which may support the finding of Kirsch et al. [48] However, this conclusion is derived only from visual inspection of the chart, which needs analysis and validation.

There are many known hashing functions. However, our work required a hashing function that is lightweight, independent, and uniformly distributed, requiring minimal computational power. For that purpose, potential hashing functions can be shortlisted as follows:

- ***Additive hash:*** The simplest hashing algorithm, with weak performance. The algorithm adds the values of the characters in a string.
- ***XOR hash:*** A simple algorithm, with less than average performance. The algorithm XORs the values of the characters in a string.
- ***Rotating hash:*** Similar to XOR hash but with multiple XOR operations. This algorithm has minimally acceptable performance.
- ***Bernstein hash[1]:*** The algorithm adds the characters of a string and multiplies the result by a constant value of 33. The performance results were not great, which led to the creation of a modified algorithm called Modify Bernstein. The new algorithm was the same, except it replaced the addition operation with XOR.

[1] This algorithm was created by Dan Bernstein.

- ***Shift-Add-XOR hash:*** A very efficient algorithm for all types of data. It is similar to rotating hash, except it replaces the multiplication with addition and chooses a different constant number for rotation. More detailed information about this algorithm can be found in [50].

- ***One-at-a-Time hash*[2]*:*** This algorithm performs very well. It consists of multiple shift, addition, and XOR operations.

- ***FNV series*[3]*:*** This algorithm is a series of XORs and multiplications. It has some weaknesses, such as collisions and sensitivity to zero values, which make it unsuitable as a cryptographic hash function.

Table 7, reproduced from [51] and [52], provides a comparison of some hashing algorithms. The size-1000 column represents the smallest hash table size greater than 1,000 entries. The Collision column represents the number of collisions that occurred when hashing 38,470 English words to 32-bit values. For this research, based on the results in Table 7, one-at-a-time and Shift-Add-XOR (similar to the rotating algorithm but with better performance) hashing algorithms were chosen for Bloom filter usage.

Table 7 Hashing Algorithms Comparison

Name	size-1000	Speed	Collision
Additive	1,009	5n+3	37,006
Rotating	1,009	6n+3	24
One-at-a-Time	1,024	$9n$ +9	0
Bernstein	1,024	$7n$ +3	4
Pearson	1,024	12n+5	0
CRC	1,024	9n+3	1
Generalized	1,024	9n+3	0
Universal	1,024	52n+3	0
Zobrist	1,024	10n+3	1
MD4	1,024	9.5n+230	1

[2] This algorithm was created by Bob Jenkins.
[3] FNV refers to the creators' names: Glenn Fowler, Landon Curt Noll, and Phong Vo.

5.4 Bloom filter evaluation

The Bloom filter plays a major role in the policy execution process. Without the Bloom filter, a sensor node would have no knowledge of which policies are available in the network. Before starting this experiment, it was necessary to define some necessary environment parameters: Bloom filter size, member's sample size, number of hashing functions, and the hashing algorithm. To find reasonable values for the Bloom filter size and the number of hashing functions, Bloom filter analysis was conducted as shown in Chapter 5 Bloom filter. Performance can be further enhanced by using the proposed method in [48] to apply more hashing functions to reduce the false positive probability.

To decide on the member's sample size, the assumption here is that a reasonable member's sample size is 1,024 members (policies), based on the fact that a conventional policy platform can accommodate up to 20 policies on each node. Therefore, 1,024 members (policies) divided by 20 policies/node equals about 51 nodes. That is considered a reasonable size for a wireless sensor network. At the other end of the spectrum, the assumption of having 18,000 members (policies) will translate to 900 nodes (18,000/20 = 900), which is considered the largest single wireless sensor network implemented to date.

Figure 17 shows the analysis of the sample size of 1,024 members. It shows that the intersection between the false positive curve and hashing functions number line lies between 1 and 2 hashing functions, with a probability of false positives between 0.2 and 0.4. The graph also shows that the Bloom filter array size is around 300 bytes.

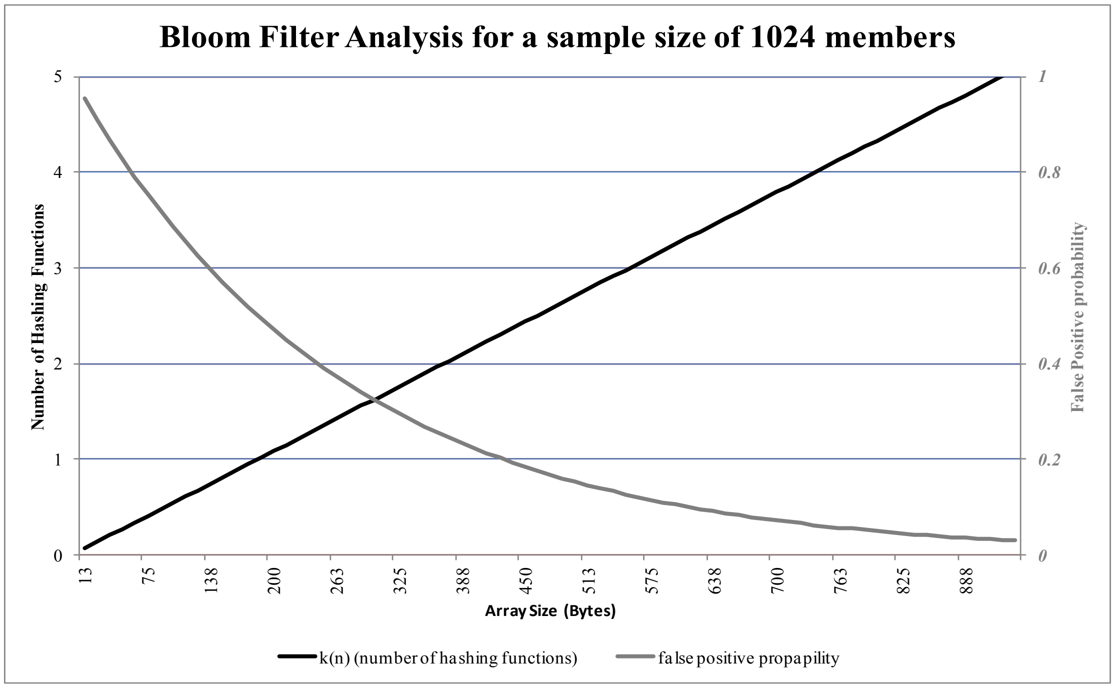


Figure 17 Bloom filter analysis for a sample size of 1,024 members

Figure 18 shows the analysis of the sample size of 18,000 members. It shows that the intersection between the false positive and hashing functions number line lies between 1 and 2 hashing functions, with a probability of false positives between 0.2 and 0.4. The graph also shows that the Bloom filter array size is around 5,225 bytes.

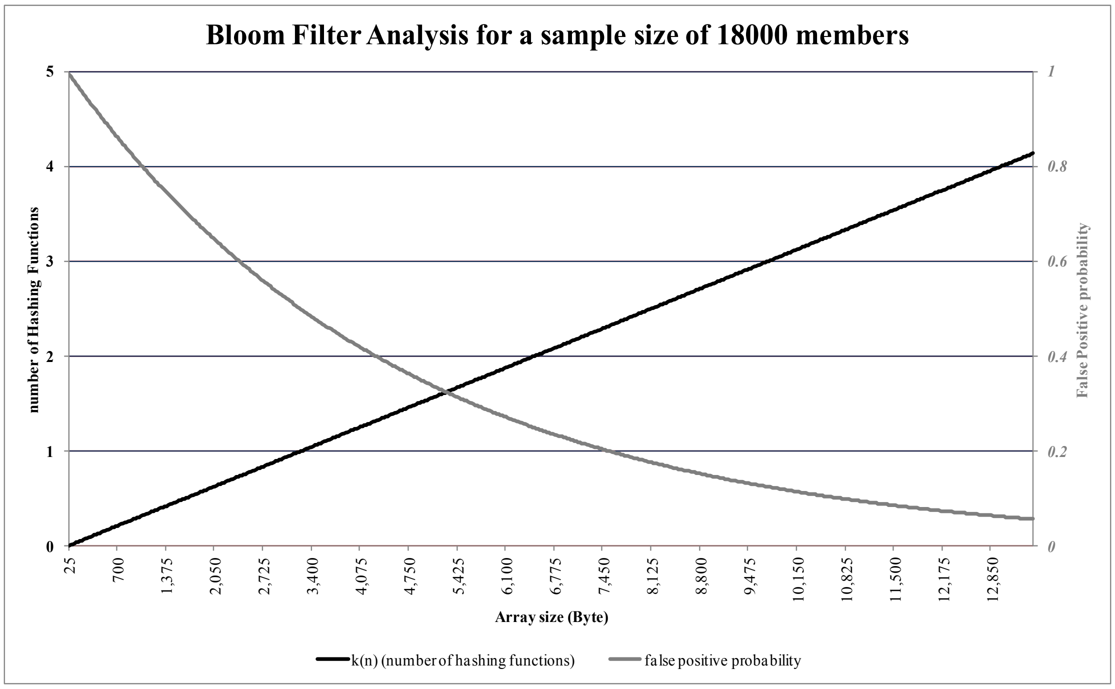


Figure 18 Bloom filter analysis for a sample size of 18,000 members

To conduct the simulation experiment, Tinyos-NesC [53] was used to code the hashing algorithm on the MicaZ platform. Avrora simulation software [54] was used to simulate the experiment. The other assumption here is that the policy ID consists of 36 characters ("0123456789abcdefghijklmnopqrstuvwxyz"). The experimental results for 1,024 members are shown in Table 8.

Table 8 Experimental Results for 1,024 Members

Hashing Algorithm Name	Time (µs)	Cycle	µJ/Cycle	Energy Consumption (µJ)	Members	Total Time (µs)	Energy Consumption (µJ)
One_At_a Time	51	176	0.0031	0.5419	1,024	52,224	554.8913
SAX	75	165	0.0031	0.5080	1,024	76,800	520.2106
Total	126	341	0.0031	1.0499	1,024	129,024	1,075.1020

The experimental results for 18,000 members are shown in Table 9. The resulting values in both tables include running the hashing algorithm and updating the Bloom filter. These two tables clearly show that the amount of resources the Bloom filter will need from a sensor node will be insignificant. For each lookup or update transaction, the sensor node will spend 126 µs and use 1.05 µJ of energy. In the first case of 1,024 members (policies), the total time needed is 129,024 µs, and the total energy consumed is 1,075.1 µJ.

Table 9 Experimental Results for 18,000 Members

Hashing Algorithm Name	Time (µs)	Cycle	µJ/Cycle	Energy Consumption (µJ)	Members	Total Time (µs)	Energy Consumption (µJ)
One_At_a Time	51	176	0.0031	0.5419	18,000	918,000	9,753.9492
SAX	75	165	0.0031	0.5080	18,000	1,350,000	9,144.3274
Total	126	341	0.0031	1.0499	18,000	2,268,000	18,898.2766

In the second case where 18,000 members (policies) were needed, the total time was 2,268,000 µs and the total energy consumption was 18,898.285 µJ.

The Bloom filter has been widely used in many application domains, especially in database management systems. This experiment shows how the Bloom filter can assist a policy-based management framework for a WSN to inspect the existence of a policy within the WSN with little computation time, minimal energy utilization, and limited traffic.

As shown earlier, each lookup or update transaction in the Bloom filter expends 126 µs and consumes 1.0499 µJ. It is known that each AA alkaline long-life battery produces 9,360 J. If each node has two such batteries then it can hypothetically execute (2 * 9360 J / 1.0499 µJ/transaction) ≈ 18 billion transactions. These numbers show that the additional overhead of the Bloom filter transactions on any sensor node will be insignificant.

Chapter 6 PolicyP2P – A Policy Overlay Network

This thesis uses the name PolicyP2P for the overlay network developed to support distributed policies in WSNs (or resource constraint devices). PolicyP2P is a collection of algorithms that are required by the overlay network component of TinyPolicy. PolicyP2P includes the following algorithms:

- ***Policy lookup and search:*** This algorithm defines the steps required to search and find any required policy.
- ***Network formation:*** This algorithm defines the steps required to build a new overlay network.
- ***Node joining the network:*** This algorithm defines the steps required to handle a new node joining the overlay network.
- ***Node leaving the network:*** This algorithm defines the steps required to handle an existing node leaving the overlay network.
- ***Network maintenance and recovery:*** Due to the nature of WSNs, a node may join or leave the network abruptly, which may disconnect the overlay tree structure and create orphan parents. This algorithm defines a mechanism to recover and maintain the healthy tree structure of the overlay network.
- ***Bloom Filter:*** A Bloom filter is a compact data structure used to support a decision-making process on membership of a data item in a set of data items. This work uses a Bloom filter to inquire about the existence of a given policy within the network before expending sensor node energy on looking up a policy that may not exist.

The PolicyP2P algorithm, which has been inspired by the Pastry algorithm [55], is an algorithm created to find the longest Node ID that matches the policy key. In other words, it makes a decision on which policy key belongs to which Node ID within the

WSN. When the policy does not exist in the local repository, PolicyP2P issues a request to the Monitor software component to acquire the targeted policy from a remote node. The only similarity between Pastry and PolicyP2P is in using the longest matching mechanism of the object hash code with the hosting node ID; no code, table structure, or other artifacts have been reused from any implementation of Pastry. The PolicyP2P algorithm builds an overlay network on top of the WSN as shown in Figure 22. The overlay network structure is in a form of a tree structure as shown in Figure 23. In order for PolicyP2P to operate, it uses the following network messages as shown in Table 10.

Table 10 Network Messages

Message Name	Description
AM_REQUEST_MSG	***Policy Request***: Issued by any node to request a policy transaction
AM_RESPONSEMSG	***Policy Response***: Issued by a targeted node in response to a policy request
AM_HELLO_MSG	***Hello Message***: Issued by a new node when it is joining the overlay network
AM_HELLO_RESP_MSG	***Hello-Response***: Issued by the parent node in response to a previously received HELLO message
AM_HELLO_ACKMSG	***Hello-Acknowledgment***: Issued by a newly joined node to confirm its new address
AM_REJOIN_MSG	***Rejoin***: Issued by a newly joined node to request all existing predecessor nodes to reconnect
AM_MAINT_MSG	***Maintenance***: Issued by the Root node to remove the defective node

	address from the parent node repository
AM_BF_MSG	***Bloom filter***: Issued by the Root node to send Bloom filter array

It is possible to design the WSN framework without an overlay network. However, the contrast between these two designs is summarized in Table 11.

Table 11 Overlay Network versus Physical Network

Overlay network	Without overlay network
Topology: Provides new information about network topology, as neighboring nodes are expected to be linked to each other, giving an abstract picture of the network topology.	**Topology:** Provides no information about network topology**.**
Content management: The overlay structure provides a new ability to control the flow of sensing data using policies. Using policies, sensing data may be directed to a target node that is closer to the source node.	**Content management:** Cannot be done without foreknowledge of nearby nodes.
Peer-to-Peer connection: The overlay network establishes a distance proximity relationship between nodes. Thus, nodes can communicate with each other in a meaningful context.	**Peer-to-Peer connection:** Nodes cannot communicate with each other in a meaningful context.

6.1 Determining policy key

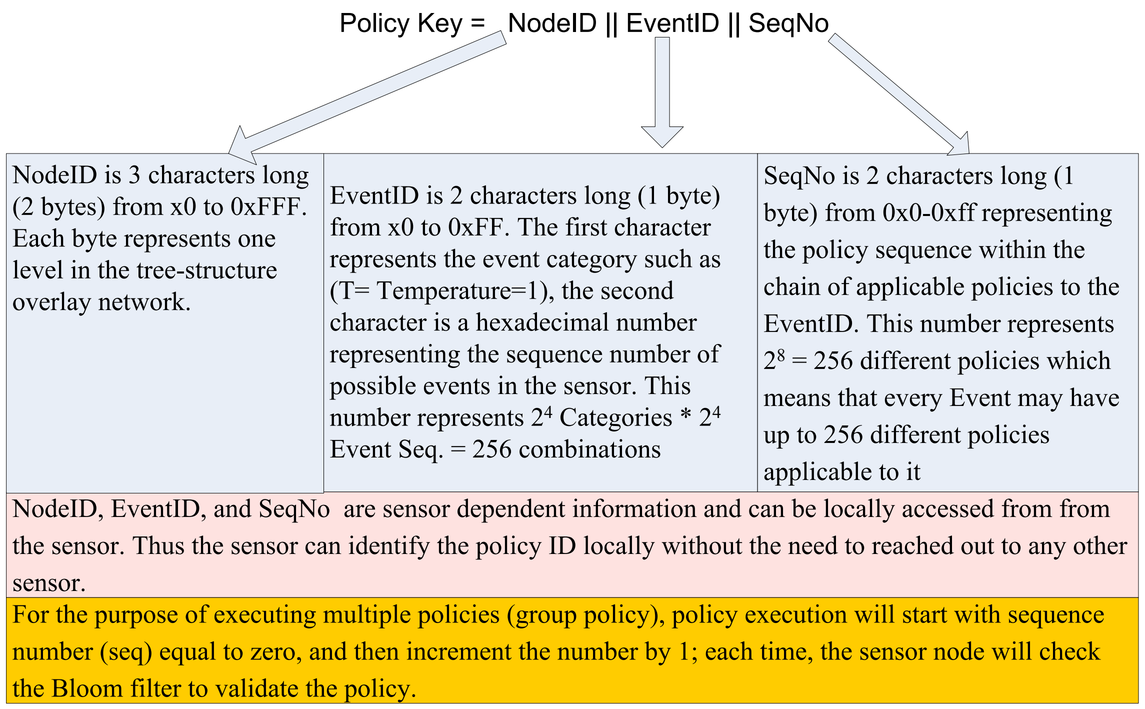
In many conventional policy-based systems, the policy key is an arbitrary number, devoid of meaning. It will not provide any helpful information to the user; on the contrary, it will add extra overhead to the process by requiring some kind of database to maintain the relationships between policy keys and applicable nodes, events, and should multiple policies be needed, the order of policies. In this thesis, the policy key is a system-generated number, which provides information about the targeted node address, event, and the order of policies in the policy chain.

The policy key plays a crucial role in our framework. The key indexing used for the policies is an important part of how PolicyP2P looks up the policy in a node's repository. The policy key also has implications for network traffic, because nodes will broadcast a message for each missing policy, which will generate unwanted traffic in the WSN. For this research, therefore, the policy key was built based on local data within the sensor node.

Thus, the policy key consists of three parts, which are Node ID, Event ID, and a sequence number. As shown in (7), these combined data are then hashed and the modulus of the largest possible node ID number is computed. The probability that a policy's hosted node will be identical to the targeted node depends on two issues: The strength of the hashing function and the size of the WSN, as fewer nodes would tend to increase this probability.

$$policy\ Key = H(nodeID \parallel eventID \parallel seqNum)\ \%\ Max(nodeID) \tag{7}$$

As shown in Figure 19, NodeID is matching the local node overlay ID number; Event ID is matching the Event ID value in the Event List data repository; and seqNum is the serial number of the policy, a value between 0 and 255. The first part of the policy key is the NodeID, which is two bytes long, similar to the local network NodeID number. The second part is the Event ID, which is one byte long. The first character represents the event category, and the second byte represents the event sequence number within the sensor node. Hence, the maximum number of event categories is $2^4 = 16$, and the total number of events per category is also $2^4 = 16$, and so the total number of possible event combinations is $(2^4 * 2^4) = 256$.


Policy Key = NodeID || EventID || SeqNo

NodeID is 3 characters long (2 bytes) from x0 to 0xFFF. Each byte represents one level in the tree-structure overlay network.

EventID is 2 characters long (1 byte) from x0 to 0xFF. The first character represents the event category such as (T= Temperature=1), the second character is a hexadecimal number representing the sequence number of possible events in the sensor. This number represents 2^4 Categories * 2^4 Event Seq. = 256 combinations

SeqNo is 2 characters long (1 byte) from 0x0-0xff representing the policy sequence within the chain of applicable policies to the EventID. This number represents 2^8 = 256 different policies which means that every Event may have up to 256 different policies applicable to it

NodeID, EventID, and SeqNo are sensor dependent information and can be locally accessed from from the sensor. Thus the sensor can identify the policy ID locally without the need to reached out to any other sensor.

For the purpose of executing multiple policies (group policy), policy execution will start with sequence number (seq) equal to zero, and then increment the number by 1; each time, the sensor node will check the Bloom filter to validate the policy.

Figure 19 Policy key

The third part of the policy key is seqNum, which is one byte long. seqNum represents the policy sequence number within the chain of applicable policies (event category). The total number of possible different policies is $2^8 = 256$; hence, every event may have up to 256 different policies applicable to it.

Based on policy key definition in this research, it should be no two policies with the same key and should be no one policy key applicable to more than one node. In some cases, it is possible to have one policy applicable to multiple nodes. The alternative solutions in this case is either to have a multiple copies of the same policy for each node or have a generic policy which applicable to multiple nodes. This research implements the first approach (multiple copies of the same policy) because the other alternative requires changes on the policy structure to store the applicable node addresses as well as creating a mechanism to be able to execute the generic policies which adding more complexity to the framework with little benefits in return.

6.2 Distributed Policy Addressing

Each policy in the system will have a policy key to facilitate the search and lookup operation in the system. The Root node is the only node that should create new policies. The node then uses a hashing function(s) to hash the policy key, which will have the same address space as the Node ID. Consequently, the node will forward the policy to the closest matching Node ID in the next level. If there is a closer matching Node ID in the lower level, then the node in the upper level will forward the policy to the other closest matching Node ID in the lower level. This process continues until there is no closest matching Node ID.

Figure 20 shows two policy storage examples for policy keys 0x1190 and 0x3119. The two policies are created by the Root node, and copies of them are forwarded to the closest matching node addresses in the Root node repository. In the first example, the policy key is 0x1190, and the closest matching Node ID in the network is 0x1100. Since the system is not centralized, the Root node has no knowledge of the existence of node 0x1100. Therefore, the Root node forwards the policy to the closest matching Node ID in its node repository (successor list). The closest matching Node ID in this case is 0x100. Afterward, Node ID 0x100 checks its successor list and forwards the policy to the closest matching node in its successor list, which is node 0x110.

The second example is for policy key 0x3119. Figure 20 shows that the closest Node ID is 0x3110. However, the system is not centralized, and Root has no knowledge of the existence of node 0x3110. Therefore, the Root node forwards the policy to the closest matching Node ID in its node repository (successor list). The closest matching Node ID is 0x300. Afterward, Node ID 0x300 checks its successor list and forwards the policy to the closest matching node in its successor list, node 0x3100. Node ID 0x3100 then checks its successor nodes list to find that Node ID 0x3110 has the Node ID with the closest match to policy key 0x3119. A copy of the policy is then forwarded to Node ID 0x3110 and saved in that node's local policy repository.

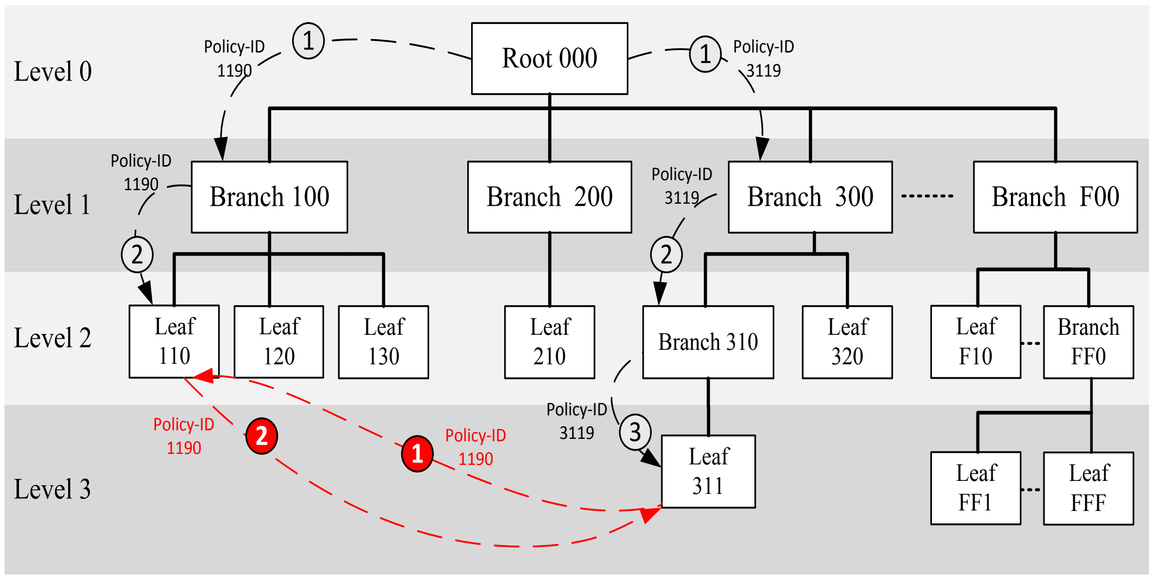


Figure 20 Policy storage examples

6.3 PolicyP2P algorithm

The input to the PolicyP2P algorithm is a hashed policy key as shown in Figure 21. The algorithm checks the leftmost hexadecimal digits against the corresponding digits in the Node ID. If they match, then the current node is the targeted Node ID, and the policy would be stored in the current Node ID or accessed from it. If there is no match, then the process checks the node repository to find if there is a matching node within the current node's children. If a match is found in the node repository then the current node sends the policy request to the remote node. If no match is found then the current node continues checking the leftmost length – 1 digits of the policy key with the current Node ID. If there is a match then the policy is stored in the current node. If there is no match then the process checks if the current address is the Root. If the current node is the Root, then the policy is stored in it; if not, the policy is not stored.

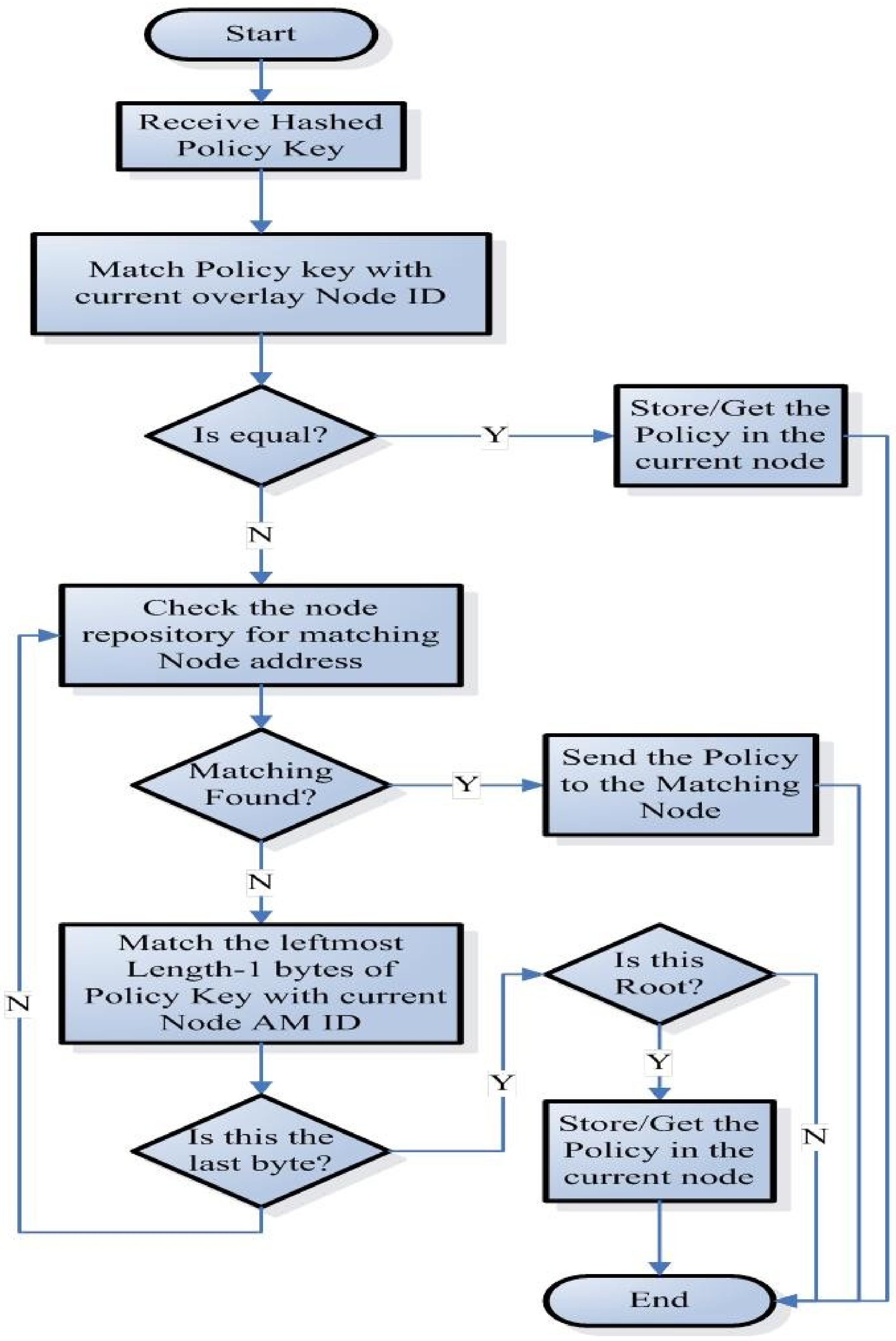

Start
Receive Hashed Policy Key
Match Policy key with current overlay Node ID
Is equal?
Y
Store/Get the Policy in the current node
N
Check the node repository for matching Node address
Matching Found?
Y
Send the Policy to the Matching Node
N
Match the leftmost Length-1 bytes of Policy Key with current Node AM ID
Is this Root?
Y
N
Y
Is this the last byte?
Store/Get the Policy in the current node
N
End

Figure 21 PolicyP2P

6.4 Policy lookup

Any node in the WSN can initiate a policy lookup request. The node that initiates the lookup request will hash the policy key and forward the request to the closest matching Node ID in level L - 1 (L is the targeted policy key level). Then the searching process starts from that level using the PolicyP2P algorithm. If for any reason the policy does not exist then a new policy request is sent to the Root by the initiating node.

Figure 20 shows a policy lookup example (dashed lines between node 311 and 110) for policy key 1190. Node ID 311 initiates the request and forwards it to Node ID 110, since it is the Node ID in level $L - 1$ that is closest to the requested policy key 1190. When the request reaches Node ID 110, the node checks its policy repository and sends the requested policy to Node ID 311. If Node ID 110 has a child with Node ID 119 and the policy does not exist on Node ID 110 then the lookup request will be forwarded to node 119.

6.5 Network formation

To implement a fully distributed system, the approach of this thesis is to build an overlay network on top of the WSN as shown in Figure 22. The overlay network structure is in the form of a tree structure with an implementation-specific number of levels. Level zero is at the top of the tree structure representing the Root node, while the lowest level is at the bottom of the tree structure representing the leaf nodes as shown in Figure 23. Any node will be able to communicate with any other node in the network; however, for a policy lookup transaction, the source node needs to send the request to a specific node (based on the policy key hash value) in level L - 1, where L is the targeted policy key level. The assumption here is that the number of available nodes will always be less than the maximum number of nodes that the network can accommodate. Therefore, the probability of finding a Node ID that matches a requested policy key is higher with a shorter address, and most likely, the parent node in level L - 1 will have a copy of the required policy.

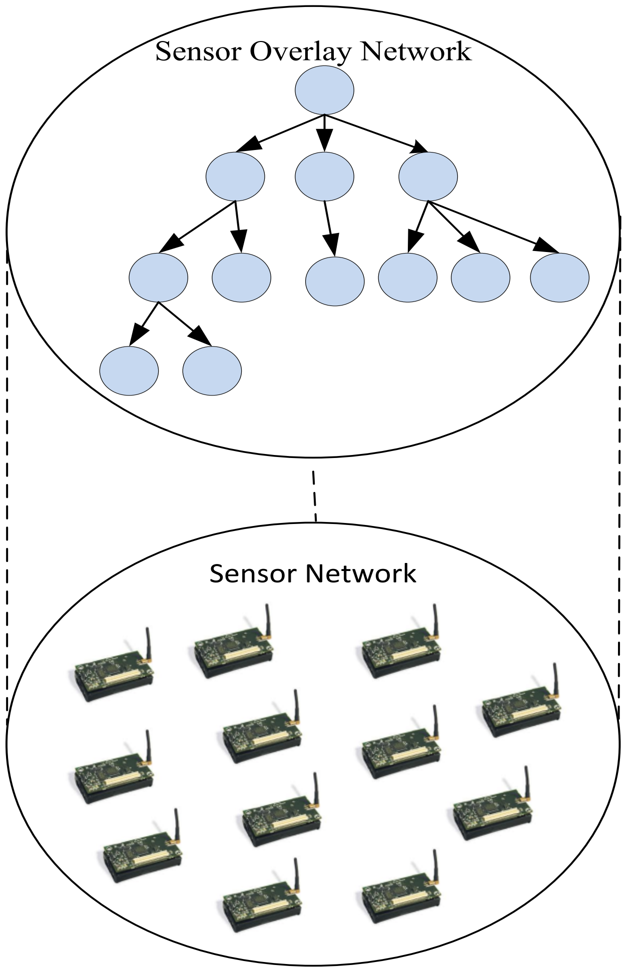


Figure 22 Overlay network for policy-based systems

As shown in Figure 23, at any given moment in the system's life cycle, each node is either a Root, Parent, or Leaf node. Root is the first node started in the WSN that has one or more successors but no predecessor, and there is only one Root node in the WSN at any given time. A Parent node has a predecessor and one or more successors. A Leaf node has a predecessor but no successors.

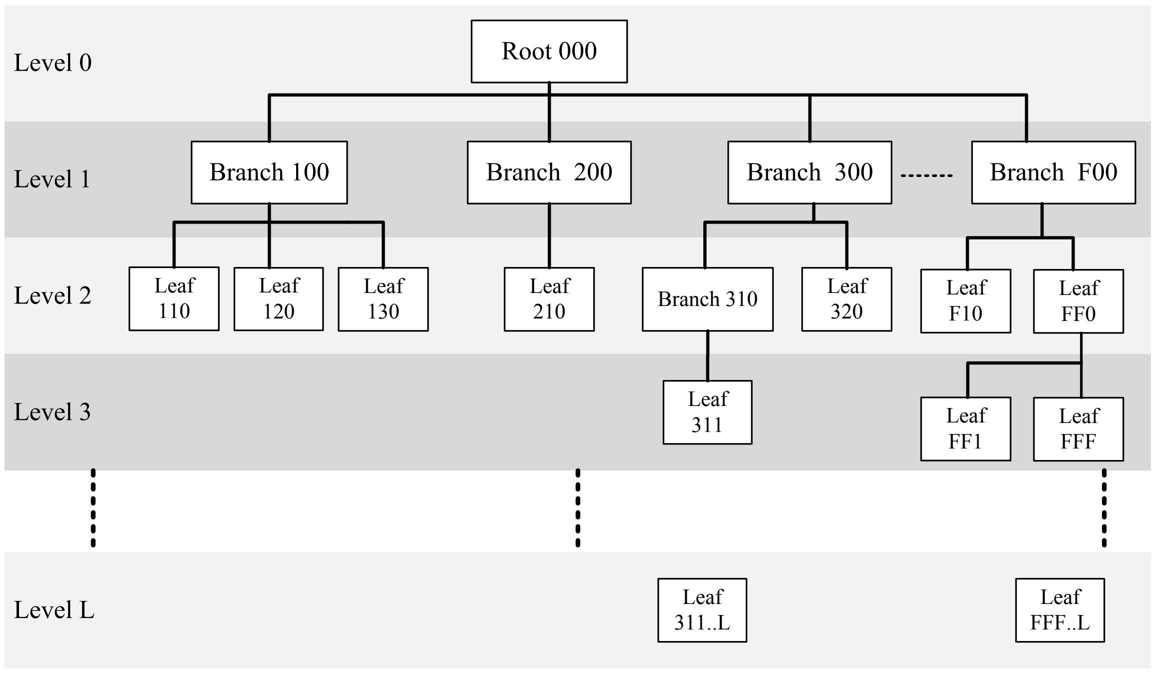


Figure 23 Tree structure for the overlay network

In our research, a three-level tree structure was chosen for the implementation. The reason is that it can accommodate up to 3,616 nodes as illustrated in Table 12 that is larger than the largest WSN that has been implemented to date of 900 nodes. Moreover, the largest number of policies in the system depends on the policy key, which should be in the same numbering space as the node ID. The network size in TinyPolicy framework is a design choice, which depends on the total number of levels in the network. Each level in the network can have 15^n nodes where n is the level number from 0 to n. the total number of nodes that can be accommodated in the network is calculated by adding up all nodes in all levels. Table 12 illustrates the calculation for a three level network, which has been implemented in this research.

Table 12 Overlay Network Size

	Number of nodes
Level 0 (Root node)	1
Level 1	15

Level 2 (15*15)	225
Level 3 (15*15)*15	3375
Total	**3616**

The overlay network starts being formed when the first node (Root) in WSN operates; then the tree structure starts being formed by each new node joining the WSN. Each new node starts its operation within the WSN by broadcasting a "Hello" message to all nodes in its range and requesting a Node ID for itself. All other (available) nodes respond by assigning and sending a new Node ID to the new node (successor). The new node accepts the first arriving Node ID and acknowledges the assigned Node ID to the originator (predecessor). The other nodes that send a Node ID to the new node will have the status of the previously given Node ID as "unconfirmed" in their node repositories and can reuse this address for other nodes in future requests. Figure 24 illustrates the message sequence for a new node joining the WSN.

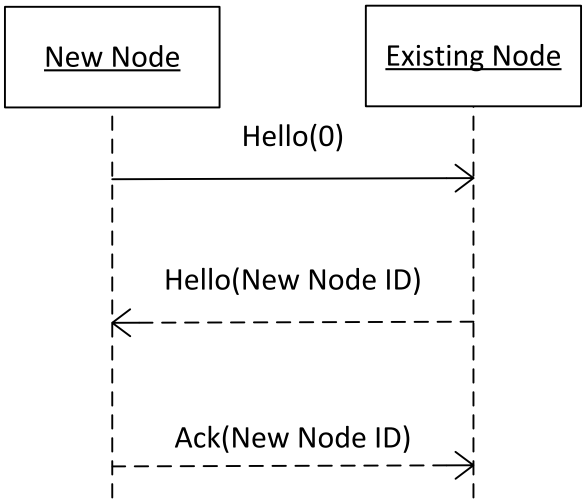


Figure 24 **Message sequence for a new node joining the WSN**

In this implementation, Node ID is a data field two bytes (16 bits) in length (0x0000 - 0xFFFF). The overlay network address uses only the first three characters (12 bits) for the Node ID. Each character in the Node ID address represents one level of the tree. As shown in Figure 23, Level 0 has only one node, which is the Root node with address 0x0000. Level 1 uses the first character from the left to represent nodes at that level. Therefore, the available address space for this level is from 0x1000 to 0xF000, which represents 15 available addresses. The second and third levels will use the second and third characters respectively. However, the third level nodes cannot generate any new Node IDs. Therefore, no new nodes can join the network through any of the third level nodes; the new nodes have to get their Node IDs from other nodes at higher levels.

6.6 Node joining the network

A new node joining an existing WSN has to broadcast a "hello" message to all nodes within its range. All nodes within the range respond with a newly generated Node ID for the newly joining node. The value of the newly generated Node ID is different, based on the parent tree level and parent Node ID. The new node overlay address is the first Node ID address received by the new node. Accordingly, the node that generated the Node ID address is the new node's predecessor. Consequently, the predecessor receives an acknowledgment of the overlay address from the new node and updates the status of the Node ID in its node repository to "confirm." Figure 25 shows the message sequence for a new node joining an existing WSN. Furthermore, the predecessor copies all related policies (based on the new Node ID) in its repository to the new node. As shown in Figure 25, the new node schedules a request to broadcast a rejoin message (after it confirms its new Node ID) to maintain the tree structure and to avoid keeping any orphan leaves or parents in the overlay network. Existing nodes that are within the address space of the new parent will acknowledge the rejoin message to the new parent node; accordingly, the new parent node will update its node repository with these new addresses.

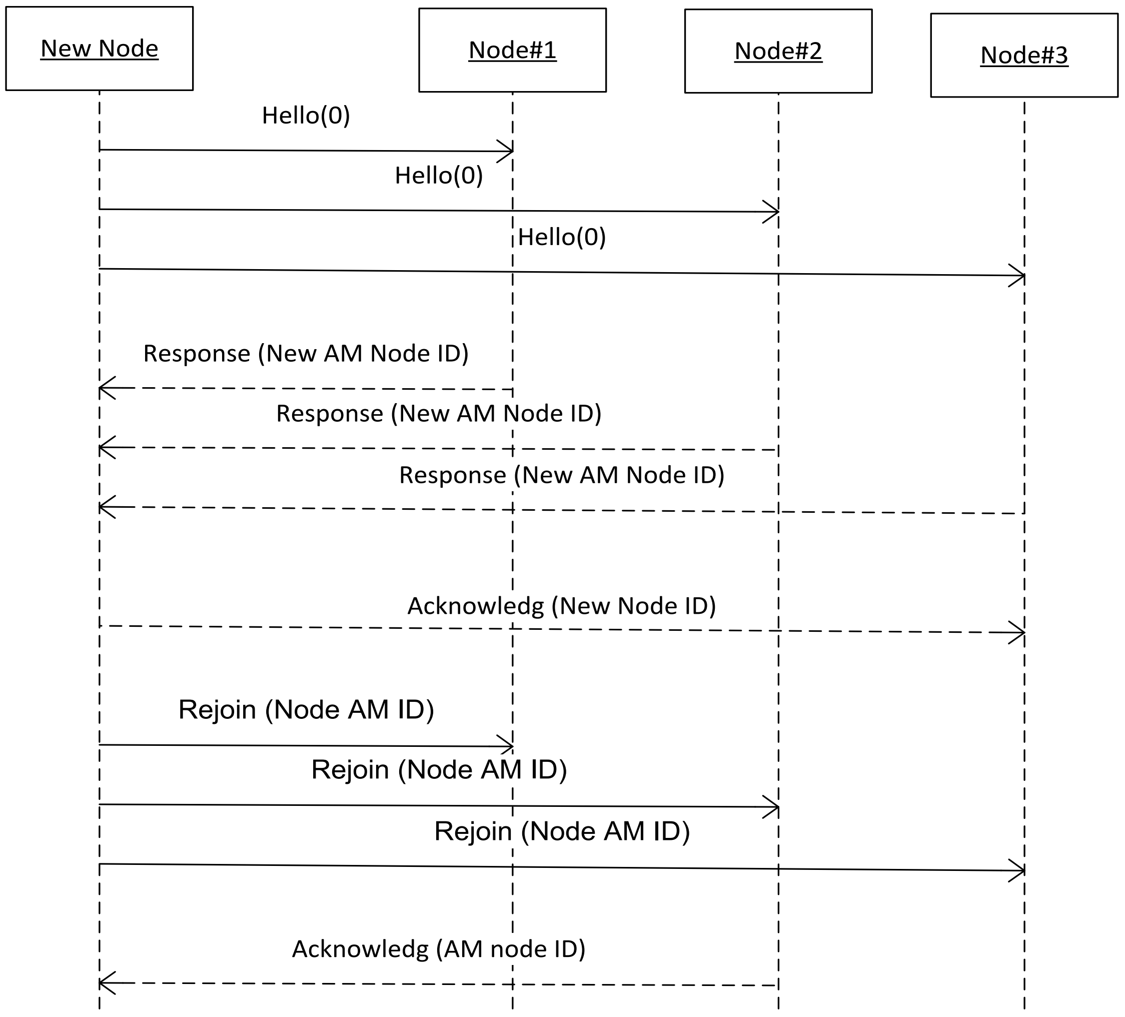
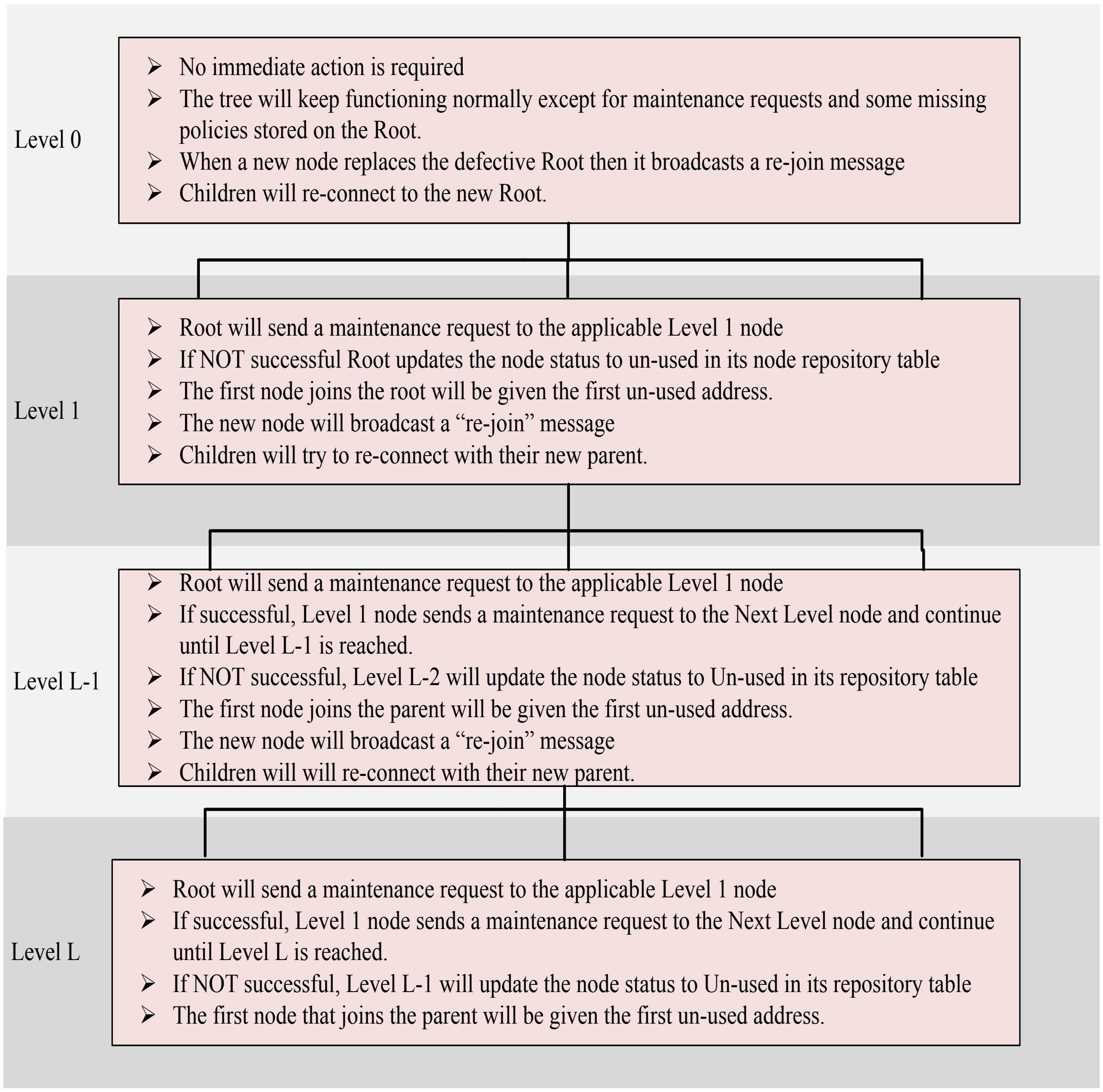


Figure 25 Message sequence for a join request to an existing WSN

6.7 Node leaving the network

A node may leave the network (overlay tree) deliberately or abruptly. This action has a small effect on the system. Only policies stored on the departed node or its successors will be partially unavailable, but the system will recover the missing policies from predecessor nodes or from the Root node, depending on the capacity of the affected nodes' policy repositories. When a node leaves the WSN for any reason, if that node has any successors then the subtree becomes an orphan tree. In this case, the system takes no immediate action. However, the Root node issues a maintenance request to maintain network reliability, with the first request to it to access any policy that should have been accessed from any other existing node in the network. Consequently, the Root node issues a maintenance request to its child that is responsible for hosting the affected policy. The maintenance request will then spread downward through the whole parent tree until it hits the defective node, removing the defective node from the node repository of its parent. The orphan parent will keep operating (as a disjointed parent) and serving related policy requests until a new node replaces the departed node (parent). At that time, the new node will rejoin the original tree because it will have been given the same node ID as the departed node. After joining the original tree network, the new node will broadcast a rejoin message, which requests all existing children to rejoin this new parent node.

The maintenance request process depends on the failure node level in the tree as shown in Figure 26. If the Root was the departed node, then the network takes no immediate action. In this case, the network will stay active but with some degradation due to the missing nodes on the Root node. The network immediately recovers from this failure once a new Root node replaces the departed node. The new Root rejoins the tree by broadcasting a re-join message. All other nodes at the next level (Level 1) respond to the new Root node. Consequently, the Root updates its node repository and reestablishes its connection to all of its predecessors. If the defective node is at Level 1, then the network takes no immediate action. Instead, it waits for the first policy request to the defective node. The node requesting the policy will get no response from the defective node; accordingly, the node will send another request to the Root. The Root will compare the policy key with its node repository. If a match is found then the Root will send a maintenance request to the related node. If the Root gets no response, then Root will remove the Node ID from its node repository. This situation will create an orphan tree, as the affected node will cause a subtree to become disconnected from the main tree. The network will keep functioning normally with some degradation related to the missing node on the defective node, but the tree will immediately recover from this once a new node replaces the departed node. The new replacement node will rejoin the tree by broadcasting a re-join message. All other nodes with Level 2 addresses will respond to the new node. Consequently, the new node will update its node repository and reestablish its connection to all of its predecessors. The process for node failure in levels 2 through L is the same, except for the total number of necessary maintenance request messages. Figure 26 illustrates the process for node failure at all levels, while Figure 27 shows the maintenance request activities.



Level 0
➢ No immediate action is required
➢ The tree will keep functioning normally except for maintenance requests and some missing policies stored on the Root.
➢ When a new node replaces the defective Root then it broadcasts a re-join message
➢ Children will re-connect to the new Root.
Level 1
➢ Root will send a maintenance request to the applicable Level 1 node
➢ If NOT successful Root updates the node status to un-used in its node repository table
➢ The first node joins the root will be given the first un-used address.
➢ The new node will broadcast a "re-join" message
➢ Children will try to re-connect with their new parent.
Level L-1
➢ Root will send a maintenance request to the applicable Level 1 node
➢ If successful, Level 1 node sends a maintenance request to the Next Level node and continue until Level L-1 is reached.
➢ If NOT successful, Level L-2 will update the node status to Un-used in its repository table
➢ The first node joins the parent will be given the first un-used address.
➢ The new node will broadcast a "re-join" message
➢ Children will will re-connect with their new parent.
Level L
➢ Root will send a maintenance request to the applicable Level 1 node
➢ If successful, Level 1 node sends a maintenance request to the Next Level node and continue until Level L is reached.
➢ If NOT successful, Level L-1 will update the node status to Un-used in its repository table
➢ The first node that joins the parent will be given the first un-used address.

Figure 26 Maintenance request process

6.8 Network structure maintenance

Nodes in WSNs are prone to failure due to environmental and hardware limitations. Nodes may fail for a variety of reasons, such as energy depletion, communication errors, or hardware failures. Node failure creates a phenomenon of an orphan tree. An orphan tree is a parent of a larger tree, which was disconnected from the main tree due to the failure of a node. This phenomenon may affect system performance, but it will not affect system operation or functionality. Although the system takes no immediate action in response to node failure, it will recover from this situation by issuing maintenance requests to the predecessor nodes of the departed node. The maintenance request will update the status of the defective node (change the failure node address to “available”) in its parent node's repository, which will then allow a new node to replace the defective node. Figure 27 illustrates the maintenance request activities.

The maintenance request is triggered by the monitoring policy request algorithm, which is illustrated in Figure 28. The system handles the maintenance request by monitoring the policy requests to the Root that are initiated by nodes. For each policy request to the Root, Root will assess if other nodes should have serviced the request. Root determines that by comparing the policy key in the request with the Node IDs in its node repository. If a match is found then Root sends a maintenance request to the affected node at the next level (Level 1). The maintenance request will keep going downward until it reaches the parent of the defective node.

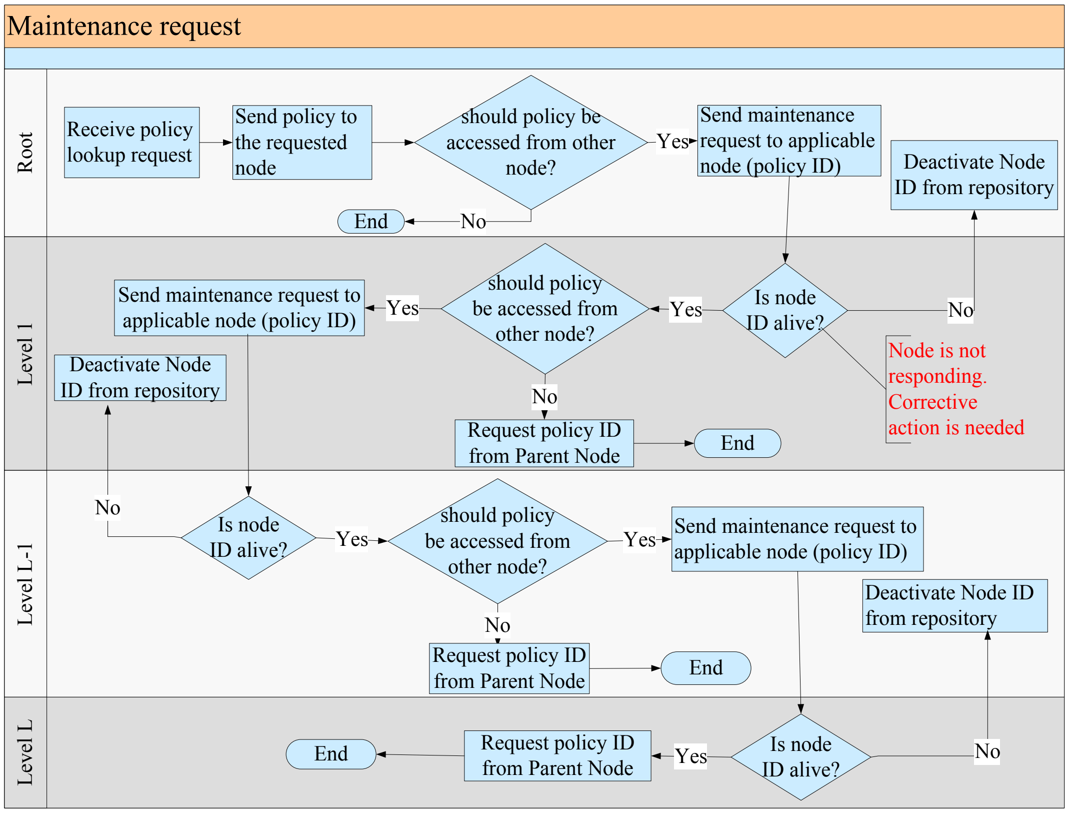

Maintenance request
Root
Receive policy lookup request
Send policy to the requested node
should policy be accessed from other node?
Yes
Send maintenance request to applicable node (policy ID)
Deactivate Node ID from repository
End
No
Level 1
Send maintenance request to applicable node (policy ID)
Yes
should policy be accessed from other node?
Yes
Is node ID alive?
No
Node is not responding. Corrective action is needed
Deactivate Node ID from repository
No
Request policy ID from Parent Node
End
Level L-1
No
Is node ID alive?
Yes
should policy be accessed from other node?
Yes
Send maintenance request to applicable node (policy ID)
Deactivate Node ID from repository
No
Request policy ID from Parent Node
End
Level L
End
Request policy ID from Parent Node
Yes
Is node ID alive?
No

Figure 27 Maintenance request activities

The complete algorithm for monitoring policy requests is illustrated in Figure 28. The algorithm relies on analyzing the policy ID and checking it against the node IDs in its local node repository. If the algorithm finds a match between the policy ID and node ID, then a new maintenance request is issued to the new matched node ID. The logic behind this process is that each policy should be stored on the node whose ID is the longest match to its policy ID. For example, if a node ID starts with 1 then all policy IDs that have the same number should be accessed from that node before it requests the targeted policy from the Root.

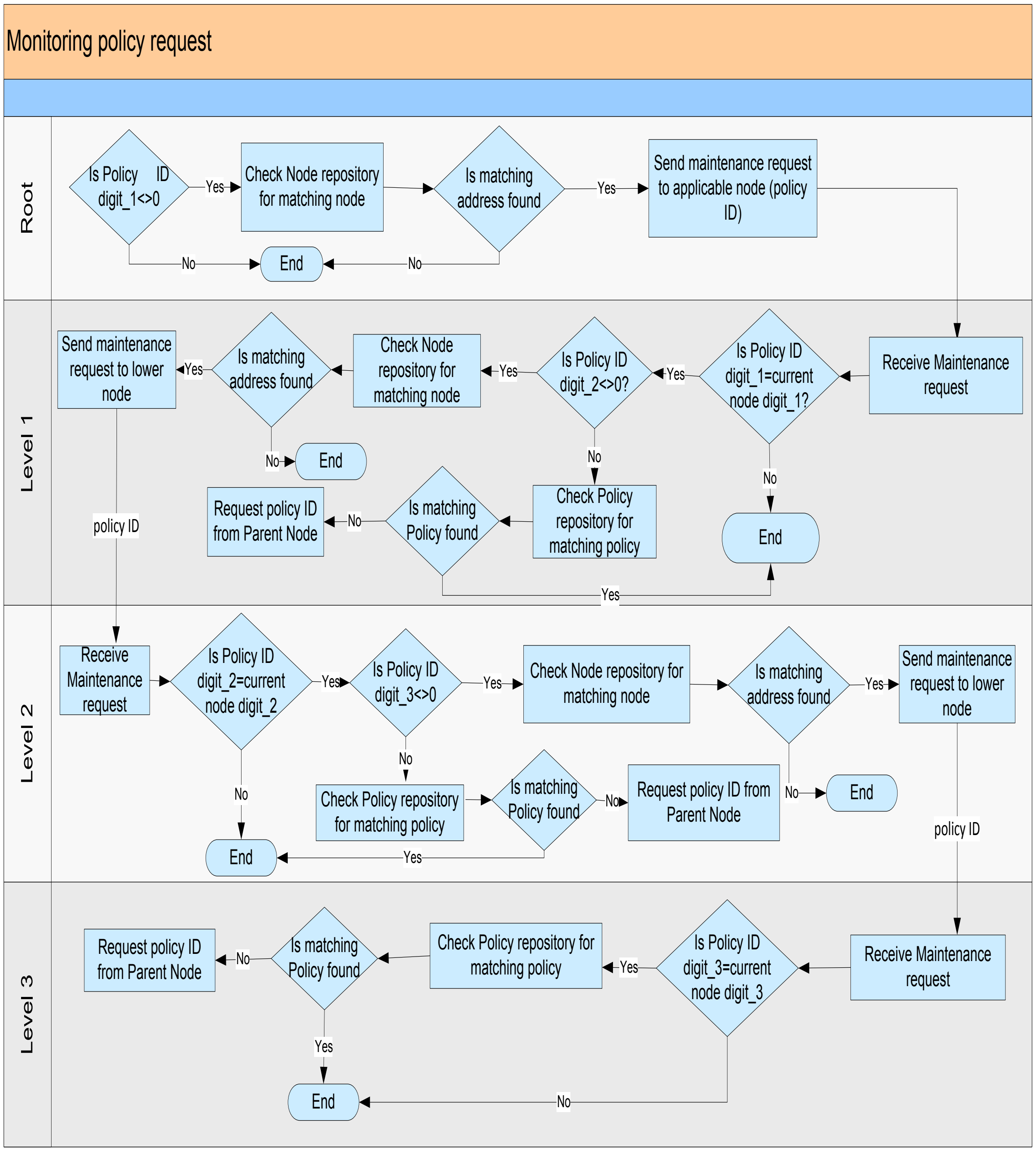

Monitoring policy request
Root
Is Policy ID digit_1<>0
Yes
Check Node repository for matching node
Is matching address found
Yes
Send maintenance request to applicable node (policy ID)
No
End
No
Level 1
Receive Maintenance request
Is Policy ID digit_1=current node digit_1?
Yes
Is Policy ID digit_2<>0?
Yes
Check Node repository for matching node
Is matching address found
Yes
Send maintenance request to lower node
No
End
No
Check Policy repository for matching policy
Is matching Policy found
No
Request policy ID from Parent Node
No
End
Yes
policy ID
Level 2
Receive Maintenance request
Is Policy ID digit_2=current node digit_2
Yes
Is Policy ID digit_3<>0
Yes
Check Node repository for matching node
Is matching address found
Yes
Send maintenance request to lower node
No
End
No
Check Policy repository for matching policy
Is matching Policy found
No
Request policy ID from Parent Node
No
End
Yes
policy ID
Level 3
Receive Maintenance request
Is Policy ID digit_3=current node digit_3
Yes
Check Policy repository for matching policy
Is matching Policy found
No
Request policy ID from Parent Node
Yes
End
No

Figure 28 Monitoring policy request flowchart

Chapter 7 Complexity analysis of TinyPolicy

There are many different tools and techniques for evaluating distributed network applications. The most common types of tools are Traffic Measurement, Simulation framework [56], and Mathematical framework. This thesis describes the dynamics of the overlay network elements with a mathematical model to perform a quantitative analysis of message complexity [57]. Our model is highly scalable as it provides fast results for "what if" analyses to evaluate network performance. However, the complexity of the model increases as the number of network elements increases.

The main objectives of this thesis are to increase the ability to support more policies in WSNs, to improve robustness of the distributed policy framework for WSNs, and to streamline the policy distribution processes. Therefore, our focus is to validate and evaluate the overlay network along with its related algorithms. The main objective of this chapter is to perform a quantitative analysis of message complexity [57] for the overlay network. Table 13 lists the network messages used in our framework.

Table 13 Network Message Sizes

Message Name	**Description**	**Message Number (Hex)**	**Message Size (Bytes)**
AM_REQUEST_MSG	***Policy Request***: Issued by any node to request a policy transaction	0x28	29
AM_RESPONSE_MSG	***Policy Response***: Issued by targeted node in response to a policy request	0x29	1

AM_HELLO_MSG	***Hello Message***: Issued by a new node when it is joining the overlay network	0x38	2
AM_HELLO_RESP_MSG	***Hello-Response***: Issued by the parent node in response to a previously received HELLO message	0x39	6
AM_HELLO_ACK_MSG	***Hello-Acknowledgment***: Issued by a newly joined node to confirm its new address	0x3a	4
AM_REJOIN_MSG	***Rejoin***: Issued by newly joined node to request all existing predecessor nodes to reconnect	0x48	2
AM_MAINT_MSG	***Maintenance***: Issued by the Root node to remove the defective node address from the parent node repository	0x49	4

For this model, a sensor network consists of a limited set of N identical nodes $\{n_1, \cdots, n_i\}$ where $i > 1$. Each policy p has to be stored in a node's local policy repository. Each node n_i stores a limited set of policies $P = \{p_1, \cdots, p_j\}$ where $j > 1$. There is one special node n_0 in the network, which is referred to as Root. Node n_0 is assumed to have the capability of storing a virtually unlimited set of policies P.

Each node has an overlay address a of length L bytes. A specific number of bits b of the overlay address represent one level v of the overlay tree structure. At any given time, a node can be either a parent with overlay address a or a leaf with overlay address a' where $a' > a$. Parent node n_i with overlay address a_i can have a limited number of leaf nodes f with overlay address a'_{ji} where the leaf number is j and i is the node number. The leaf node's overlay address has to be within the domain of its parent overlay address $a'_{ji} - a_i / 2^{b*v_j} < 2^b$ where the child level number in the overlay tree structure is v_j. All nodes have the same rate of transmission T.

7.1 Network formation messages

The overlay network starts being formed with the startup of the Root node. Each consecutive node has to broadcast a Hello message to join the network and wait for a response with an overlay address from neighboring nodes. Once a response arrives, the new node has to issue an Acknowledgment message to the parent node. Since all nodes except Root have to broadcast one Hello message, the expected total number of Hello messages (THM) can be calculated in (8), which is of a linear complexity $(N-1)$ or $O(N)$.

$$\text{THM} \qquad N-1 \tag{8}$$

All neighboring nodes have to respond to the new node with a Hello-Response message. At least two nodes are required to have one Hello-Response message; therefore, the expected total number of Hello-Response messages (THRM) can be calculated in (9), which has a complexity of $\left(\frac{N^2-N}{2}\right)$ or $O(N^2)$.

$$THRM = \sum_{i=2}^{N} i - 1 \tag{9}$$

After a Hello-Response message arrives with the overlay address, the new node has to acknowledge the new overlay address by responding with one Acknowledgment message to the new parent node. Thus, the expected total number of Acknowledgment messages (TAM) is given in (10), which is of a linear complexity $(N-1)$ or $O(N)$.

$$TAM = N - 1 \tag{10}$$

Finally, the total number of messages required to form an overlay network is the total of equations (8), (9), and (10).

7.2 Overhead messages

The overlay network has to maintain its tree structure. Therefore, some of the network messages are for maintaining the overlay structure; these include Re-Join, Re-join response, and maintenance messages.

Re-Join message

Each node, including the Root node, has to broadcast Re-join messages to re-establish relationships with child nodes if it has been previously disconnected for any reason. Hence, the total number of Re-join messages (TRM) is given in (11), which is of a linear complexity N or $O(N)$.

$$TRM = N \tag{11}$$

Re-join response message

Responses to a Re-join message will only come from legitimate children that fall within the assigned domain space of the issuing (parent) node. Hence, the total number of Re-join response messages (TRRM) is given in (12). During the formation of a new network, the total number of Re-join response messages should be zero, because the new joining node would always be a child node, not a parent node.

$$TRRM = \sum_{i=0}^{N} \sum_{j=1}^{f} j : \ a'_{ji} - a_i / 2^{b * v_i} < 2^b \tag{12}$$

Formula (12) plotted in Figure 29 **Error! Reference source not found.**and data table is in Appendix B Mathematical Model Data. The data table shows various network sizes ranging from 2 nodes to 25 nodes with a 3-level overlay tree structure. Each node has a local policy repository with a capacity of 20 entries, which means that each node can have a maximum of 20 policies in its memory. Analysis data shows number of nodes, Re-join message (number of messages, bytes and time), Re-join response message (number of messages, bytes and time). The chart trend shows that the formula has a liner complexity until the point where the number of nodes equal or greater than the capacity of the node repository then the formula complexity becomes constant.

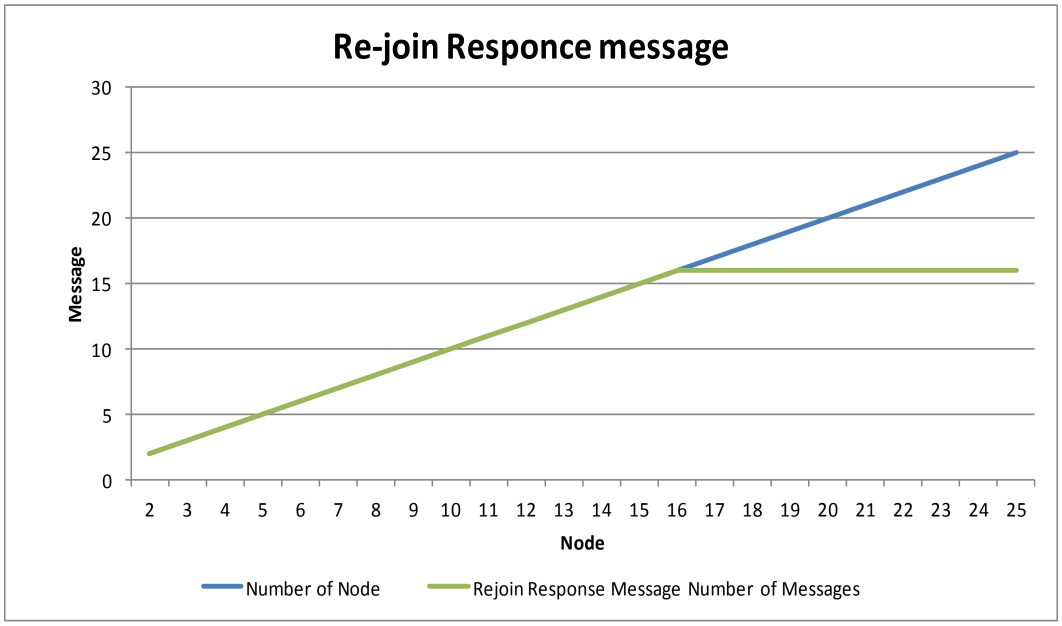


Figure 29 Re-join responce message

Maintenance message

Maintenance messages help the overlay network maintain a healthy structure. The maintenance messages are initiated by Root as a result of servicing a policy that it should not have serviced. This situation indicates that there is a missing policy or defective node and that maintenance service is required. Thus, Root requests all nodes in the affected parent to update the status of their associated (child) nodes. The number of maintenance messages issued depends on the level of the affected node. If the affected node is v_i, where v is the level of defective node i, then the total number of maintenance messages (TMM) is given in (13).

$$TMM = \sum_{i=1}^{N} \sum_{j=1}^{p} v_i : \nexists p_{ij} \in n_i \quad (13)$$

Formula (13) plotted in Figure 30 and data table is in Appendix B Mathematical Model Data. The data table shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node has a local policy repository with a capacity of 20 entries. Analysis data shows number of nodes, policy repository size, network total policies, network tree levels, and number of maintenance messages. The chart trend shows that the formula has a liner complexity.

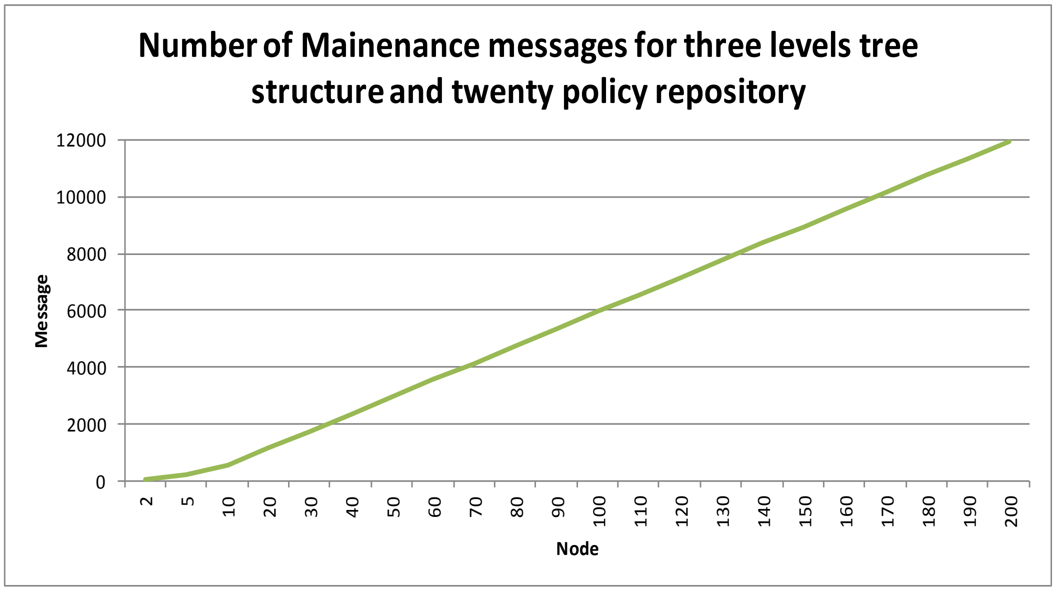


Figure 30 total number of maintenance messages

7.3 Policy administration message

The operations of policy administration are: Load, Remove, Enable, Disable, Get, Send, and Reload. The Load operation requests the system to issue one message to store the policy in the targeted node. However, the targeted node may issue consecutive requests if the targeted node has a longer matching node address in its repository. Therefore, the total number of policy load messages (TPLM) is given in (14).

$$TPLM = \sum_{i=1}^{N}\sum_{k=1}^{p} v_i : \nexists p_{ik} \in n_i \wedge \exists n_a \in N \; where \; n_{a_i} \approx p_{ik} \quad (14)$$

Formula (14) is similar in complexity to formula (13), which plotted in Figure 30. The chart trend shows that the formula has a liner complexity.

The system will broadcast an administrative message for Remove, Enable, and Disable. Therefore, the maximum possible number of messages (TADM) is given in (15).

$$\text{TADM} = \sum_{i=1}^{N}\sum_{k=1}^{p} 1 : \exists p_{ik} \in n_i \quad (15)$$

Formula (15) plotted in Figure 31 and data table is in Appendix B Mathematical Model Data. The data table shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node has a local policy repository with a capacity of 20 entries. Analysis data shows number of nodes, policy repository size, network total policies, network tree levels, and total number of administrative messages. The chart trend shows that the formula has a liner complexity.

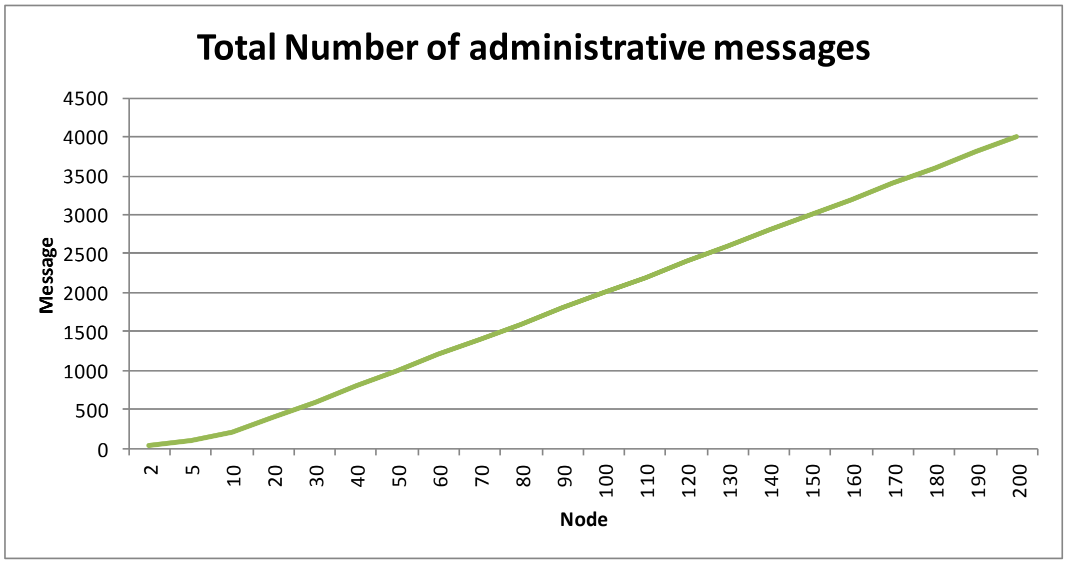


Figure 31 Total number of administrative messages

The number of messages for Get depends on the level of the targeted node. The system tries to get the policy from the targeted node. If the policy does not exist then the targeted node searches the repositories of its children for addresses matching the required policy. The Get message is forwarded to the child node if a match is found; otherwise, the requesting node has to get the policy from the Root. Hence, the total number of Get messages (TGMT) is given in (16) if the policy exists in the target node. The total number of Get messages (TGMC) is given in (17) if the policy exists in a child of the targeted node. The total number of Get messages (TGMR) is given in (18) if the policy exists in the Root node.

$$\text{TGMT} = \sum_{i=1}^{N} \sum_{k=1}^{p} 1 : \ \exists p_{ik} \in n_i \tag{16}$$

$$\text{TGMC} = \sum_{i=1}^{N} \sum_{k=1}^{p} 2 : \ \nexists p_{ik} \in n_i \ \wedge \exists \boldsymbol{n_a} \in \boldsymbol{n_i} \, , \boldsymbol{n_{a_i}} \approx \boldsymbol{p_{ik}} \wedge \boldsymbol{p_{ik}} \in \boldsymbol{n_{a_i}} \tag{17}$$

$$\mathrm{TGMR} = \sum_{i=1}^{N}\sum_{k=1}^{p} \mathbf{3} : \exists \boldsymbol{p_{ik}} \in \boldsymbol{n_0} \qquad \textbf{(18)}$$

Formula (16), (17), and (18) are similar in complexity to formula (15), which plotted in Figure 31. The chart trend shows that the formula has a liner complexity.

The system responds with only one message if the targeted policy exists in the node's repository. Hence, the total number of policy response messages (TPRM) is given in (19).

$$\mathrm{TPRM} = \sum_{i=1}^{N}\sum_{k=1}^{p} 1 : \exists p_{ik} \in n_i \qquad \textbf{(19)}$$

Formula (19) is similar in complexity to formula (15), which plotted in Figure 31. The chart trend shows that the formula has a liner complexity.

In our mathematical model, all equations have a complexity that is constant, linear, or quadratic in input size *n*. To illustrate the mathematical model by example, let us assume a WSN with 20 sensor nodes and a 4-level overlay tree structure, with each node's local policy repository capable of storing up to 20 policies. The formation activities of the network are shown in Table 14.

Table 14 Network Performance for WSN with 20 Nodes

Message type	Equation #	Number of Messages	Message Size (bytes)	Total Message Size (bytes)
New network formation:				
Hello	(8)	19	2	38
Hello response	(9)	190	6	1,140
Acknowledgment	(10)	19	4	76
Rejoin	(11)	20	2	40

Rejoin response	(12)	0[4]	6[5]	0
Policy administration: Load	(14)	400-1200	29	11,600-34,800

The total number of messages required for overlay network formation is 19 (Hello) + 190 (Hello response) + 19 (Acknowledgment) + 20 (Rejoin) = 248. The total size of the data consumed for overlay network formation is the total size of all required network messages, which is $38 + 1{,}140 + 76 + 40 = 1{,}294$ bytes. Loading all policies in the network requires between 400 and 1200 messages, which translates into a data size between 11,600 and 34,800 bytes. Therefore, the total data size for forming the overlay network and loading all policies into the network is between 11,600 + 1,294 = 12,894 and 34,800 + 1,294 = 36,094 bytes. If each sensor node has a transmission rate of 250 kbps or 250 kbps/8 bits = 31,250 bytes/s, then the time required for one node to handle the overlay network formation and policy loading is between $\frac{12{,}894}{31{,}250} \approx 0.41$ s and $\frac{36{,}094}{31{,}250} \approx 1.15$ s. Since there are 20 nodes in the network, if we assume that they will share the load equally, then the time required by the network is between $\frac{0.41}{20} \approx 0.02$ s and $\frac{1.15}{20} \approx 0.06$ s.

Heterogeneous network is a network with different node's resource limitations. Heterogeneous network affects the formation process by influencing only the number messages for the Hello-Response and the Rejoin-Rresponse messages. Nodes that are more restricted are expected to issue less number of these messages as they quickly reach their full repository capacity. The other impact of the heterogeneous network is the network topology as it is expected for nodes to be clustered around the higher capacity node. Therefore, restricted nodes should have fewer children than the more capable nodes.

[4] No orphan nodes must exist at the startup of a new network.
[5] The re-join message and the Hello response message are the same size

The following example illustrates the impact of a heterogeneous network. Let us assume a WSN with 20 nodes and 4-levels overlay tree structure. Five of the sensor nodes have enough memory capacity to store fifteen entries for each node and policy repository. Five other nodes can store up to ten entries for each node and policy repository. Five more nodes have a capacity of five entries for each node and policy repository. The remaining five nodes have limited capacity of two entries for each node and policy repository. The formation activities of the network are shown in Table 15.

Table 15 Network Performance for heterogeneous WSN with 20 Nodes

Message type	Equation #	Number of Messages	Message Size (bytes)	Total Message Size (bytes)
New network formation:				
Hello	(8)	19	2	38
Hello response	(9)	105	6	630
Acknowledgment	(10)	19	4	76
Rejoin	(11)	20	2	40
Rejoin response	(12)	0[6]	6[7]	0
Policy administration:				
Load	(14)	160-480	29	4,640 – 13,920

The total number of messages required for overlay network formation is 19 (Hello) + 105 (Hello response) + 19 (Acknowledgment) + 20 (Rejoin) = 163. In this example, we have less Hello response messages than the previous example (heterogeneous WSN) because after each two new nodes joining the network one of the five very limited resource nodes will reach its capacity and stop issue any farther Hello response messages. The total size of the data consumed for overlay network formation

[6] No orphan nodes must exist at the startup of a new network.

[7] The re-join message and the Hello response message are the same size

is the total size of all required network messages, which is $38 + 630 + 76 + 40 = 784$ bytes. Loading all policies in the network requires between 160 and 480 messages, which translates into a data size between 4,640 and 13,920 bytes. Therefore, the total data size for forming the overlay network and loading all policies into the network is between 4,640 + 784 = 5,424 and 13,920 + 784 = 14,704 bytes. If the transmission rate is 250 kbps or 250 kbps/8 bits = 31,250 bytes/s for each sensor, then the time required for one node to handle the overlay network formation and policy loading is between $\frac{5{,}424}{31{,}250} \approx 0.17$ s and $\frac{14{,}704}{31{,}250} \approx 0.47$ s.

7.4 Data analysis

This discrete mathematical model analyzes the complexity of significant elements of the overlay network. The mathematical model can be expanded to include more network elements, which will require adding proper statistical models. However, our goal was to focus on analyzing the complexity of just the overlay network, isolating the impact of other network elements. Our analysis has yielded data on overlay network formation, policy loading, and the Bloom filter, as well as data on how the performance of the central policy repository approach compares with that of the distributed policy repository approach. Detailed results from our mathematical model are provided in Appendix B Mathematical Model Data.

Network formation performance

Results of our analysis of network formation are provided in Appendix B Mathematical Model Data. The table shows various network sizes ranging from 2 nodes to 200 nodes. Each node has a leaf table with a capacity of 16 entries, meaning that each parent node can have a maximum of 16 children. Data from our analysis show the number of messages, bytes, and time (in seconds) of all types of messages required for network formation: Hello, Response, Acknowledgment, and Rejoin. Figure 32 illustrates that Response messages consumed 37.5% of total network formation time for a network of 2 nodes, increasing to 98.7% for a larger network of 200 nodes. The percentage of time declined significantly for all other types of overlay network messages. Hello messages declined from 12.5% to 0.33%. Acknowledgment messages declined from 25% to 0.66%. Finally, Re-join messages declined from 25% to 0.33%.

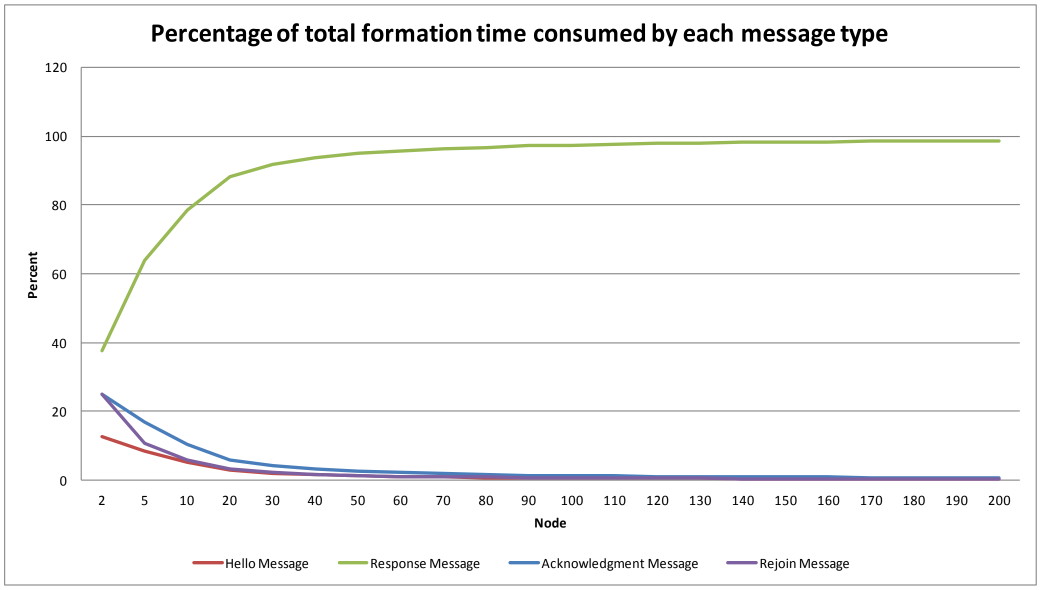


Figure 32 Percentage of total formation time

Policy loading performance

Analysis data for policy loading performance are provided in Appendix B Mathematical Model Data. The table shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node's local policy repository has a capacity of 20 entries, meaning that each node can have a maximum of 20 policies in its memory. Data from our analysis show the number of messages, bytes, and time (in seconds) of all types of required messages (Get and Response) to load policies into the network for P2P algorithm usage and into the local node for local node usage. The table shows the minimum, maximum, and average performance of each category. Figure 33 illustrates that the best-case performance (minimum time required) for policies loading into the network is 33% of the total time versus 67% for loading policies into the local node, a difference of 34%. This difference declined to 14% in the worst-case performance (maximum time required), with 43% of the total time for loading policies into the network versus 57% for loading policies into the local node. This decline is caused by the increase in time required to load policies into the network from 33% to 43%, a 10% increase. On the other side, the time required to load policies into the local node declined from 67% to 57%, a 10% decrease. The best-case performance (Min) assumes that the Root will only need to store the policies on one of the first level nodes, while the worst-case performance (Max) assumes that policies have to travel all the way down to the lowest level (third level is assumed for this analysis). The other difference is that loading policies into the local node as the best-case performance assumes that the policy always exists on the targeted node, while the worst-case performance assumes that the process searches the targeted node and then an applicable child, and it finally accesses the policy from the Root node. The performance can be improved by using different network message structures for policy lookup and access. At present, the system uses the same network message structure for both policy lookup and access. Using a shorter message structure for policy lookup can significantly improve the performance of loading policies into the local node.

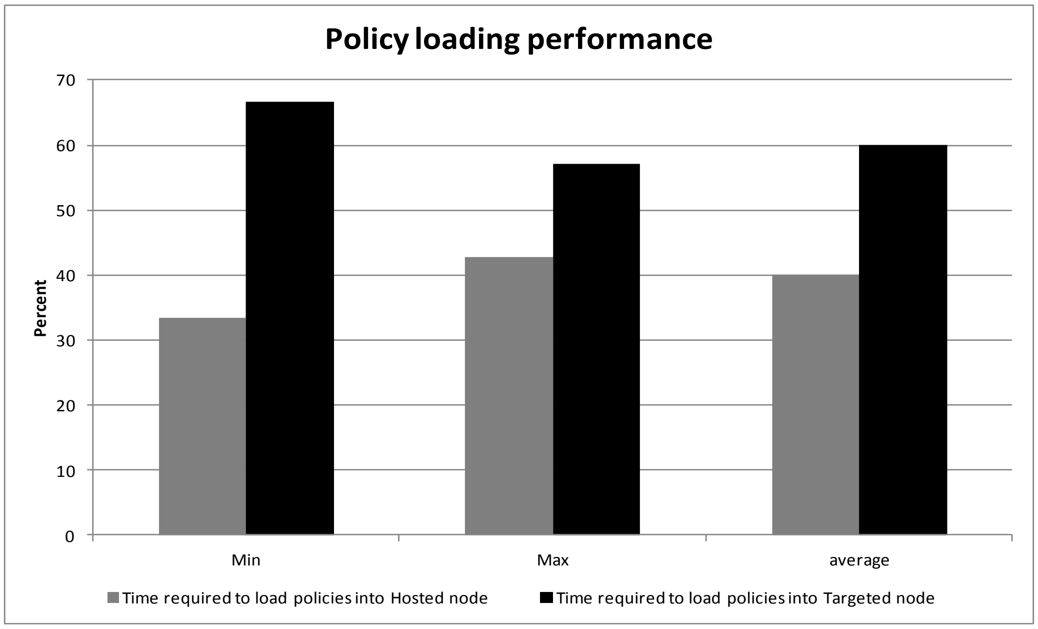


Figure 33 Policy loading performance

Bloom filter performance

Results from our analysis of Bloom filter performance are provided in Appendix B Mathematical Model Data. The table shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node's local policy repository has a capacity of 20 entries, meaning that each node can have a maximum of 20 policies in its memory. Data from our analysis show the total number of policies, number of messages, bytes, and time (in seconds) required to look up policies. The table shows the minimum, maximum, and average performance of each category. Finally, the table shows the amount of time saved by using the Bloom filter, assuming that the rate of missing policies is 30%. Figure 34 illustrates that the time required to look up all policies in the best-case performance (Min) is almost equal to the total time saved by the Bloom filter under the worst-case performance (Max). From the table, one can also deduce that the average saving of the Bloom filter is 60% in the best-case performance and 20% in the worst-case performance of looking up all policies.

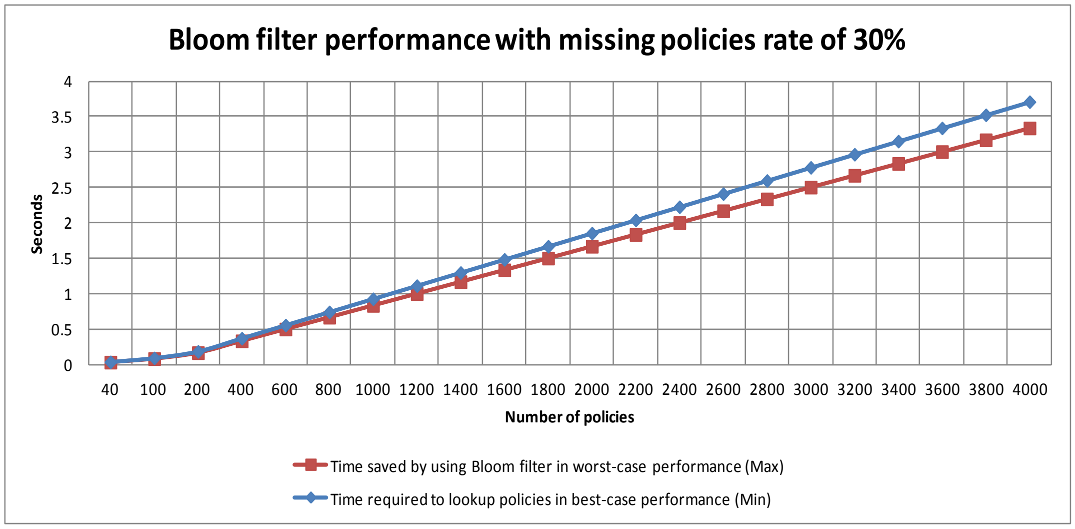


Figure 34 Bloom filter performance with missing policies rate of 30%

Central policy repository performance

In a system with a central policy repository, the Root node functions as the only policy repository in the network. Since there is no central policy repository system to evaluate, TinyPolicy was modified to resemble central repository system operation. The table in Appendix B Mathematical Model Data shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node's local policy repository has a capacity of 20 entries, meaning that each node can have a maximum of 20 policies in its memory. Data from our analysis show central policy repository size (Root), total number of policies, number of messages, number of bytes, and time (in seconds) required to load policies into the local node repository using the central repository approach. Finally, the table shows the amount of time required to load the same number of policies using the distributed policy repository approach. The data shows that the central policy repository system may perform 150% faster than the distributed repository approach, as shown in Figure 35. However, the central policy repository system will not provide the benefits of the distributed system, such as reliability by having multiple policy repositories and multiple copies of the same policy in the WSN. Load distribution is another benefit of the distributed approach. In the centralized approach, policies are concentrated on the Root node; the more policies that exist in the WSN, the more overhead the Root node will incur. In the distributed approach, the load is uniformly distributed among all WSN nodes. Resiliency is another benefit of the distributed system, as the network will keep operating even with the loss of many hosted nodes, whereas the central repository approach will not be able to deliver any policies if the Root fails. A more detailed discussion of the benefits of the distributed repository approach is provided in Chapter 3 TinyPolicy: A Distributed Policy Framework.

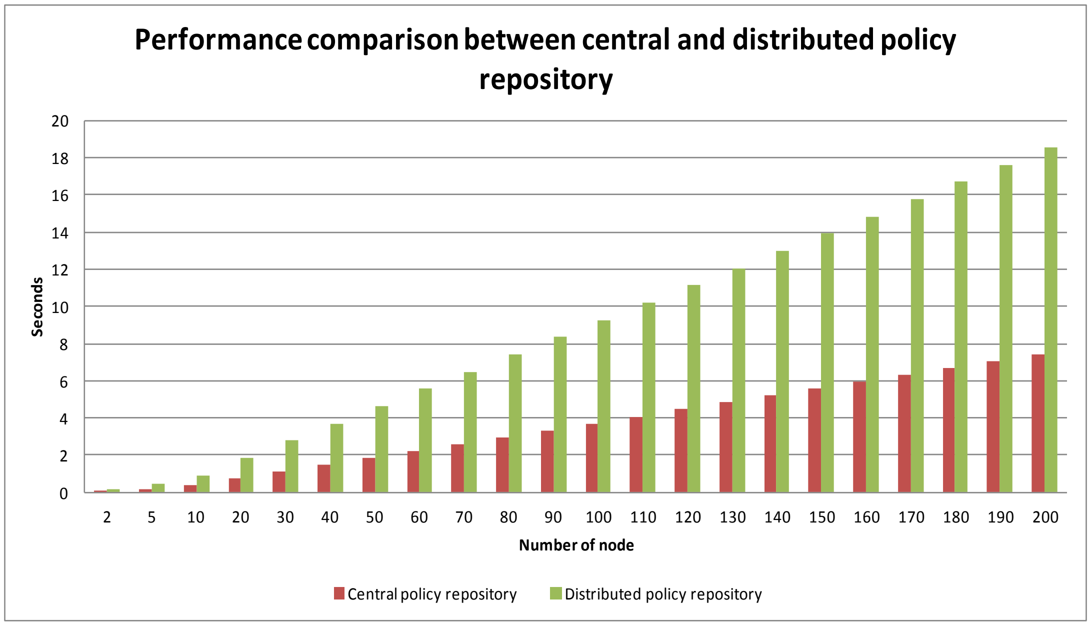


Figure 35 Central and distributed policy repository performance

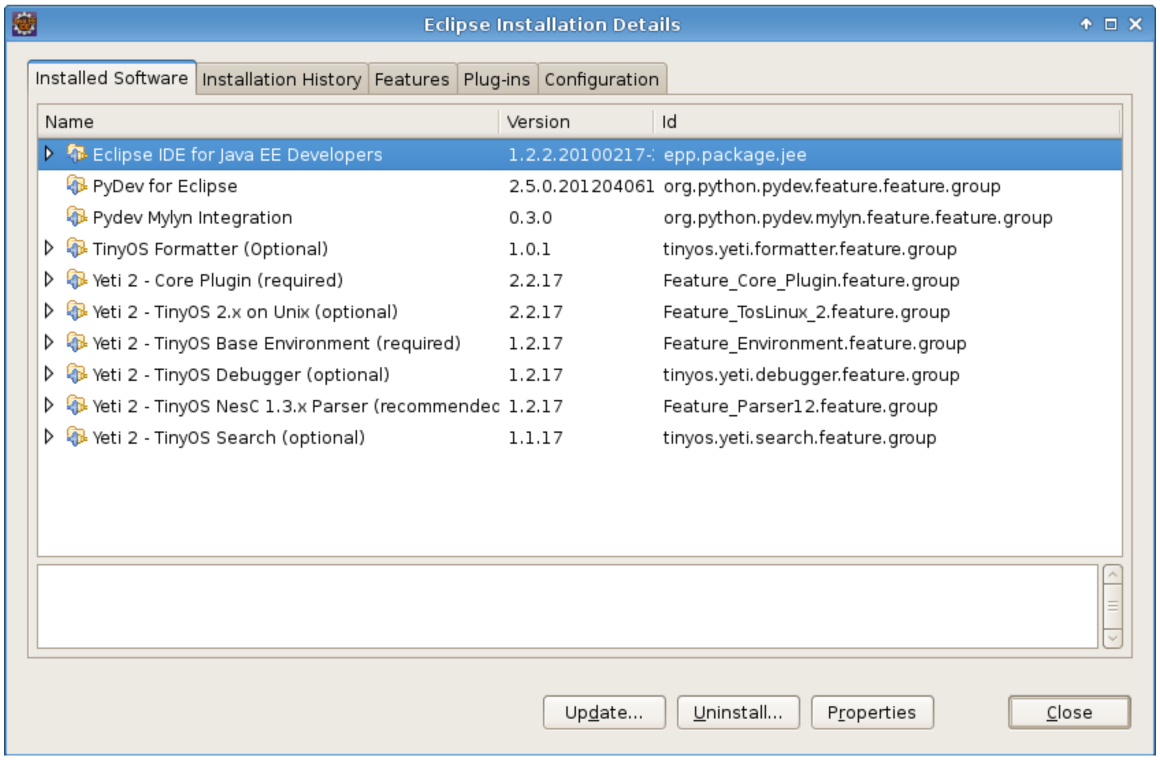
Chapter 8 Validation of TinyPolicy through implementation in TinyOS

Our work was greatly inspired by a policy-based platform called Finger/Finger2 developed by Bourdenas et al. [6], [14], [15] However, significant design and implementation changes were made to it to accommodate the premises of this thesis. The working process of our research was as follows:

1. Acquire the Finger2 source code from the original author but without the security component, which is unrelated to this research as the security issue is out of our scope.
2. Analyze the Finger2 architecture and design. The current architecture of Finger2 does not support distributed policies. Therefore, major modifications are required that are discussed in Chapter 3 TinyPolicy: A Distributed Policy Framework.
3. Create a new debugging tool to facilitate the simulation process, and test the policy execution in the simulation environment.
4. Simulate the Finger2 platform using the TOSSIM simulator [58].
5. Simulate the new framework using the selected simulation software. Capture, analyze, and comment on the new framework simulation results.
6. Develop simulation scenarios. (As discussed in section 8.2, various scenarios were created to validate specific objectives.)
7. Analyze and compare the capabilities of the distributed policy system and Finger2.

The development environment for this research consisted of the following software:

1. The underlying network, assumed to be a single hub network using Box-Mac 2 protocol [59]. Figure 37 illustrates the physical network setup.
2. Ubuntu version 7.04 operating system (Linux-like operating system) [60].
3. TinyOS version 2.1.0 operating system [53].
4. nesC programming language [53].
5. TOSSIM simulation software [53].
6. Eclipse Integrated Development Environment (IDE) [61] Build id: 20100218-1602. Figure 36 shows more detailed information about the software installed on Eclipse.
7. Python programming language [62] version 2.5.1.
8. Yeti plug-in for tinyOS [63] version 2.
9. GTK multi-platform toolkit for creating graphical user interfaces [64].


Eclipse Installation Details

Installed Software | Installation History | Features | Plug-ins | Configuration

Name	Version	Id
Eclipse IDE for Java EE Developers	1.2.2.20100217-:	epp.package.jee
PyDev for Eclipse	2.5.0.201204061	org.python.pydev.feature.feature.group
Pydev Mylyn Integration	0.3.0	org.python.pydev.mylyn.feature.feature.group
TinyOS Formatter (Optional)	1.0.1	tinyos.yeti.formatter.feature.group
Yeti 2 - Core Plugin (required)	2.2.17	Feature_Core_Plugin.feature.group
Yeti 2 - TinyOS 2.x on Unix (optional)	2.2.17	Feature_TosLinux_2.feature.group
Yeti 2 - TinyOS Base Environment (required)	1.2.17	Feature_Environment.feature.group
Yeti 2 - TinyOS Debugger (optional)	1.2.17	tinyos.yeti.debugger.feature.group
Yeti 2 - TinyOS NesC 1.3.x Parser (recommendec	1.2.17	Feature_Parser12.feature.group
Yeti 2 - TinyOS Search (optional)	1.1.17	tinyos.yeti.search.feature.group

Figure 36 Development environment

Figure 37 illustrates the physical network setup, which consists of a number of sensor nodes and one Root node connected to a computer through a USB cable. The Administrator can use the computer to communicate with the Root and any other sensor node in the network.

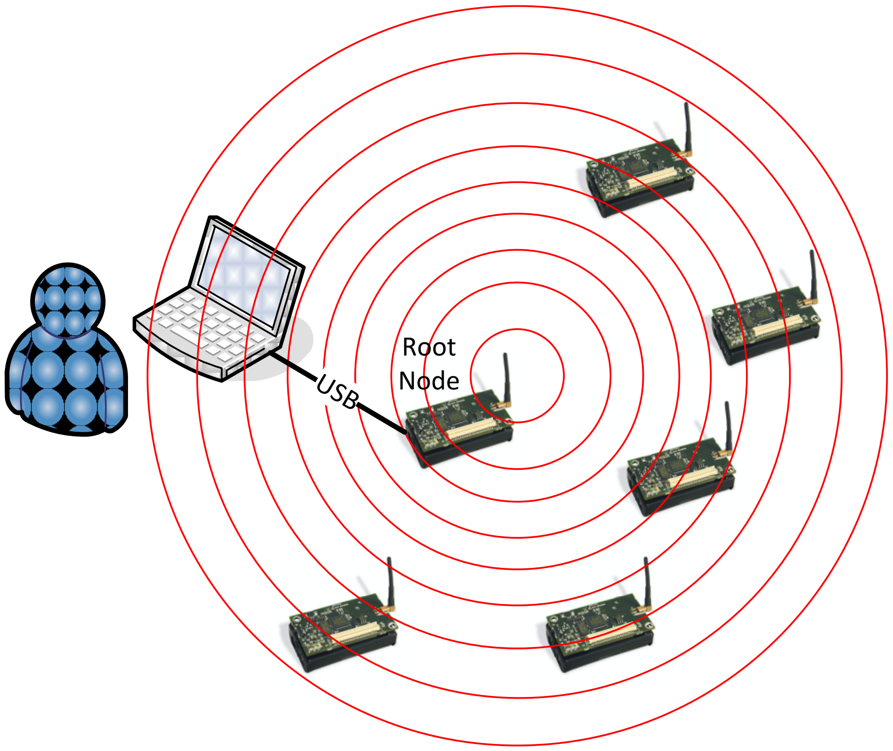


Figure 37 Physical setup

The policy-based framework of this thesis is running on the TinyOS platform version 2.1.0, which is an open-source operating system for wireless sensor networks [53]. The engineering design process for the TinyOS platform requires the developer to specify and link the needed software components for the system during development. The developer is required to "wire" these components together to establish static links among them. This "wiring" permits the invocation and handling of methods and events provided by a component. These relationships can be depicted in a components diagram that illustrates the interaction of these objects and their interconnections as shown in Figure 39 and Figure 40. A module diagram is a type of diagram that depicts the relationships between different modules (packages) of the system as shown in Figure 38.

Our framework is divided into two parts as shown in Figure 38. The first part is the policy management part, which is responsible for administering, controlling, monitoring, and executing policies in the node. The central point of this part is the ObligationManagerP module, which represents the Policy Decision Point (PDP) in the

IETF/DMTF policy architecture model [18]. The policy management part consists of the following main modules:

ObligationManagerP: The policy decision point (PDP) of the engine that is responsible for interpreting policies. Based on the module's decision, actions may be triggered by forwarding the process to the ActionRepositoryP module.

- ***PolicyRepositoryP***: Maintains local policies and provides access to policies when required.
- ***EventManagerP***: Intercepts internal and external events and forwards them to ObligationManagerP for analysis and decisions.
- ***ActionRepositoryP***: Stores all available actions and executes any actions that may be required by the PDP ObligationManagerP.
- ***PredicateRepositoryP***: Stores all available predicates (logical operations) and helps interpret policy conditions when needed by the PDP ObligationManagerP.
- ***HashingP***: Stores the Bloom filter array and performs any hashing request.
- ***RequestHandlerP***: Receives external policy requests and forwards them to EventManagerP to take an appropriate action.

The second part is the node management for the overlay network, which is responsible for forming, administering, and maintaining the overlay network nodes. As illustrated in Figure 38, node management consists of the following main modules:

- ***RequestNodeHandlerP***: Receives all overlay network communications and takes appropriate actions accordingly.
- ***DemoAppP***: Performs the startup tasks and initiates the process of forming the overlay network.

- ***NodeRepositoryP***: Performs the repository initialization process and maintains information about the current node's children.

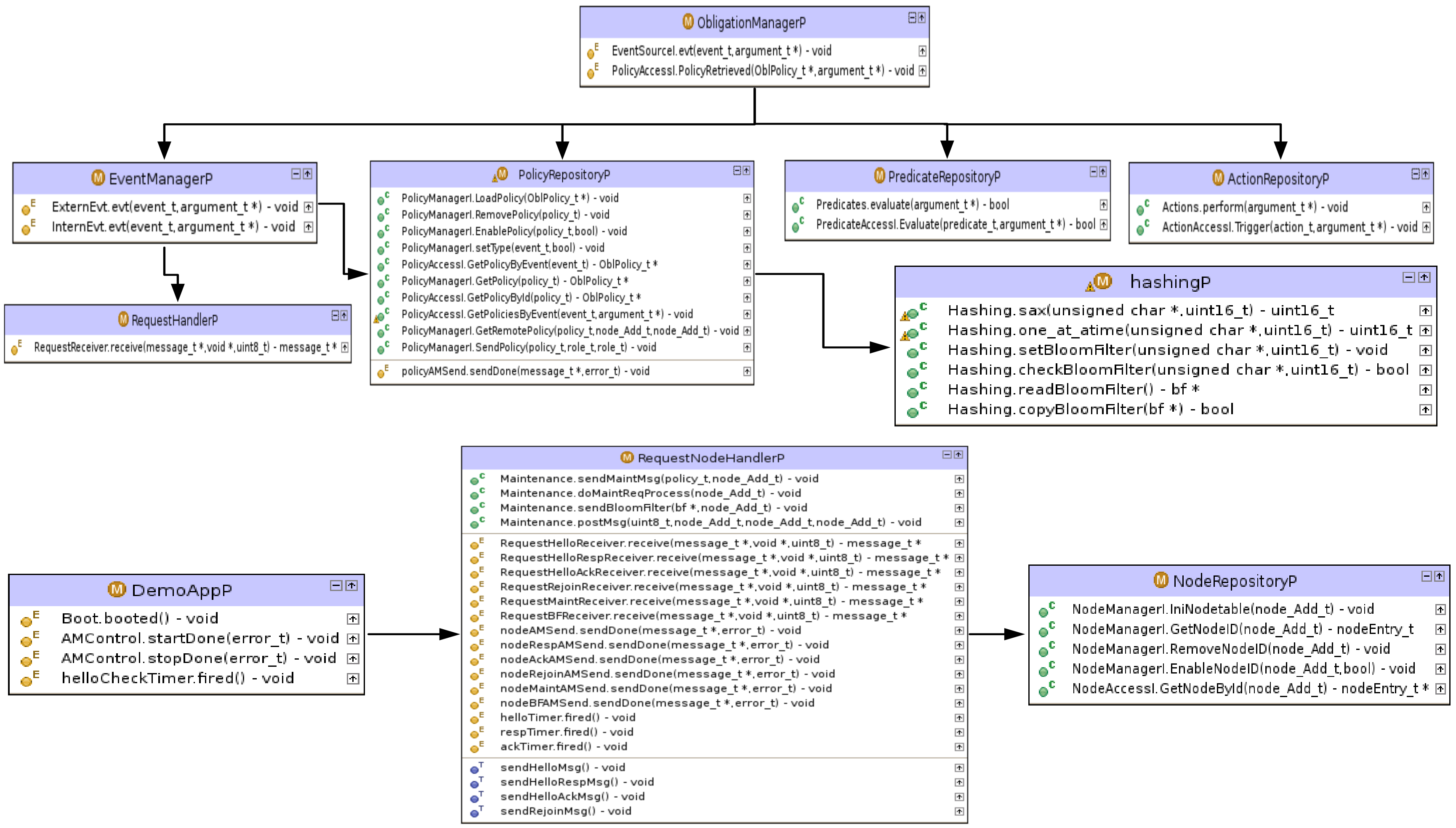
Figure 39 and Figure 40 are the component diagrams for the framework. They illustrate the interaction between the framework's objects and their interconnections. It is useful to show all components in the system, both operating system components and user-created components. In addition, a management tool, Policy IDE, was created to help administer, test, and debug policies. More information about this tool is provided in Chapter 8 Validation of TinyPolicy through implementation in TinyOS.

The new features of our fully distributed policy-based framework come with an expected overhead in program size and performance, compared with conventional policy management systems like Finger/Finger2. Table 16 shows that the program size of our new system is 29.3 kB, compared with 12.4 kB for the Finger2 system. This increase was mainly due to the new functionalities of the overlay network and other P2P-associated algorithms. Although the program size is roughly double that of Finger2, it is still within the normal limit for wireless sensor nodes; a Mica or IRIS device has 128 kB of in-system programmable flash memory and 4 kB of in-system programmable EEPROM. The TinyOS operating system divides any compiled program into two parts for ROM and RAM memory. ROM includes the code and initialized data, while RAM includes both initialized and uninitialized data but not stack data.

Table 16 Program Size in Bytes

	RAM (Bytes)	ROM (Bytes)	Total (Bytes)
Finger 2	913	11,534	12,447
TinyPolicy	6,994	22,308	29,302

It is misleading to think that the difference in code size of 16,855 byte (29,302-12,447) can be used to store more policies on the sensor node because the increase in size is divided into two types of memories that are ROM and RAM as shown in Table 16. The framework uses 10,774 byte (22,308 - 11,534) more in ROM memory, which can be used only for code and initialized data. Therefore, we cannot use this memory space for data storage such as more policies. On the other hand, the framework uses 6,081 (6,994 – 913) byte more in RAM memory. RAM memory can be used for data storage such as policies. However, if the Bloom filter size of 5.5K byte is deducted from that increase then the actual size increase is 581 byte, which is equal to 20 policies (581 byte / 29 byte per policy).


ObligationManagerP
EventSourceI.evt(event_t,argument_t *) - void
PolicyAccessI.PolicyRetrieved(OblPolicy_t *,argument_t *) - void
EventManagerP
ExternEvt.evt(event_t,argument_t *) - void
InternEvt.evt(event_t,argument_t *) - void
RequestHandlerP
RequestReceiver.receive(message_t *,void *,uint8_t) - message_t *
PolicyRepositoryP
PolicyManagerI.LoadPolicy(OblPolicy_t *) - void
PolicyManagerI.RemovePolicy(policy_t) - void
PolicyManagerI.EnablePolicy(policy_t,bool) - void
PolicyManagerI.setType(event_t,bool) - void
PolicyAccessI.GetPolicyByEvent(event_t) - OblPolicy_t *
PolicyManagerI.GetPolicy(policy_t) - OblPolicy_t *
PolicyAccessI.GetPolicyById(policy_t) - OblPolicy_t *
PolicyAccessI.GetPoliciesByEvent(event_t,argument_t *) - void
PolicyManagerI.GetRemotePolicy(policy_t,node_Add_t,node_Add_t) - void
PolicyManagerI.SendPolicy(policy_t,role_t,role_t) - void
policyAMSend.sendDone(message_t *,error_t) - void
PredicateRepositoryP
Predicates.evaluate(argument_t *) - bool
PredicateAccessI.Evaluate(predicate_t,argument_t *) - bool
ActionRepositoryP
Actions.perform(argument_t *) - void
ActionAccessI.Trigger(action_t,argument_t *) - void
hashingP
Hashing.sax(unsigned char *,uint16_t) - uint16_t
Hashing.one_at_atime(unsigned char *,uint16_t) - uint16_t
Hashing.setBloomFilter(unsigned char *,uint16_t) - void
Hashing.checkBloomFilter(unsigned char *,uint16_t) - bool
Hashing.readBloomFilter() - bf *
Hashing.copyBloomFilter(bf *) - bool
RequestNodeHandlerP
Maintenance.sendMaintMsg(policy_t,node_Add_t) - void
Maintenance.doMaintReqProcess(node_Add_t) - void
Maintenance.sendBloomFilter(bf *,node_Add_t) - void
Maintenance.postMsg(uint8_t,node_Add_t,node_Add_t,node_Add_t) - void
RequestHelloReceiver.receive(message_t *,void *,uint8_t) - message_t *
RequestHelloRespReceiver.receive(message_t *,void *,uint8_t) - message_t *
RequestHelloAckReceiver.receive(message_t *,void *,uint8_t) - message_t *
RequestRejoinReceiver.receive(message_t *,void *,uint8_t) - message_t *
RequestMaintReceiver.receive(message_t *,void *,uint8_t) - message_t *
RequestBFReceiver.receive(message_t *,void *,uint8_t) - message_t *
nodeAMSend.sendDone(message_t *,error_t) - void
nodeRespAMSend.sendDone(message_t *,error_t) - void
nodeAckAMSend.sendDone(message_t *,error_t) - void
nodeRejoinAMSend.sendDone(message_t *,error_t) - void
nodeMaintAMSend.sendDone(message_t *,error_t) - void
nodeBFAMSend.sendDone(message_t *,error_t) - void
helloTimer.fired() - void
respTimer.fired() - void
ackTimer.fired() - void
sendHelloMsg() - void
sendHelloRespMsg() - void
sendHelloAckMsg() - void
sendRejoinMsg() - void
DemoAppP
Boot.booted() - void
AMControl.startDone(error_t) - void
AMControl.stopDone(error_t) - void
helloCheckTimer.fired() - void
NodeRepositoryP
NodeManagerI.IniNodetable(node_Add_t) - void
NodeManagerI.GetNodeID(node_Add_t) - nodeEntry_t
NodeManagerI.RemoveNodeID(node_Add_t) - void
NodeManagerI.EnableNodeID(node_Add_t,bool) - void
NodeAccessI.GetNodeById(node_Add_t) - nodeEntry_t *

M: Module E: External C: Class T: Task

Figure 38 Module diagram

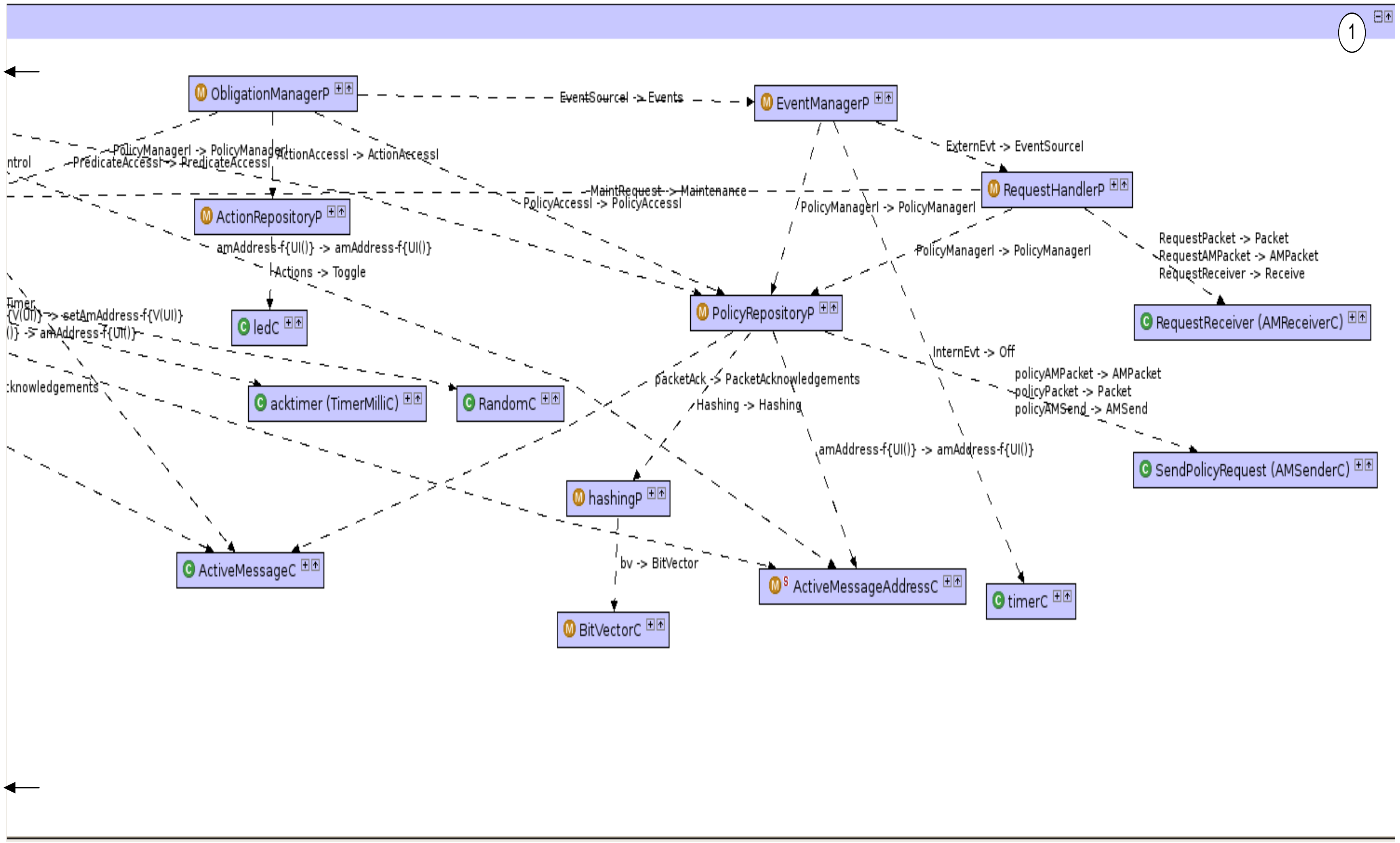

1
ObligationManagerP
EventManagerP
EventSourceI -> Events
ExternEvt -> EventSourceI
PolicyManagerI -> PolicyManagerI
PredicateAccessI -> PredicateAccessI
ActionAccessI -> ActionAccessI
ntrol
MaintRequest -> Maintenance
PolicyAccessI -> PolicyAccessI
RequestHandlerP
ActionRepositoryP
PolicyManagerI -> PolicyManagerI
amAddress-f{UI()} -> amAddress-f{UI()}
Actions -> Toggle
PolicyManagerI -> PolicyManagerI
RequestPacket -> Packet
RequestAMPacket -> AMPacket
RequestReceiver -> Receive
ledC
PolicyRepositoryP
RequestReceiver (AMReceiverC)
Timer
{V(UI)} -> setAmAddress-f{V(UI)}
()} -> amAddress-f{UI()}
InternEvt -> Off
policyAMPacket -> AMPacket
policyPacket -> Packet
policyAMSend -> AMSend
:knowledgements
acktimer (TimerMilliC)
RandomC
packetAck -> PacketAcknowledgements
Hashing -> Hashing
amAddress-f{UI()} -> amAddress-f{UI()}
SendPolicyRequest (AMSenderC)
hashingP
bv -> BitVector
ActiveMessageC
ActiveMessageAddressC
timerC
BitVectorC

Figure 39 Components diagram 1/2

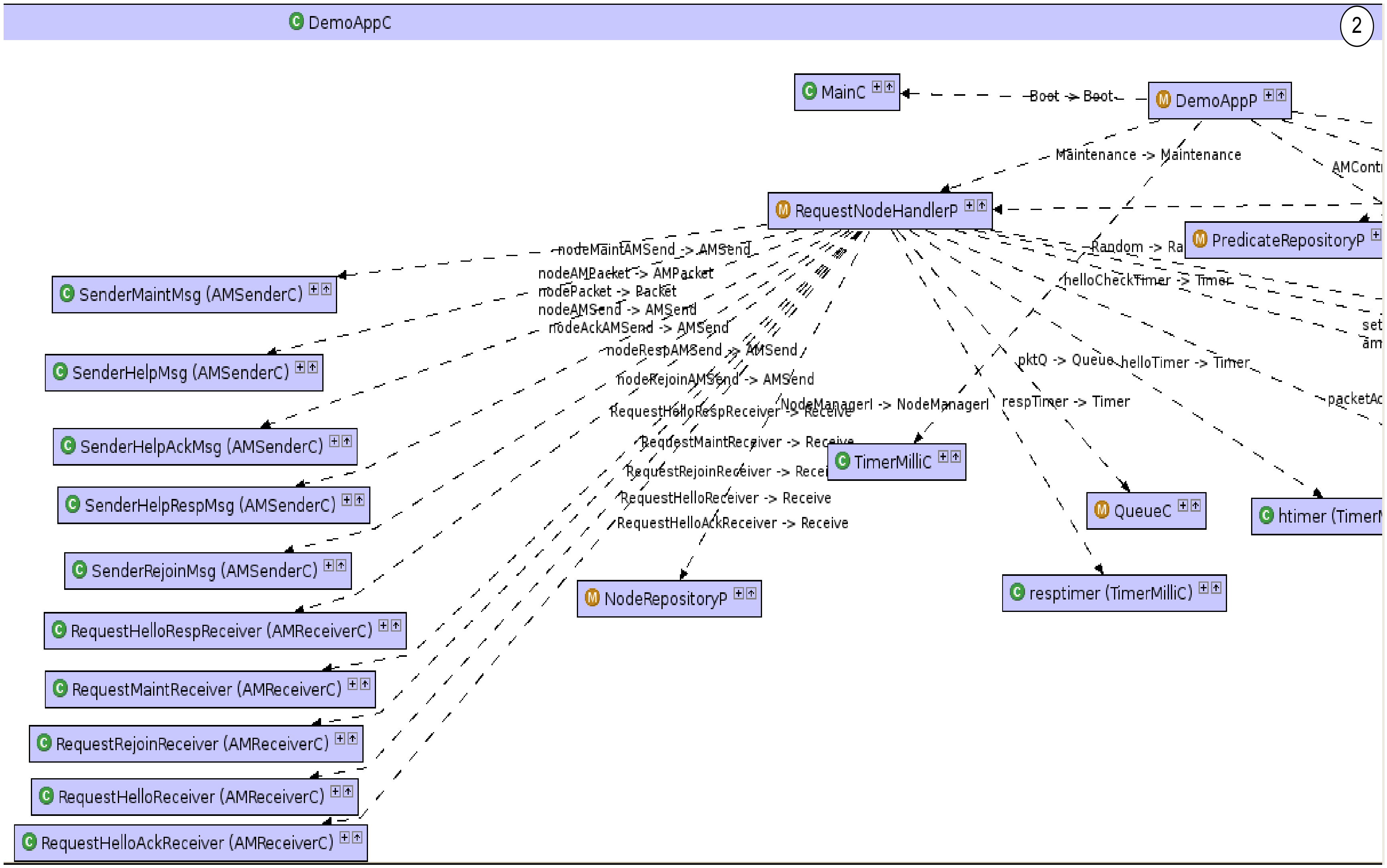

DemoAppC
2
MainC
Boot -> Boot
DemoAppP
Maintenance -> Maintenance
AMConti
RequestNodeHandlerP
PredicateRepositoryP
Random -> Ra
helloCheckTimer -> Timer
nodeMaintAMSend -> AMSend
nodeAMPacket -> AMPacket
nodePacket -> Packet
nodeAMSend -> AMSend
nodeAckAMSend -> AMSend
nodeRespAMSend -> AMSend
nodeRejoinAMSend -> AMSend
RequestHelloRespReceiver -> Receive
NodeManagerI -> NodeManagerI
RequestMaintReceiver -> Receive
RequestRejoinReceiver -> Recei
RequestHelloReceiver -> Receive
RequestHelloAckReceiver -> Receive
pktQ -> Queue
helloTimer -> Timer
respTimer -> Timer
set
am
packetAc
SenderMaintMsg (AMSenderC)
SenderHelpMsg (AMSenderC)
SenderHelpAckMsg (AMSenderC)
SenderHelpRespMsg (AMSenderC)
SenderRejoinMsg (AMSenderC)
RequestHelloRespReceiver (AMReceiverC)
RequestMaintReceiver (AMReceiverC)
RequestRejoinReceiver (AMReceiverC)
RequestHelloReceiver (AMReceiverC)
RequestHelloAckReceiver (AMReceiverC)
TimerMilliC
QueueC
htimer (TimerM
NodeRepositoryP
resptimer (TimerMilliC)

Figure 40 Components diagram 2/2

8.1 Policy management tool (Policy IDE)

The policy management tool is part of the IETF/DMTF policy architecture model [18]. This part of the architecture could not be omitted from our research as had been originally planned. During development, it proved to be a necessity for debugging, testing, and validating policy code. The capability for policy code debugging and validation does not exist in the current policy-based applications development environment. Developers must use other methods, such as static analysis, batch scripts, and emulators or simulators to perform debugging and testing through tedious and complex manual tests. The other alternative was to leverage the scope of the policy management tool to include debugging and policy testing capabilities. Clearly, the best choice was to develop a policy management tool that meets the following requirements, as shown in Figure 13:

- Integrated with the current development environment (for our work, this was Eclipse [61], TOSSIM [53], TinyOS [53], nesC [53], and Python [62]).
- User friendly with a Graphical User Interface (GUI).
- Manage policy operations.
- Control policy-based application simulation environments.
- Interactively test policy code.
- Interactively debug policy code.
- Provide real-time debugging and testing data during the policy testing and debugging process.

Figure 41 illustrates the architecture of the policy management tool in the simulation environment. However, it has been modified to work in the sensor's physical environment, as demonstrated by the Finger2IPv6 project [16], or even in the client/server environment, as demonstrated by the TOSServ project [17]. A more detailed explanation and examples of this tool can also be found in [65]. In our work, TOSSIM connects to the Policy IDE through a communications channel that is created by packet injection. A Java message interface supports the passing of messages and the creation of network packets. Messages carry control instructions for the policy-based application, such as create, delete, enable, or disable policies. Messages can also invoke wired events or other overlay network events, such as join, re-join, or maintenance.

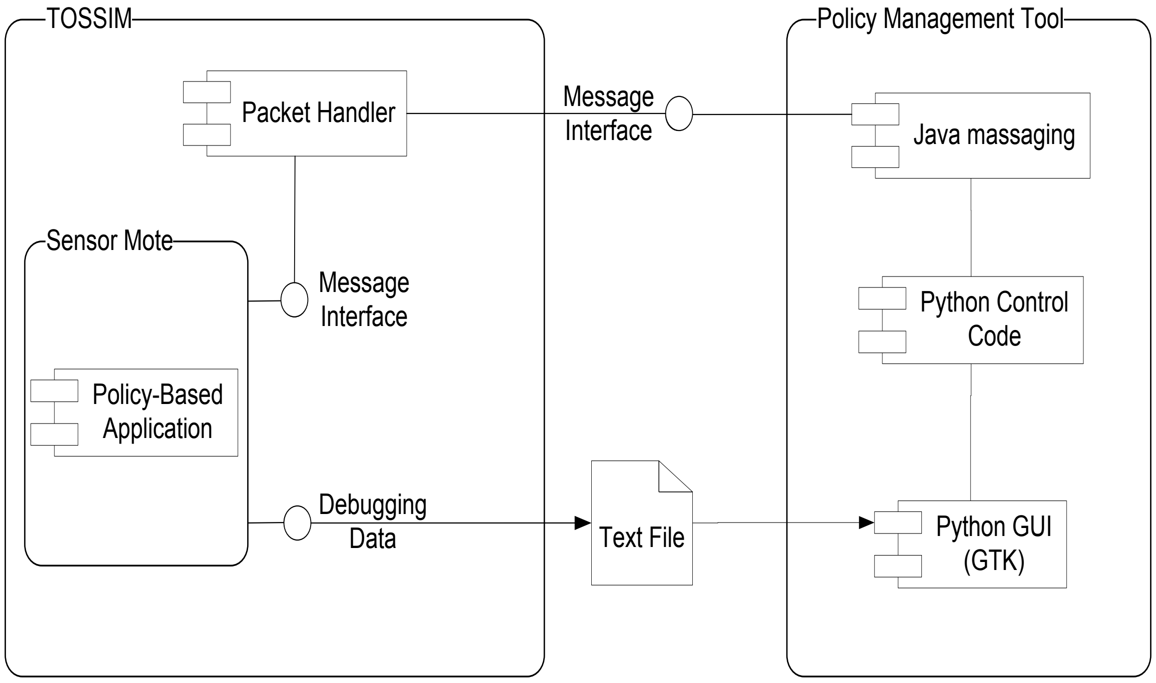


Figure 41 IDE architecture

The user interface of this tool was built with a GUI based on Python and GTK [64], which has supporting backend classes for the construction of packet fields required by policy-based applications. Messages are sent via Java to TOSSIM and then through its packet handler to the destination sensor mote. The sensor mote provides debugging and test data to the tool through a dedicated communication channel, which writes to a text file. It is possible for a developer to create many different communication channels and dedicate each one for a specific purpose, such as testing, debugging, or alert data. This approach would be very useful to separate different types of feedback messages and easily monitor policy execution. To display the text file content on the tool, the GTK text widget is pre-linked at development time to the targeted text file. After every simulation command, updated data will display in the text widget. A detailed description and illustration of this tool is in Appendix A Policy management tool (Policy IDE) interface.

8.2 Thesis validation

This thesis has validated its objectives by collecting data using simulation, observation, and analysis techniques. Our work employed the TOSSIM [53] software simulator and the AVRORA [54] software emulator. These tools are open-source software and specially designed for embedded systems and WSNs. The main advantage of using these tools is that no additional changes need to be made to the code to execute it in both the simulation environment and on a physical sensor node.

This thesis validated its objectives using the following approaches:

Increase the ability to support more policies in a WSN.

Increasing the number of policies for any sensor node implies an increase in management capabilities. To validate this objective, our research used the TOSSIM simulation software to monitor the mechanism of acquiring policies by sensor nodes, given the fact that the policies are now distributed.

Improve robustness of the distributed policy framework for a WSN.

Simulation scenarios were created to show the communication activities among WSN nodes to form the overlay network. In addition, this showed how the network works to maintain its overlay network structure.

Streamline the policy distribution processes.

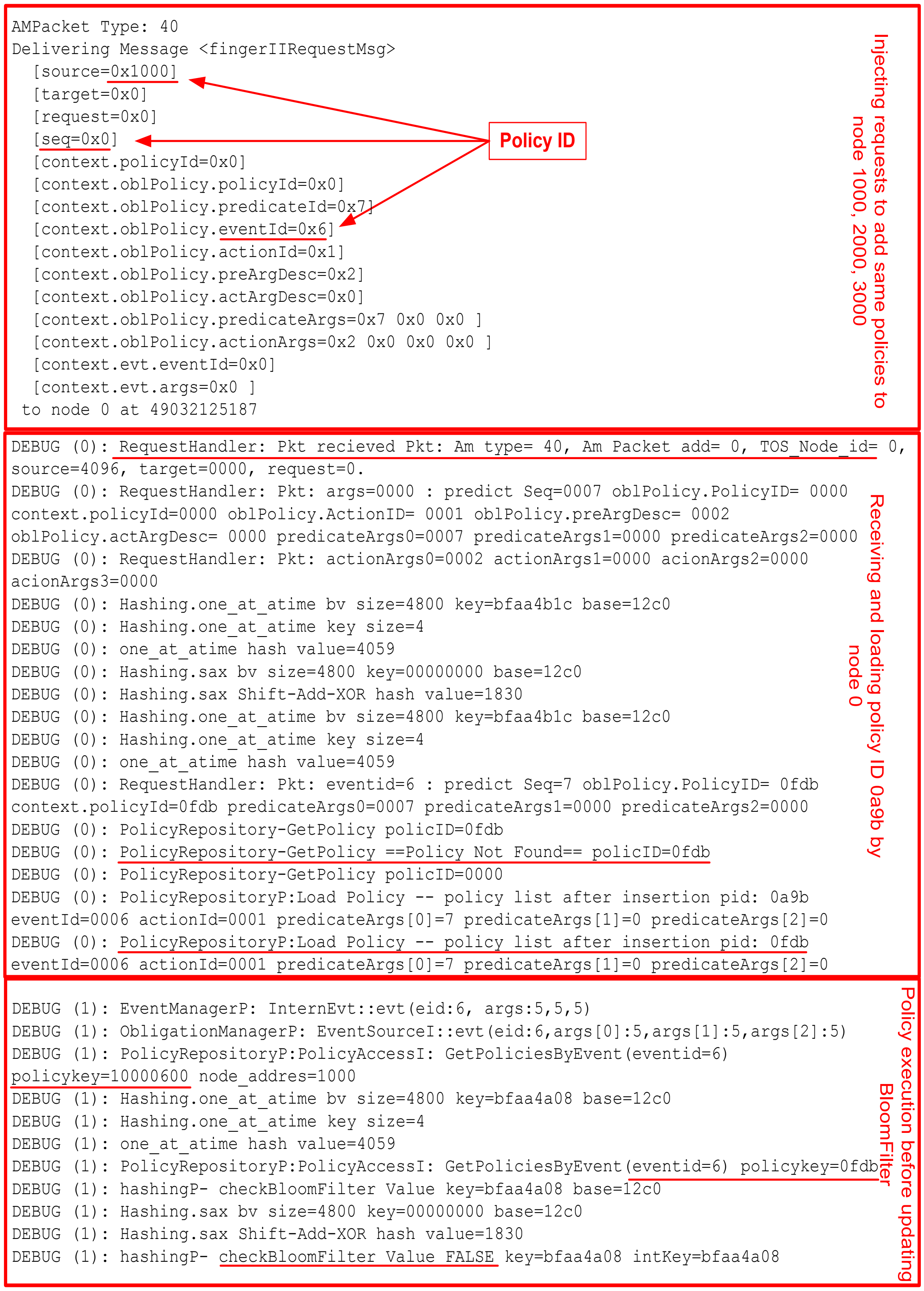
To validate this objective, the new distribution process was analyzed and compared with the existing process. A contrast table was created to summarize and contrast the two approaches. Moreover, a simulation case was created to show the policy's deployment process and policy key generation.

8.3 Increase the ability to support more policies in a WSN

Increasing the number of policies for any sensor node implies an increase in management capabilities. To validate this objective, our research used the TOSSIM simulation software to monitor the mechanism of acquiring policies by sensor nodes, given the fact that the policies are now fully distributed.

In this section, simulation results are used to illustrate the policy execution algorithm as shown in Figure 15. This simulation case consists of four sensor nodes: the Root node and three child nodes. After the network formation process, a new policy for event “Timer” on sensors 1, 2, and 3 is loaded into the network. All new policies are added to node 0 (Root) because it is considered the policy creation authority for the whole network. A load new policy request is injected into node 1000; the simulation results for the load new policy command and policy execution afterward are shown in Figure 42, Figure 43, Figure 44, and Figure 45.

The Bloom filter plays a major role in the policy execution process. Without the Bloom filter, a sensor node would have no knowledge of which policies are available in the network. Figure 42 demonstrates a similar case in which a node is not able to locate applicable policies in the network due to an outdated Bloom filter value. Therefore, the next step in the simulation after loading the new policy is to transfer the Bloom filter to the targeted nodes, as shown in Figure 43.



```
AMPacket Type: 40
Delivering Message <fingerIIRequestMsg>
  [source=0x1000]
  [target=0x0]
  [request=0x0]
  [seq=0x0]
  [context.policyId=0x0]
  [context.oblPolicy.policyId=0x0]
  [context.oblPolicy.predicateId=0x7]
  [context.oblPolicy.eventId=0x6]
  [context.oblPolicy.actionId=0x1]
  [context.oblPolicy.preArgDesc=0x2]
  [context.oblPolicy.actArgDesc=0x0]
  [context.oblPolicy.predicateArgs=0x7 0x0 0x0 ]
  [context.oblPolicy.actionArgs=0x2 0x0 0x0 0x0 ]
  [context.evt.eventId=0x0]
  [context.evt.args=0x0 ]
 to node 0 at 49032125187
```

```
DEBUG (0): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 0, TOS_Node_id= 0,
source=4096, target=0000, request=0.
DEBUG (0): RequestHandler: Pkt: args=0000 : predict Seq=0007 oblPolicy.PolicyID= 0000
context.policyId=0000 oblPolicy.ActionID= 0001 oblPolicy.preArgDesc= 0002
oblPolicy.actArgDesc= 0000 predicateArgs0=0007 predicateArgs1=0000 predicateArgs2=0000
DEBUG (0): RequestHandler: Pkt: actionArgs0=0002 actionArgs1=0000 acionArgs2=0000
acionArgs3=0000
DEBUG (0): Hashing.one_at_atime bv size=4800 key=bfaa4b1c base=12c0
DEBUG (0): Hashing.one_at_atime key size=4
DEBUG (0): one_at_atime hash value=4059
DEBUG (0): Hashing.sax bv size=4800 key=00000000 base=12c0
DEBUG (0): Hashing.sax Shift-Add-XOR hash value=1830
DEBUG (0): Hashing.one_at_atime bv size=4800 key=bfaa4b1c base=12c0
DEBUG (0): Hashing.one_at_atime key size=4
DEBUG (0): one_at_atime hash value=4059
DEBUG (0): RequestHandler: Pkt: eventid=6 : predict Seq=7 oblPolicy.PolicyID= 0fdb
context.policyId=0fdb predicateArgs0=0007 predicateArgs1=0000 predicateArgs2=0000
DEBUG (0): PolicyRepository-GetPolicy policID=0fdb
DEBUG (0): PolicyRepository-GetPolicy ==Policy Not Found== policID=0fdb
DEBUG (0): PolicyRepository-GetPolicy policID=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0a9b
eventId=0006 actionId=0001 predicateArgs[0]=7 predicateArgs[1]=0 predicateArgs[2]=0
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0fdb
eventId=0006 actionId=0001 predicateArgs[0]=7 predicateArgs[1]=0 predicateArgs[2]=0
```

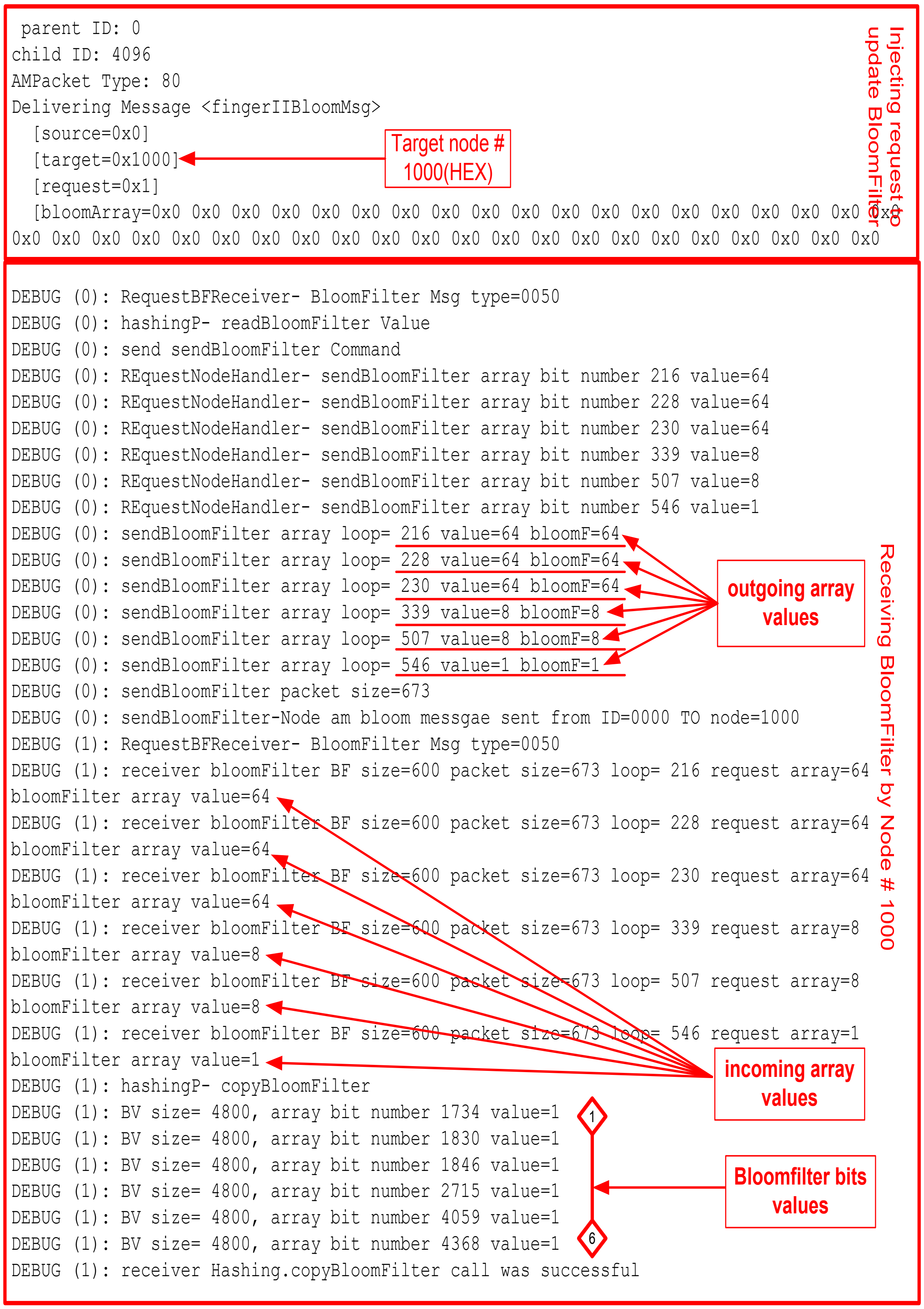
```
DEBUG (1): EventManagerP: InternEvt::evt(eid:6, args:5,5,5)
DEBUG (1): ObligationManagerP: EventSourceI::evt(eid:6,args[0]:5,args[1]:5,args[2]:5)
DEBUG (1): PolicyRepositoryP:PolicyAccessI: GetPoliciesByEvent(eventid=6)
policykey=10000600 node_addres=1000
DEBUG (1): Hashing.one_at_atime bv size=4800 key=bfaa4a08 base=12c0
DEBUG (1): Hashing.one_at_atime key size=4
DEBUG (1): one_at_atime hash value=4059
DEBUG (1): PolicyRepositoryP:PolicyAccessI: GetPoliciesByEvent(eventid=6) policykey=0fdb
DEBUG (1): hashingP- checkBloomFilter Value key=bfaa4a08 base=12c0
DEBUG (1): Hashing.sax bv size=4800 key=00000000 base=12c0
DEBUG (1): Hashing.sax Shift-Add-XOR hash value=1830
DEBUG (1): hashingP- checkBloomFilter Value FALSE key=bfaa4a08 intKey=bfaa4a08
```

Policy ID
Injecting requests to add same policies to node 1000, 2000, 3000
Receiving and loading policy ID 0a9b by node 0
Policy execution before updating BloomFilter

Figure 42 Policy execution step 1

The framework (TinyPolicy) considers node 0 as the Root of the overlay network and policy creation authority for all other nodes in the overlay network. Therefore, node 0 should always have all required policies, which implies that the Root node is a node connected to a computer or has enough power and memory to handle the required tasks as shown in Figure 37. Figure 43 shows the process of injecting a request to transfer the Bloom filter from node 0 (Root) to node 1000 (child). Figure 43 also shows the result of the process of converting the Bloom filter from a vector data structure to an array data structure so it can be embedded in a network packet to transfer it to another node.

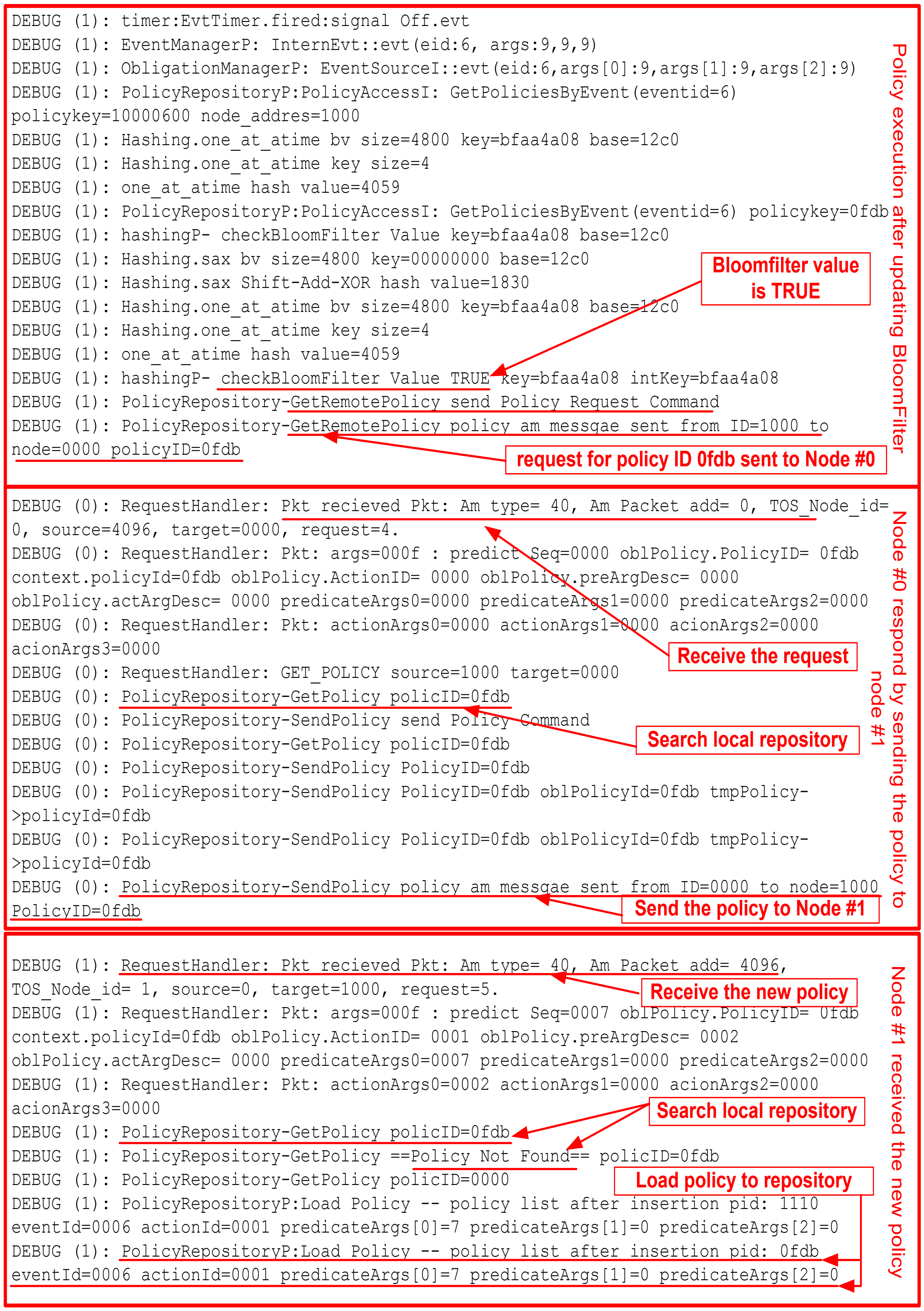
After the targeted node (node 1000) receives the updated value of the Bloom filter, it can then check for applicable policies within the overlay network. This case can be observed by comparing the results in Figure 43 and Figure 44. In Figure 43, the Bloom filter check is negative; in Figure 44, it is positive. This difference in policy execution result is due solely to the updated value of the Bloom filter in node 1000. In Figure 44, the Bloom filter check is positive, but the required policy does not exist in the local policy repository for node 1000. Therefore, node 1000 requests the missing policy from a remote node. The targeted address of the remote node is calculated based on the policy ID. Hence, the targeted address for the remote node is node 0, because the policy ID is 0fdb and node 0 is the closest matching address for that number. Thus, node 1000 requests the missing policy from node 0. When node 0 receives the request, it fetches its local policy repository and sends the requested policy to the requesting node (node 1000) as shown in Figure 44.



```
 parent ID: 0
child ID: 4096
AMPacket Type: 80
Delivering Message <fingerIIBloomMsg>
  [source=0x0]
  [target=0x1000]
  [request=0x1]
  [bloomArray=0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0

DEBUG (0): RequestBFReceiver- BloomFilter Msg type=0050
DEBUG (0): hashingP- readBloomFilter Value
DEBUG (0): send sendBloomFilter Command
DEBUG (0): REquestNodeHandler- sendBloomFilter array bit number 216 value=64
DEBUG (0): REquestNodeHandler- sendBloomFilter array bit number 228 value=64
DEBUG (0): REquestNodeHandler- sendBloomFilter array bit number 230 value=64
DEBUG (0): REquestNodeHandler- sendBloomFilter array bit number 339 value=8
DEBUG (0): REquestNodeHandler- sendBloomFilter array bit number 507 value=8
DEBUG (0): REquestNodeHandler- sendBloomFilter array bit number 546 value=1
DEBUG (0): sendBloomFilter array loop= 216 value=64 bloomF=64
DEBUG (0): sendBloomFilter array loop= 228 value=64 bloomF=64
DEBUG (0): sendBloomFilter array loop= 230 value=64 bloomF=64
DEBUG (0): sendBloomFilter array loop= 339 value=8 bloomF=8
DEBUG (0): sendBloomFilter array loop= 507 value=8 bloomF=8
DEBUG (0): sendBloomFilter array loop= 546 value=1 bloomF=1
DEBUG (0): sendBloomFilter packet size=673
DEBUG (0): sendBloomFilter-Node am bloom messgae sent from ID=0000 TO node=1000
DEBUG (1): RequestBFReceiver- BloomFilter Msg type=0050
DEBUG (1): receiver bloomFilter BF size=600 packet size=673 loop= 216 request array=64
bloomFilter array value=64
DEBUG (1): receiver bloomFilter BF size=600 packet size=673 loop= 228 request array=64
bloomFilter array value=64
DEBUG (1): receiver bloomFilter BF size=600 packet size=673 loop= 230 request array=64
bloomFilter array value=64
DEBUG (1): receiver bloomFilter BF size=600 packet size=673 loop= 339 request array=8
bloomFilter array value=8
DEBUG (1): receiver bloomFilter BF size=600 packet size=673 loop= 507 request array=8
bloomFilter array value=8
DEBUG (1): receiver bloomFilter BF size=600 packet size=673 loop= 546 request array=1
bloomFilter array value=1
DEBUG (1): hashingP- copyBloomFilter
DEBUG (1): BV size= 4800, array bit number 1734 value=1
DEBUG (1): BV size= 4800, array bit number 1830 value=1
DEBUG (1): BV size= 4800, array bit number 1846 value=1
DEBUG (1): BV size= 4800, array bit number 2715 value=1
DEBUG (1): BV size= 4800, array bit number 4059 value=1
DEBUG (1): BV size= 4800, array bit number 4368 value=1
DEBUG (1): receiver Hashing.copyBloomFilter call was successful
```

Figure 43 Policy execution step 2



Policy execution after updating BloomFilter
Bloomfilter value is TRUE
request for policy ID 0fdb sent to Node #0
Node #0 respond by sending the policy to node #1
Receive the request
Search local repository
Send the policy to Node #1
Node #1 received the new policy
Receive the new policy
Search local repository
Load policy to repository

```
DEBUG (1): timer:EvtTimer.fired:signal Off.evt
DEBUG (1): EventManagerP: InternEvt::evt(eid:6, args:9,9,9)
DEBUG (1): ObligationManagerP: EventSourceI::evt(eid:6,args[0]:9,args[1]:9,args[2]:9)
DEBUG (1): PolicyRepositoryP:PolicyAccessI: GetPoliciesByEvent(eventid=6)
policykey=10000600 node_addres=1000
DEBUG (1): Hashing.one_at_atime bv size=4800 key=bfaa4a08 base=12c0
DEBUG (1): Hashing.one_at_atime key size=4
DEBUG (1): one_at_atime hash value=4059
DEBUG (1): PolicyRepositoryP:PolicyAccessI: GetPoliciesByEvent(eventid=6) policykey=0fdb
DEBUG (1): hashingP- checkBloomFilter Value key=bfaa4a08 base=12c0
DEBUG (1): Hashing.sax bv size=4800 key=00000000 base=12c0
DEBUG (1): Hashing.sax Shift-Add-XOR hash value=1830
DEBUG (1): Hashing.one_at_atime bv size=4800 key=bfaa4a08 base=12c0
DEBUG (1): Hashing.one_at_atime key size=4
DEBUG (1): one_at_atime hash value=4059
DEBUG (1): hashingP- checkBloomFilter Value TRUE key=bfaa4a08 intKey=bfaa4a08
DEBUG (1): PolicyRepository-GetRemotePolicy send Policy Request Command
DEBUG (1): PolicyRepository-GetRemotePolicy policy am messgae sent from ID=1000 to
node=0000 policyID=0fdb
```

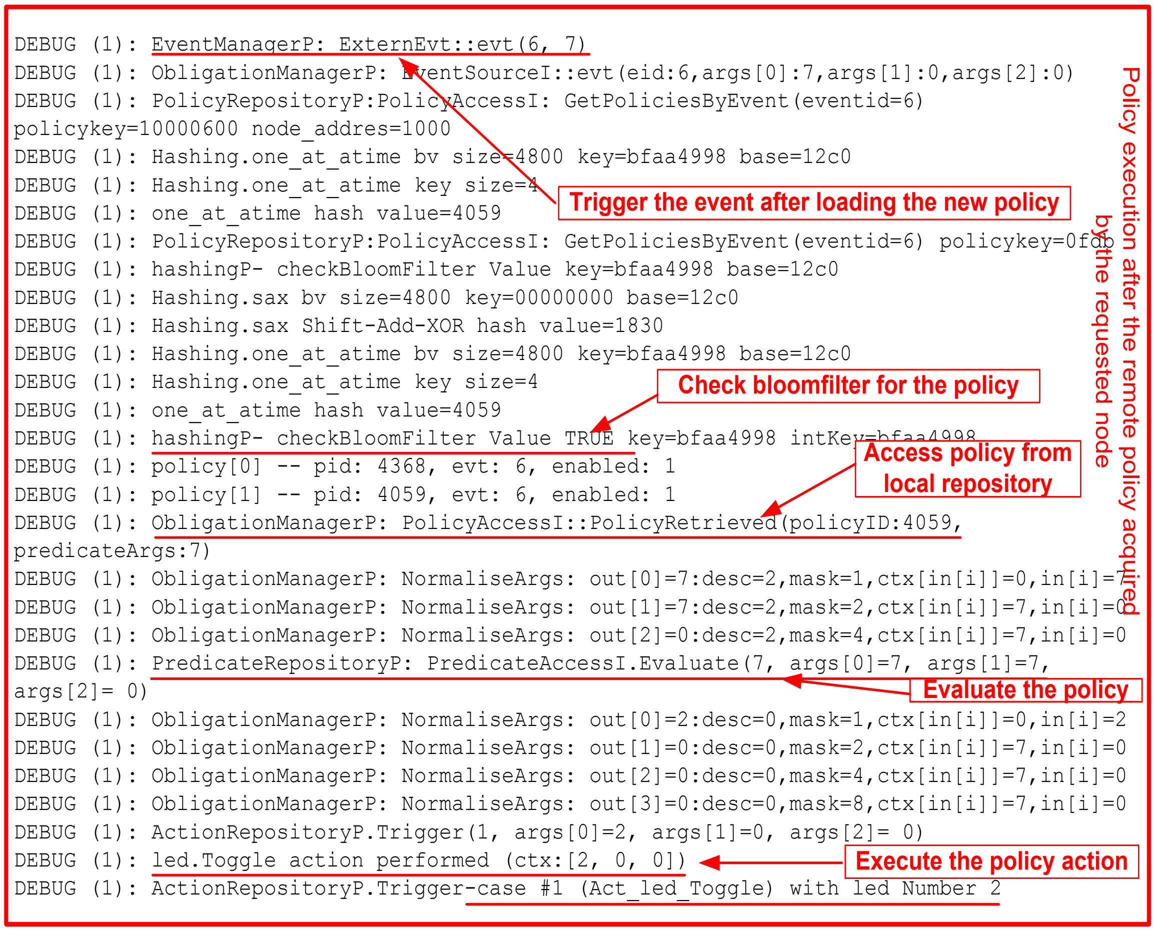
```
DEBUG (0): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 0, TOS_Node_id=
0, source=4096, target=0000, request=4.
DEBUG (0): RequestHandler: Pkt: args=000f : predict Seq=0000 oblPolicy.PolicyID= 0fdb
context.policyId=0fdb oblPolicy.ActionID= 0000 oblPolicy.preArgDesc= 0000
oblPolicy.actArgDesc= 0000 predicateArgs0=0000 predicateArgs1=0000 predicateArgs2=0000
DEBUG (0): RequestHandler: Pkt: actionArgs0=0000 actionArgs1=0000 acionArgs2=0000
acionArgs3=0000
DEBUG (0): RequestHandler: GET_POLICY source=1000 target=0000
DEBUG (0): PolicyRepository-GetPolicy policID=0fdb
DEBUG (0): PolicyRepository-SendPolicy send Policy Command
DEBUG (0): PolicyRepository-GetPolicy policID=0fdb
DEBUG (0): PolicyRepository-SendPolicy PolicyID=0fdb
DEBUG (0): PolicyRepository-SendPolicy PolicyID=0fdb oblPolicyId=0fdb tmpPolicy-
>policyId=0fdb
DEBUG (0): PolicyRepository-SendPolicy PolicyID=0fdb oblPolicyId=0fdb tmpPolicy-
>policyId=0fdb
DEBUG (0): PolicyRepository-SendPolicy policy am messgae sent from ID=0000 to node=1000
PolicyID=0fdb
```

```
DEBUG (1): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 4096,
TOS_Node_id= 1, source=0, target=1000, request=5.
DEBUG (1): RequestHandler: Pkt: args=000f : predict Seq=0007 oblPolicy.PolicyID= 0fdb
context.policyId=0fdb oblPolicy.ActionID= 0001 oblPolicy.preArgDesc= 0002
oblPolicy.actArgDesc= 0000 predicateArgs0=0007 predicateArgs1=0000 predicateArgs2=0000
DEBUG (1): RequestHandler: Pkt: actionArgs0=0002 actionArgs1=0000 acionArgs2=0000
acionArgs3=0000
DEBUG (1): PolicyRepository-GetPolicy policID=0fdb
DEBUG (1): PolicyRepository-GetPolicy ==Policy Not Found== policID=0fdb
DEBUG (1): PolicyRepository-GetPolicy policID=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 1110
eventId=0006 actionId=0001 predicateArgs[0]=7 predicateArgs[1]=0 predicateArgs[2]=0
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0fdb
eventId=0006 actionId=0001 predicateArgs[0]=7 predicateArgs[1]=0 predicateArgs[2]=0
```

Figure 44 Policy execution step 3

After the requested policy arrives at the targeted node (node 1000), the node checks its local policy repository to load the new policy if it does not exist, or overwrites it if it already exists. Following the loading of the new policy, the node triggers the applicable event to execute the new policy as shown in Figure 45. The new policy is then evaluated as shown in Figure 45.

The policy evaluation process has two main parts. The first part is the predicate evaluation, where the conditions of the policy are analyzed and evaluated. The second part is the action evaluation, where the target actions of the policy are analyzed and trigger the desired action by passing the execution to the targeted action component module. The result of the policy execution simulation is shown in Figure 45.



```
DEBUG (1): EventManagerP: ExternEvt::evt(6, 7)
DEBUG (1): ObligationManagerP: EventSourceI::evt(eid:6,args[0]:7,args[1]:0,args[2]:0)
DEBUG (1): PolicyRepositoryP:PolicyAccessI: GetPoliciesByEvent(eventid=6)
policykey=10000600 node_addres=1000
DEBUG (1): Hashing.one_at_atime bv size=4800 key=bfaa4998 base=12c0
DEBUG (1): Hashing.one_at_atime key size=4
DEBUG (1): one_at_atime hash value=4059
DEBUG (1): PolicyRepositoryP:PolicyAccessI: GetPoliciesByEvent(eventid=6) policykey=0fdb
DEBUG (1): hashingP- checkBloomFilter Value key=bfaa4998 base=12c0
DEBUG (1): Hashing.sax bv size=4800 key=00000000 base=12c0
DEBUG (1): Hashing.sax Shift-Add-XOR hash value=1830
DEBUG (1): Hashing.one_at_atime bv size=4800 key=bfaa4998 base=12c0
DEBUG (1): Hashing.one_at_atime key size=4
DEBUG (1): one_at_atime hash value=4059
DEBUG (1): hashingP- checkBloomFilter Value TRUE key=bfaa4998 intKey=bfaa4998
DEBUG (1): policy[0] -- pid: 4368, evt: 6, enabled: 1
DEBUG (1): policy[1] -- pid: 4059, evt: 6, enabled: 1
DEBUG (1): ObligationManagerP: PolicyAccessI::PolicyRetrieved(policyID:4059,
predicateArgs:7)
DEBUG (1): ObligationManagerP: NormaliseArgs: out[0]=7:desc=2,mask=1,ctx[in[i]]=0,in[i]=7
DEBUG (1): ObligationManagerP: NormaliseArgs: out[1]=7:desc=2,mask=2,ctx[in[i]]=7,in[i]=0
DEBUG (1): ObligationManagerP: NormaliseArgs: out[2]=0:desc=2,mask=4,ctx[in[i]]=7,in[i]=0
DEBUG (1): PredicateRepositoryP: PredicateAccessI.Evaluate(7, args[0]=7, args[1]=7,
args[2]= 0)
DEBUG (1): ObligationManagerP: NormaliseArgs: out[0]=2:desc=0,mask=1,ctx[in[i]]=0,in[i]=2
DEBUG (1): ObligationManagerP: NormaliseArgs: out[1]=0:desc=0,mask=2,ctx[in[i]]=7,in[i]=0
DEBUG (1): ObligationManagerP: NormaliseArgs: out[2]=0:desc=0,mask=4,ctx[in[i]]=7,in[i]=0
DEBUG (1): ObligationManagerP: NormaliseArgs: out[3]=0:desc=0,mask=8,ctx[in[i]]=7,in[i]=0
DEBUG (1): ActionRepositoryP.Trigger(1, args[0]=2, args[1]=0, args[2]= 0)
DEBUG (1): led.Toggle action performed (ctx:[2, 0, 0])
DEBUG (1): ActionRepositoryP.Trigger-case #1 (Act_led_Toggle) with led Number 2
```

Trigger the event after loading the new policy
Check bloomfilter for the policy
Access policy from local repository
Evaluate the policy
Execute the policy action
Policy execution after the remote policy acquired bythe requested node

Figure 45 Policy execution step 4

8.4 Improve robustness of the distributed policy framework for WSNs

The dynamism and robustness of the WSN was improved by creating an overlay network. The overlay network allows all nodes in the WSN to connect in a tree-structured form. This new structure has the following advantages:

1. Nodes in the new structure are able to know more about another node's relationship with the rest of the WSN nodes. Using the node ID, other nodes can calculate the ID of the parent and possible child nodes of a targeted node, which can help mitigate the risk of node errors, such as missing data or a defective node. Any node can use this approach to find a missing policy at the parent node instead of requesting it from the Root. In this case, the node needs to know the address of the defective node, which can easily be calculated from the policy key.

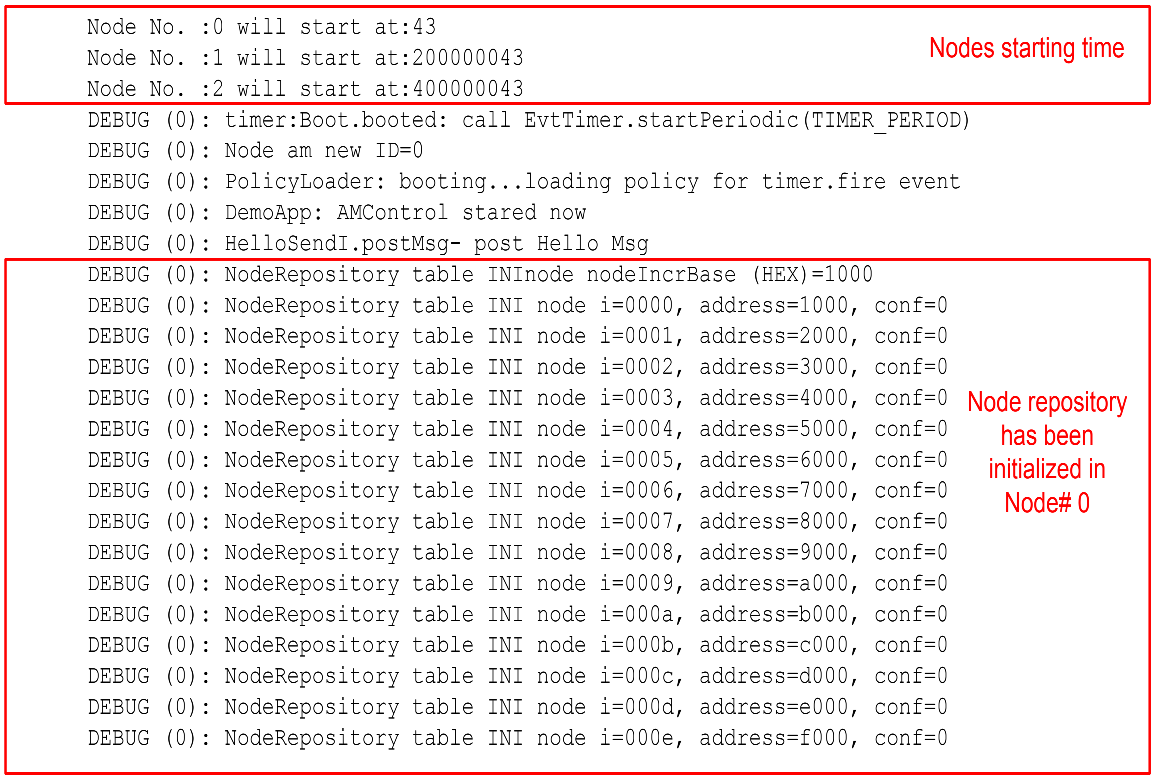
2. Nodes can join and leave the WSN without affecting the availability of any policy, because there are multiple copies of each policy distributed on different nodes within the WSN.

3. In the new structure, WSN operation will not be disrupted by any node leaving the network, regardless of whether the node left the WSN in an orderly or abrupt fashion. Missing nodes will not affect the operation or the functionality of the network, because the WSN is now decentralized and policies are distributed on multiple nodes. However, some performance degradation may occur depending on the departed node's role, as discussed in section 6.7.

4. The network can automatically discover and replace defective nodes by monitoring the policy request to the Root and assigning the defective node address to a new node, as discussed in section 6.8.

5. A new node does not have to pre-load all applicable policies into its repository. The node acquires all applicable policies from the network during operation and only when they are needed, as shown in section 7.2.

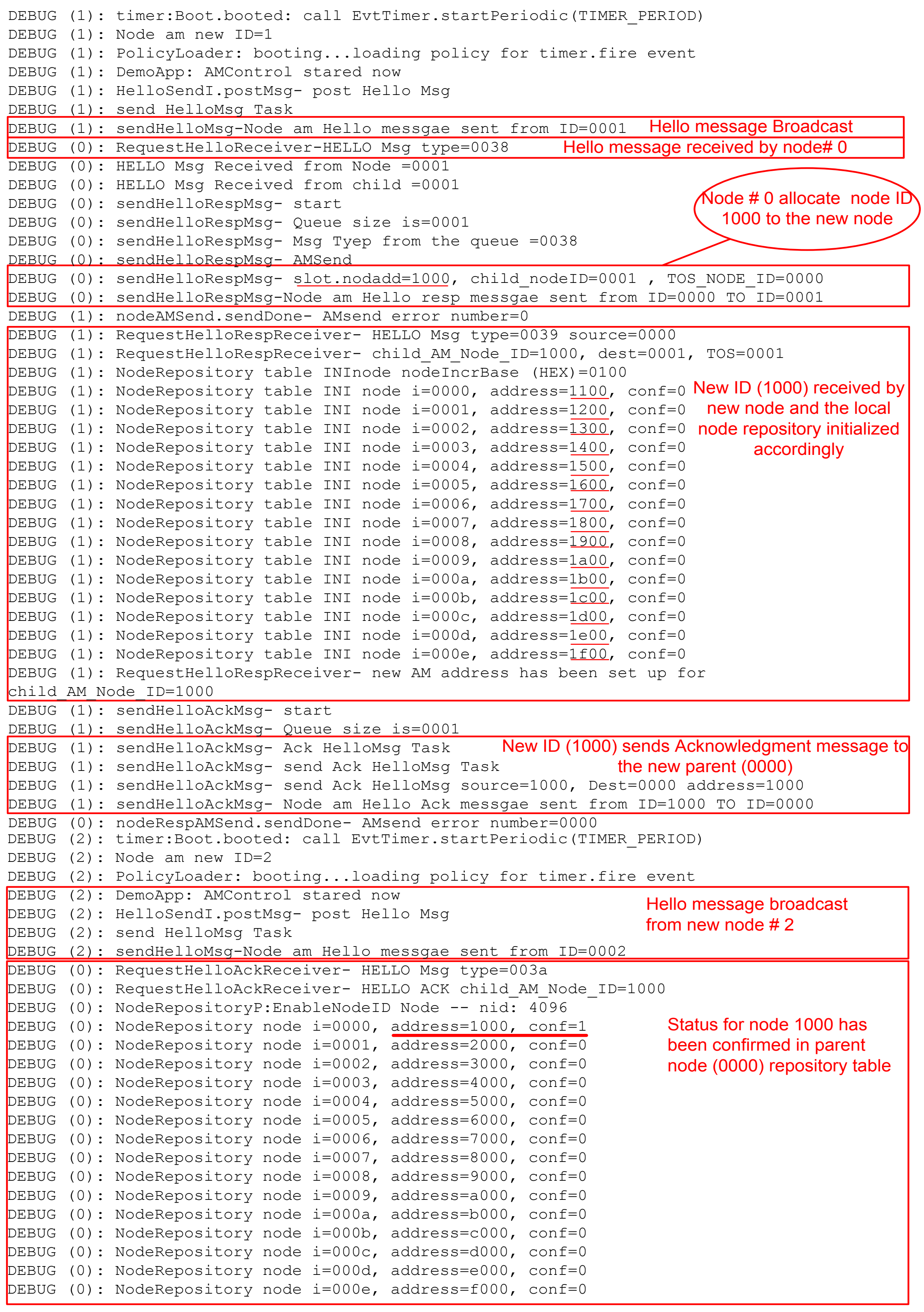
To validate this objective, two simulation scenarios were created. The first scenario shows the communication activities between WSN nodes to form the overlay network, while the second scenario shows node failure activities. The first scenario, for overlay network formation, starts by booting the Root node and initializing its node repository, as illustrated in Figure 46.



```
Node No. :0 will start at:43
Node No. :1 will start at:200000043
Node No. :2 will start at:400000043
DEBUG (0): timer:Boot.booted: call EvtTimer.startPeriodic(TIMER_PERIOD)
DEBUG (0): Node am new ID=0
DEBUG (0): PolicyLoader: booting...loading policy for timer.fire event
DEBUG (0): DemoApp: AMControl stared now
DEBUG (0): HelloSendI.postMsg- post Hello Msg
DEBUG (0): NodeRepository table INInode nodeIncrBase (HEX)=1000
DEBUG (0): NodeRepository table INI node i=0000, address=1000, conf=0
DEBUG (0): NodeRepository table INI node i=0001, address=2000, conf=0
DEBUG (0): NodeRepository table INI node i=0002, address=3000, conf=0
DEBUG (0): NodeRepository table INI node i=0003, address=4000, conf=0
DEBUG (0): NodeRepository table INI node i=0004, address=5000, conf=0
DEBUG (0): NodeRepository table INI node i=0005, address=6000, conf=0
DEBUG (0): NodeRepository table INI node i=0006, address=7000, conf=0
DEBUG (0): NodeRepository table INI node i=0007, address=8000, conf=0
DEBUG (0): NodeRepository table INI node i=0008, address=9000, conf=0
DEBUG (0): NodeRepository table INI node i=0009, address=a000, conf=0
DEBUG (0): NodeRepository table INI node i=000a, address=b000, conf=0
DEBUG (0): NodeRepository table INI node i=000b, address=c000, conf=0
DEBUG (0): NodeRepository table INI node i=000c, address=d000, conf=0
DEBUG (0): NodeRepository table INI node i=000d, address=e000, conf=0
DEBUG (0): NodeRepository table INI node i=000e, address=f000, conf=0
```

Figure 46 Overlay network formation step 1

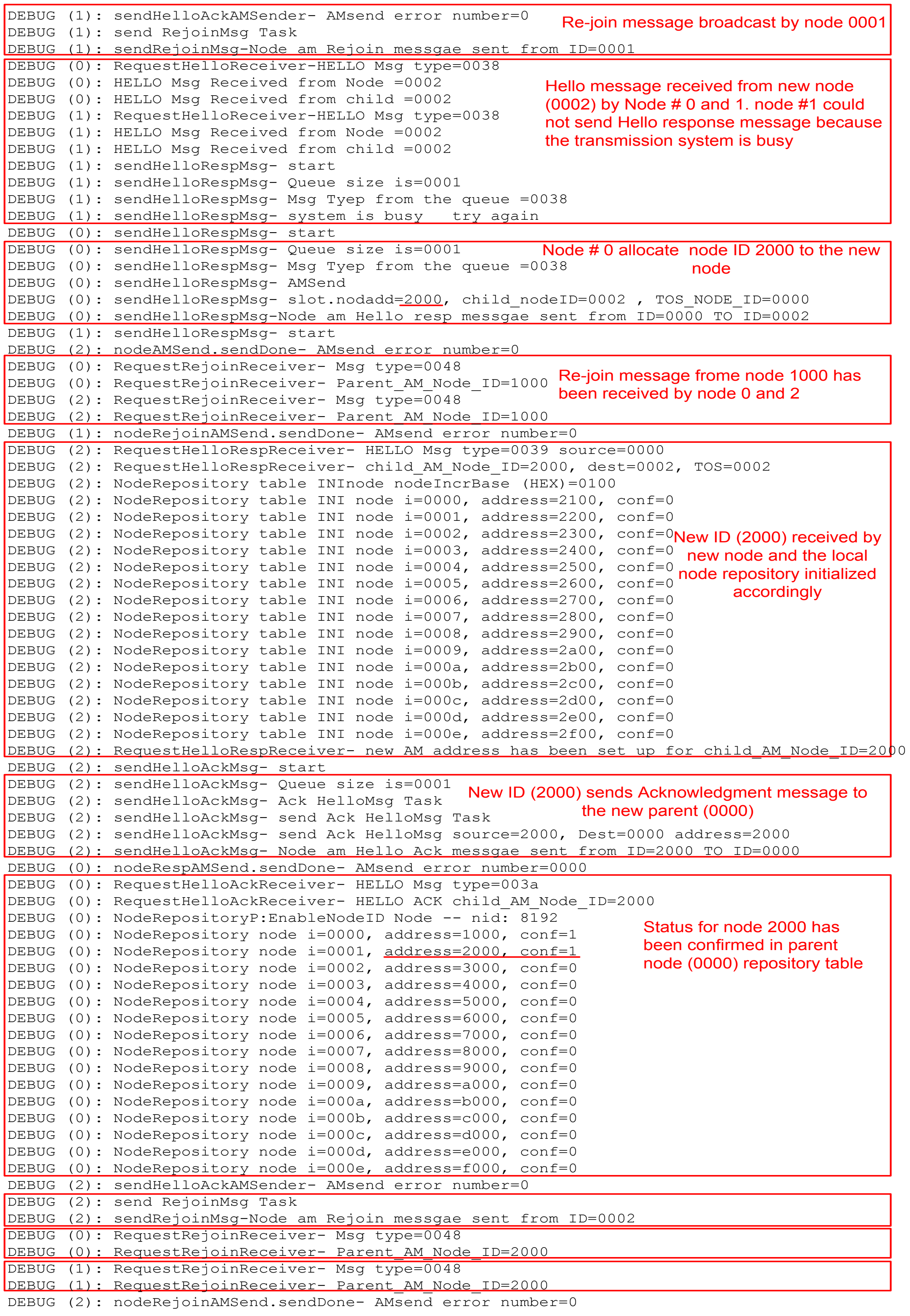
The node's repository initialization process includes setting up the repository array with all available node IDs that can be allocated to new child nodes later, as illustrated in Figure 46. Each new child node joins the network by broadcasting a Hello message. All nearby nodes will respond to the new node with a Hello Response message. The response message includes the new node ID, which has been issued by the parent node to the new child node. The node ID has been issued, but the status of that ID is still unconfirmed at the parent node repository. The status of the new node ID will be confirmed only when the parent node receives the Hello Response message from the new child node. The full communication process is illustrated in Figure 47. After the relationship between the parent and child node is established and the child node ID is configured in the new node, the child node broadcasts a Re-Join message to request reconnection of possibly existing successor nodes. This step is not important for forming a new tree, but it is important when the new node is replacing an old defective node. It will help reconnect parts of the tree that were disconnected by the departure of defective nodes. Results from a simulation of this process are shown in Figure 48, which illustrates the process of establishing join and Re-join relationships.



```
DEBUG (1): timer:Boot.booted: call EvtTimer.startPeriodic(TIMER_PERIOD)
DEBUG (1): Node am new ID=1
DEBUG (1): PolicyLoader: booting...loading policy for timer.fire event
DEBUG (1): DemoApp: AMControl stared now
DEBUG (1): HelloSendI.postMsg- post Hello Msg
DEBUG (1): send HelloMsg Task
DEBUG (1): sendHelloMsg-Node am Hello messgae sent from ID=0001
DEBUG (0): RequestHelloReceiver-HELLO Msg type=0038
DEBUG (0): HELLO Msg Received from Node =0001
DEBUG (0): HELLO Msg Received from child =0001
DEBUG (0): sendHelloRespMsg- start
DEBUG (0): sendHelloRespMsg- Queue size is=0001
DEBUG (0): sendHelloRespMsg- Msg Tyep from the queue =0038
DEBUG (0): sendHelloRespMsg- AMSend
DEBUG (0): sendHelloRespMsg- slot.nodadd=1000, child_nodeID=0001 , TOS_NODE_ID=0000
DEBUG (0): sendHelloRespMsg-Node am Hello resp messgae sent from ID=0000 TO ID=0001
DEBUG (1): nodeAMSend.sendDone- AMsend error number=0
DEBUG (1): RequestHelloRespReceiver- HELLO Msg type=0039 source=0000
DEBUG (1): RequestHelloRespReceiver- child_AM_Node_ID=1000, dest=0001, TOS=0001
DEBUG (1): NodeRepository table INInode nodeIncrBase (HEX)=0100
DEBUG (1): NodeRepository table INI node i=0000, address=1100, conf=0
DEBUG (1): NodeRepository table INI node i=0001, address=1200, conf=0
DEBUG (1): NodeRepository table INI node i=0002, address=1300, conf=0
DEBUG (1): NodeRepository table INI node i=0003, address=1400, conf=0
DEBUG (1): NodeRepository table INI node i=0004, address=1500, conf=0
DEBUG (1): NodeRepository table INI node i=0005, address=1600, conf=0
DEBUG (1): NodeRepository table INI node i=0006, address=1700, conf=0
DEBUG (1): NodeRepository table INI node i=0007, address=1800, conf=0
DEBUG (1): NodeRepository table INI node i=0008, address=1900, conf=0
DEBUG (1): NodeRepository table INI node i=0009, address=1a00, conf=0
DEBUG (1): NodeRepository table INI node i=000a, address=1b00, conf=0
DEBUG (1): NodeRepository table INI node i=000b, address=1c00, conf=0
DEBUG (1): NodeRepository table INI node i=000c, address=1d00, conf=0
DEBUG (1): NodeRepository table INI node i=000d, address=1e00, conf=0
DEBUG (1): NodeRepository table INI node i=000e, address=1f00, conf=0
DEBUG (1): RequestHelloRespReceiver- new AM address has been set up for
child_AM_Node_ID=1000
DEBUG (1): sendHelloAckMsg- start
DEBUG (1): sendHelloAckMsg- Queue size is=0001
DEBUG (1): sendHelloAckMsg- Ack HelloMsg Task
DEBUG (1): sendHelloAckMsg- send Ack HelloMsg Task
DEBUG (1): sendHelloAckMsg- send Ack HelloMsg source=1000, Dest=0000 address=1000
DEBUG (1): sendHelloAckMsg- Node am Hello Ack messgae sent from ID=1000 TO ID=0000
DEBUG (0): nodeRespAMSend.sendDone- AMsend error number=0000
DEBUG (2): timer:Boot.booted: call EvtTimer.startPeriodic(TIMER_PERIOD)
DEBUG (2): Node am new ID=2
DEBUG (2): PolicyLoader: booting...loading policy for timer.fire event
DEBUG (2): DemoApp: AMControl stared now
DEBUG (2): HelloSendI.postMsg- post Hello Msg
DEBUG (2): send HelloMsg Task
DEBUG (2): sendHelloMsg-Node am Hello messgae sent from ID=0002
DEBUG (0): RequestHelloAckReceiver- HELLO Msg type=003a
DEBUG (0): RequestHelloAckReceiver- HELLO ACK child_AM_Node_ID=1000
DEBUG (0): NodeRepositoryP:EnableNodeID Node -- nid: 4096
DEBUG (0): NodeRepository node i=0000, address=1000, conf=1
DEBUG (0): NodeRepository node i=0001, address=2000, conf=0
DEBUG (0): NodeRepository node i=0002, address=3000, conf=0
DEBUG (0): NodeRepository node i=0003, address=4000, conf=0
DEBUG (0): NodeRepository node i=0004, address=5000, conf=0
DEBUG (0): NodeRepository node i=0005, address=6000, conf=0
DEBUG (0): NodeRepository node i=0006, address=7000, conf=0
DEBUG (0): NodeRepository node i=0007, address=8000, conf=0
DEBUG (0): NodeRepository node i=0008, address=9000, conf=0
DEBUG (0): NodeRepository node i=0009, address=a000, conf=0
DEBUG (0): NodeRepository node i=000a, address=b000, conf=0
DEBUG (0): NodeRepository node i=000b, address=c000, conf=0
DEBUG (0): NodeRepository node i=000c, address=d000, conf=0
DEBUG (0): NodeRepository node i=000d, address=e000, conf=0
DEBUG (0): NodeRepository node i=000e, address=f000, conf=0
```

Hello message Broadcast
Hello message received by node# 0
Node # 0 allocate node ID 1000 to the new node
New ID (1000) received by new node and the local node repository initialized accordingly
New ID (1000) sends Acknowledgment message to the new parent (0000)
Hello message broadcast from new node # 2
Status for node 1000 has been confirmed in parent node (0000) repository table

Figure 47 Overlay network formation step 2



```
DEBUG (1): sendHelloAckAMSender- AMsend error number=0
DEBUG (1): send RejoinMsg Task
DEBUG (1): sendRejoinMsg-Node am Rejoin messgae sent from ID=0001
DEBUG (0): RequestHelloReceiver-HELLO Msg type=0038
DEBUG (0): HELLO Msg Received from Node =0002
DEBUG (0): HELLO Msg Received from child =0002
DEBUG (1): RequestHelloReceiver-HELLO Msg type=0038
DEBUG (1): HELLO Msg Received from Node =0002
DEBUG (1): HELLO Msg Received from child =0002
DEBUG (1): sendHelloRespMsg- start
DEBUG (1): sendHelloRespMsg- Queue size is=0001
DEBUG (1): sendHelloRespMsg- Msg Tyep from the queue =0038
DEBUG (1): sendHelloRespMsg- system is busy   try again
DEBUG (0): sendHelloRespMsg- start
DEBUG (0): sendHelloRespMsg- Queue size is=0001
DEBUG (0): sendHelloRespMsg- Msg Tyep from the queue =0038
DEBUG (0): sendHelloRespMsg- AMSend
DEBUG (0): sendHelloRespMsg- slot.nodadd=2000, child_nodeID=0002 , TOS_NODE_ID=0000
DEBUG (0): sendHelloRespMsg-Node am Hello resp messgae sent from ID=0000 TO ID=0002
DEBUG (1): sendHelloRespMsg- start
DEBUG (2): nodeAMSend.sendDone- AMsend error number=0
DEBUG (0): RequestRejoinReceiver- Msg type=0048
DEBUG (0): RequestRejoinReceiver- Parent_AM_Node_ID=1000
DEBUG (2): RequestRejoinReceiver- Msg type=0048
DEBUG (2): RequestRejoinReceiver- Parent_AM_Node_ID=1000
DEBUG (1): nodeRejoinAMSend.sendDone- AMsend error number=0
DEBUG (2): RequestHelloRespReceiver- HELLO Msg type=0039 source=0000
DEBUG (2): RequestHelloRespReceiver- child_AM_Node_ID=2000, dest=0002, TOS=0002
DEBUG (2): NodeRepository table INInode nodeIncrBase (HEX)=0100
DEBUG (2): NodeRepository table INI node i=0000, address=2100, conf=0
DEBUG (2): NodeRepository table INI node i=0001, address=2200, conf=0
DEBUG (2): NodeRepository table INI node i=0002, address=2300, conf=0
DEBUG (2): NodeRepository table INI node i=0003, address=2400, conf=0
DEBUG (2): NodeRepository table INI node i=0004, address=2500, conf=0
DEBUG (2): NodeRepository table INI node i=0005, address=2600, conf=0
DEBUG (2): NodeRepository table INI node i=0006, address=2700, conf=0
DEBUG (2): NodeRepository table INI node i=0007, address=2800, conf=0
DEBUG (2): NodeRepository table INI node i=0008, address=2900, conf=0
DEBUG (2): NodeRepository table INI node i=0009, address=2a00, conf=0
DEBUG (2): NodeRepository table INI node i=000a, address=2b00, conf=0
DEBUG (2): NodeRepository table INI node i=000b, address=2c00, conf=0
DEBUG (2): NodeRepository table INI node i=000c, address=2d00, conf=0
DEBUG (2): NodeRepository table INI node i=000d, address=2e00, conf=0
DEBUG (2): NodeRepository table INI node i=000e, address=2f00, conf=0
DEBUG (2): RequestHelloRespReceiver- new AM address has been set up for child_AM_Node_ID=2000
DEBUG (2): sendHelloAckMsg- start
DEBUG (2): sendHelloAckMsg- Queue size is=0001
DEBUG (2): sendHelloAckMsg- Ack HelloMsg Task
DEBUG (2): sendHelloAckMsg- send Ack HelloMsg Task
DEBUG (2): sendHelloAckMsg- send Ack HelloMsg source=2000, Dest=0000 address=2000
DEBUG (2): sendHelloAckMsg- Node am Hello Ack messgae sent from ID=2000 TO ID=0000
DEBUG (0): nodeRespAMSend.sendDone- AMsend error number=0000
DEBUG (0): RequestHelloAckReceiver- HELLO Msg type=003a
DEBUG (0): RequestHelloAckReceiver- HELLO ACK child_AM_Node_ID=2000
DEBUG (0): NodeRepositoryP:EnableNodeID Node -- nid: 8192
DEBUG (0): NodeRepository node i=0000, address=1000, conf=1
DEBUG (0): NodeRepository node i=0001, address=2000, conf=1
DEBUG (0): NodeRepository node i=0002, address=3000, conf=0
DEBUG (0): NodeRepository node i=0003, address=4000, conf=0
DEBUG (0): NodeRepository node i=0004, address=5000, conf=0
DEBUG (0): NodeRepository node i=0005, address=6000, conf=0
DEBUG (0): NodeRepository node i=0006, address=7000, conf=0
DEBUG (0): NodeRepository node i=0007, address=8000, conf=0
DEBUG (0): NodeRepository node i=0008, address=9000, conf=0
DEBUG (0): NodeRepository node i=0009, address=a000, conf=0
DEBUG (0): NodeRepository node i=000a, address=b000, conf=0
DEBUG (0): NodeRepository node i=000b, address=c000, conf=0
DEBUG (0): NodeRepository node i=000c, address=d000, conf=0
DEBUG (0): NodeRepository node i=000d, address=e000, conf=0
DEBUG (0): NodeRepository node i=000e, address=f000, conf=0
DEBUG (2): sendHelloAckAMSender- AMsend error number=0
DEBUG (2): send RejoinMsg Task
DEBUG (2): sendRejoinMsg-Node am Rejoin messgae sent from ID=0002
DEBUG (0): RequestRejoinReceiver- Msg type=0048
DEBUG (0): RequestRejoinReceiver- Parent_AM_Node_ID=2000
DEBUG (1): RequestRejoinReceiver- Msg type=0048
DEBUG (1): RequestRejoinReceiver- Parent_AM_Node_ID=2000
DEBUG (2): nodeRejoinAMSend.sendDone- AMsend error number=0
```

Re-join message broadcast by node 0001

Hello message received from new node (0002) by Node # 0 and 1. node #1 could not send Hello response message because the transmission system is busy

Node # 0 allocate node ID 2000 to the new node

Re-join message frome node 1000 has been received by node 0 and 2

New ID (2000) received by new node and the local node repository initialized accordingly

New ID (2000) sends Acknowledgment message to the new parent (0000)

Status for node 2000 has been confirmed in parent node (0000) repository table

Figure 48 Overlay network formation step 3

The second simulated scenario depicts a failure of a hosted policy node. In this case, a node that has hosted a policy fails, and the targeted node tries to get the policy from another node. As illustrated in Figure 49, node 3 with overlay address 3000 requires policy ID 1110. Node 3000 searches its local policy repository but does not find the policy. It then checks its Bloom filter array and confirms that the policy ID 1110 exists in the network. Consequently, node 3000 sends a request to the hosted policy node to get the policy. In this simulated scenario, node ID 1110 (the hosted policy node) is defective and does not acknowledge the request. Therefore, node 3000 sends another request to node 0 (Root). The next part of the simulation output in Figure 49 illustrates the communication between the Root node and the targeted node (node ID 3000) to acquire the required policy. The rest of the simulation output in Figure 49 illustrates the execution steps for the acquired policy on the targeted node (node 3000).

```
DEBUG (3): one_at_atime hash value=4368
DEBUG (3): hashingP- checkBloomFilter Value TRUE key=bff7dee8 intKey=bff7dee8

DEBUG (3): PolicyRepository-GetRemotePolicy send Policy Request Command
DEBUG (3): PolicyRepository-GetRemotePolicy policy am messgae sent from ID=3000 to node=1110 policyID=1110


DEBUG (3): policyAMSend.sendDone- NOT Acknowledged then add code to send the policy request to the root
DEBUG (3): PolicyRepository-GetRemotePolicy send Policy Request Command
DEBUG (3): PolicyRepository-GetRemotePolicy policy am messgae sent from ID=3000 to node=0000 policyID=1110
DEBUG (0): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 0, TOS_Node_id= 0
DEBUG (0): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 0, TOS_Node_id= 0, source=12288, target=0000,
request=4.
DEBUG (0): RequestHandler: Pkt: args=0011 : predict Seq=0000 oblPolicy.PolicyID= 1110 context.policyId=1110 oblPolicy.ActionID=
0000 oblPolicy.preArgDesc= 0000 oblPolicy.actArgDesc= 0000 predicateArgs0=0000 predicateArgs1=0000 predicateArgs2=0000
DEBUG (0): RequestHandler: Pkt: actionArgs0=0000 actionArgs1=0000 acionArgs2=0000 acionArgs3=0000
DEBUG (0): RequestHandler: GET_POLICY source=3000 target=0000
DEBUG (0): PolicyRepository-GetPolicy policID=1110
DEBUG (0): PolicyRepository-SendPolicy send Policy Command
DEBUG (0): PolicyRepository-GetPolicy policID=1110
DEBUG (0): PolicyRepository-SendPolicy policy am messgae sent from ID=0000 to node=3000 PolicyID=1110
DEBUG (3): policyAMSend.sendDone- AMsend error number=0
DEBUG (3): policyAMSend.sendDone- Acknowledged

DEBUG (3): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 12288, TOS_Node_id= 3
DEBUG (3): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 12288, TOS_Node_id= 3, source=0, target=3000, request=5.
DEBUG (3): RequestHandler: Pkt: args=0011 : predict Seq=0007 oblPolicy.PolicyID= 1110 context.policyId=1110 oblPolicy.ActionID= 0001
oblPolicy.preArgDesc= 0002 oblPolicy.actArgDesc= 0000 predicateArgs0=0003 predicateArgs1=0000 predicateArgs2=0000
DEBUG (3): RequestHandler: Pkt: actionArgs0=0002 actionArgs1=0000 acionArgs2=0000 acionArgs3=0000
DEBUG (3): PolicyRepository-GetPolicy policID=1110
DEBUG (3): PolicyRepository-GetPolicy ==Policy Not Found== policID=1110
DEBUG (3): PolicyRepository-GetPolicy policID=0000
DEBUG (3): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 1110 eventId=0006 actionId=0001 predicateArgs[0]=3
predicateArgs[1]=0 predicateArgs[2]=0

DEBUG (3): hashingP- checkBloomFilter Value TRUE key=bff7de78 intKey=bff7de78
DEBUG (3): policy[0] -- pid: 4368, evt: 6, enabled: 1
DEBUG (3): ObligationManagerP: PolicyAccessI::PolicyRetrieved(policyID:4368,
predicateArgs:3)
DEBUG (3): ObligationManagerP: NormaliseArgs: out[0]=3:desc=2,mask=1,ctx[in[i]]=2,in[i]=3
DEBUG (3): ObligationManagerP: NormaliseArgs: out[1]=3:desc=2,mask=2,ctx[in[i]]=3,in[i]=0
DEBUG (3): ObligationManagerP: NormaliseArgs: out[2]=0:desc=2,mask=4,ctx[in[i]]=3,in[i]=0
DEBUG (3): PredicateRepositoryP: PredicateAccessI.Evaluate(7, args[0]=3, args[1]=3, args[2]=
0)
DEBUG (3): ObligationManagerP: NormaliseArgs: out[0]=2:desc=0,mask=1,ctx[in[i]]=0,in[i]=2
DEBUG (3): ObligationManagerP: NormaliseArgs: out[1]=0:desc=0,mask=2,ctx[in[i]]=3,in[i]=0
DEBUG (3): ObligationManagerP: NormaliseArgs: out[2]=0:desc=0,mask=4,ctx[in[i]]=3,in[i]=0
DEBUG (3): ObligationManagerP: NormaliseArgs: out[3]=0:desc=0,mask=8,ctx[in[i]]=3,in[i]=0
DEBUG (3): ActionRepositoryP.Trigger(1, args[0]=2, args[1]=0, args[2]= 0)
DEBUG (3): led.Toggle action performed (ctx:[2, 0, 0])
DEBUG (3): ActionRepositoryP.Trigger-case #1 (Act_led_Toggle) with led Number 2
```

Figure 49 Node failure case

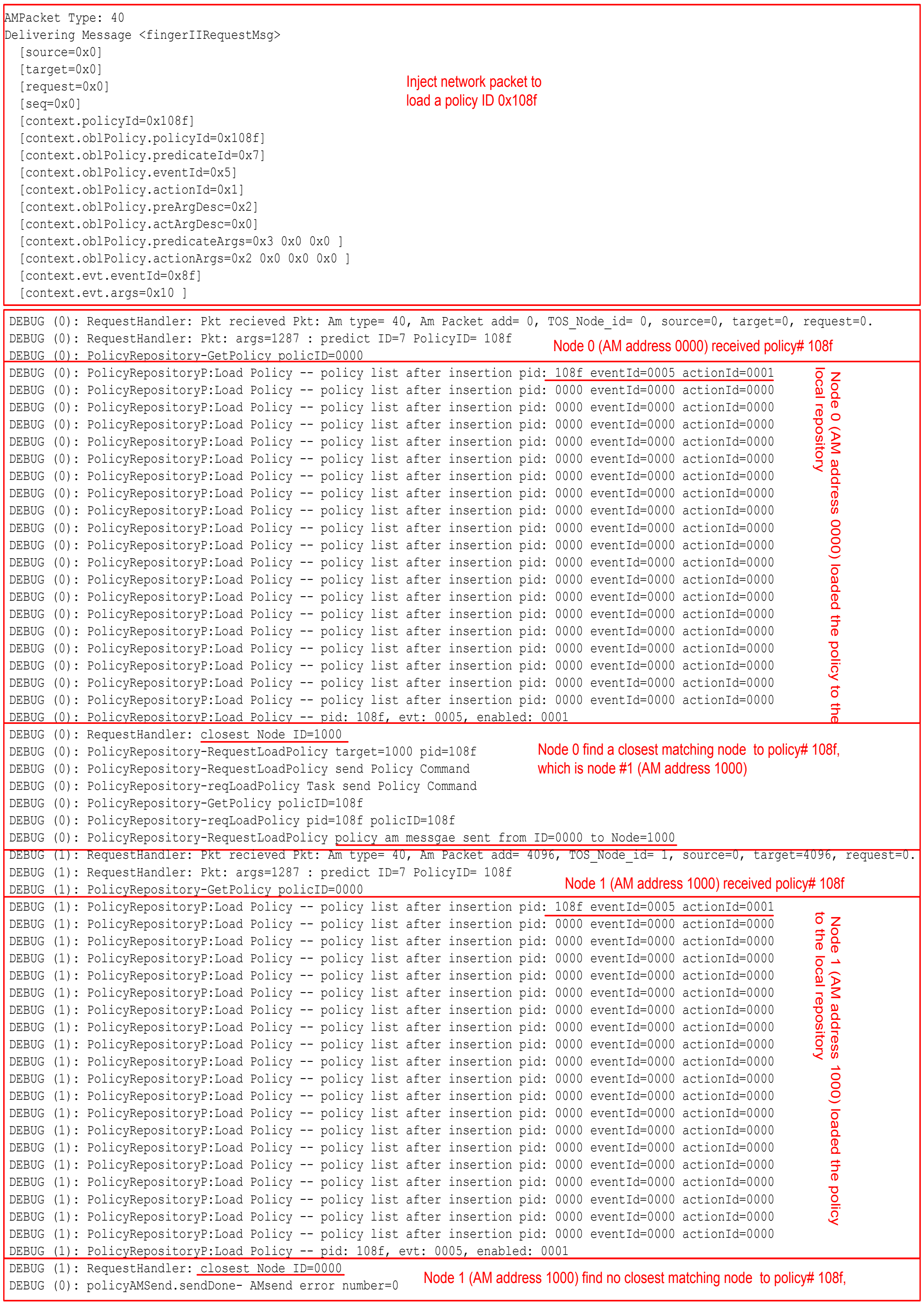
8.5 Streamline the policy distribution processes

To validate this objective, the new distribution process was analyzed and compared with the existing process. A table was created to summarize and contrast the two approaches.

Table 19 contrasts an existing policy-based platform (Finger2) with our framework (TinyPolicy). Policy deployment in TinyPolicy is dynamic and mathematically calculated, based on the policy's ID. With this approach, the new framework relieves the administrator of the burden of specifying a targeted node for every policy in the system. The new framework not only creates a fully distributed policy system but also creates a backup repository system which all nodes can access as a last resort to find missing policies. The deployment process always starts from node 0 (Root), which is the policy creation authority. Two simulation experiments have been conducted to load two new policies as shown in Figure 50 and Figure 51 respectively. The result of the first simulation experiment is shown in Figure 50. It starts by injecting a load policy message into node 0 (Root). Node 0 (Root) stores the new policy in its local policy repository and then checks its node repository for the longest matching node ID. Node 0 finds that node 1000 is the node ID that is closest to policy key 108f. Therefore, node 0 sends a copy of the new policy to the matched node, which is node 1000. The result of the second simulation experiment is shown in Figure 51. It starts by injecting a load policy message into node 0. Node 0 stores the new policy in its local policy repository and then checks its node repository for the longest matching node ID. Node 0 finds that node 2000 is the closest node ID to policy number 208f. Therefore, node 0 sends a copy of the new policy to the matched node, which is node number 2000.

The system must have only one policy creation authority, which is responsible for creating new policies and acting as a last resort for any missing policies. Any node has the capability to create new policies. However, the policy creation authority is assumed to be the backup policy repository for the whole system as well. Therefore, it must be the only node in the network acting as policy creation authority and repository backup.

It is possible to have multiple policy creation authorities in the network, but there would have to be a process to synchronize them to ensure they always have identical replicas of the policy repository. Node 0 has to have adequate resources to store all system policies, which may not be the case for other sensor nodes. There are several approaches to resolving this issue. One approach is to have the needed resources on the node itself, which means node 0 should have more resources than do the rest of the nodes in the network. Another approach is to connect node 0 to a computer through a USB connection as shown in Figure 37. Node 0 can then use the computer as a policy repository for all the nodes in the network.



```
AMPacket Type: 40
Delivering Message <fingerIIRequestMsg>
  [source=0x0]
  [target=0x0]
  [request=0x0]
  [seq=0x0]
  [context.policyId=0x108f]
  [context.oblPolicy.policyId=0x108f]
  [context.oblPolicy.predicateId=0x7]
  [context.oblPolicy.eventId=0x5]
  [context.oblPolicy.actionId=0x1]
  [context.oblPolicy.preArgDesc=0x2]
  [context.oblPolicy.actArgDesc=0x0]
  [context.oblPolicy.predicateArgs=0x3 0x0 0x0 ]
  [context.oblPolicy.actionArgs=0x2 0x0 0x0 0x0 ]
  [context.evt.eventId=0x8f]
  [context.evt.args=0x10 ]
```

Inject network packet to load a policy ID 0x108f

```
DEBUG (0): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 0, TOS_Node_id= 0, source=0, target=0, request=0.
DEBUG (0): RequestHandler: Pkt: args=1287 : predict ID=7 PolicyID= 108f
DEBUG (0): PolicyRepository-GetPolicy policID=0000
```

Node 0 (AM address 0000) received policy# 108f

```
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 108f eventId=0005 actionId=0001
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- pid: 108f, evt: 0005, enabled: 0001
```

Node 0 (AM address 0000) loaded the policy to the local repository

```
DEBUG (0): RequestHandler: closest Node ID=1000
DEBUG (0): PolicyRepository-RequestLoadPolicy target=1000 pid=108f
DEBUG (0): PolicyRepository-RequestLoadPolicy send Policy Command
DEBUG (0): PolicyRepository-reqLoadPolicy Task send Policy Command
DEBUG (0): PolicyRepository-GetPolicy policID=108f
DEBUG (0): PolicyRepository-reqLoadPolicy pid=108f policID=108f
DEBUG (0): PolicyRepository-RequestLoadPolicy policy am messgae sent from ID=0000 to Node=1000
```

Node 0 find a closest matching node to policy# 108f, which is node #1 (AM address 1000)

```
DEBUG (1): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 4096, TOS_Node_id= 1, source=0, target=4096, request=0.
DEBUG (1): RequestHandler: Pkt: args=1287 : predict ID=7 PolicyID= 108f
DEBUG (1): PolicyRepository-GetPolicy policID=0000
```

Node 1 (AM address 1000) received policy# 108f

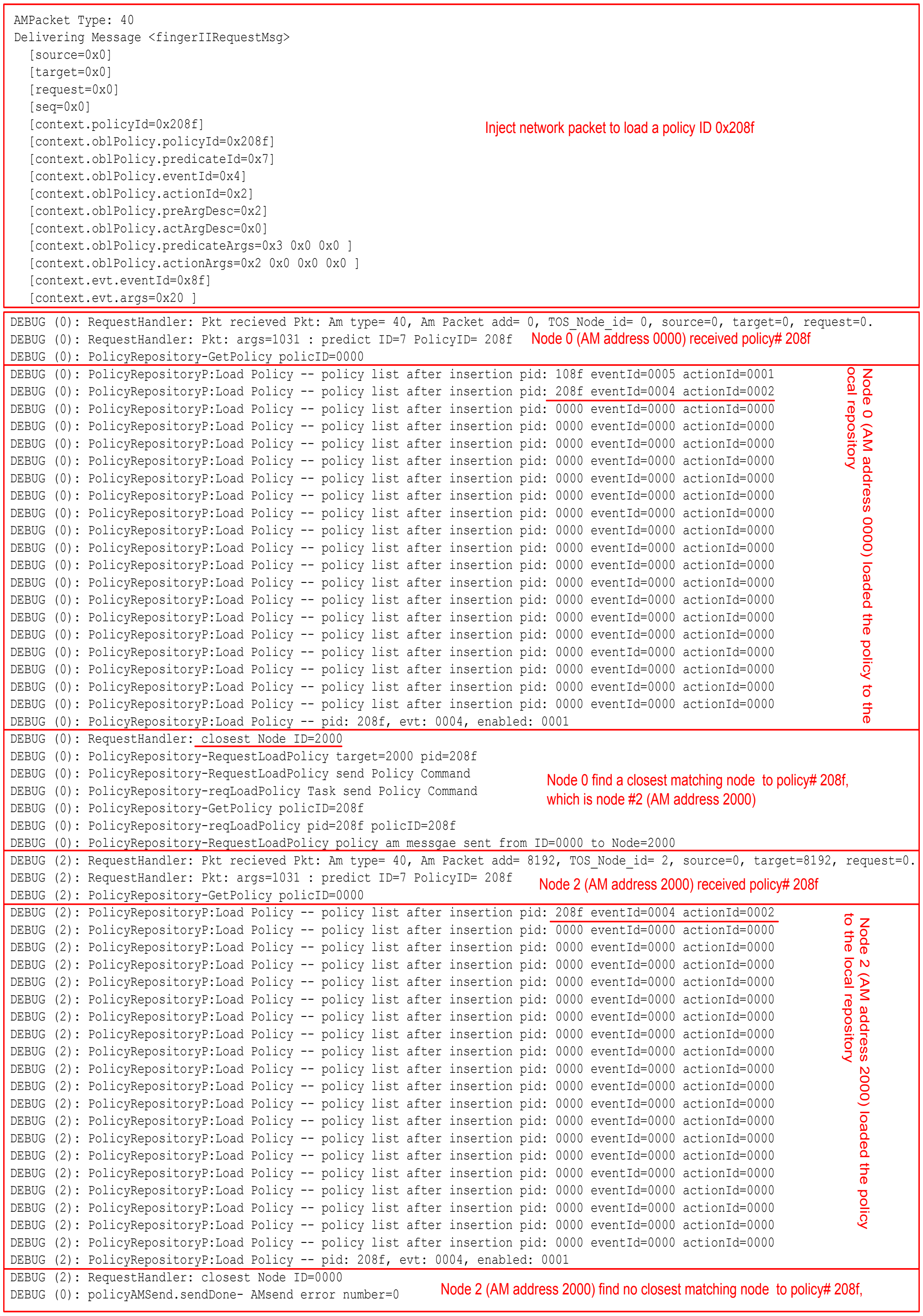
```
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 108f eventId=0005 actionId=0001
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (1): PolicyRepositoryP:Load Policy -- pid: 108f, evt: 0005, enabled: 0001
```

Node 1 (AM address 1000) loaded the policy to the local repository

```
DEBUG (1): RequestHandler: closest Node ID=0000
DEBUG (0): policyAMSend.sendDone- AMsend error number=0
```

Node 1 (AM address 1000) find no closest matching node to policy# 108f,

Figure 50 Loading first policy 108f



```
AMPacket Type: 40
Delivering Message <fingerIIRequestMsg>
  [source=0x0]
  [target=0x0]
  [request=0x0]
  [seq=0x0]
  [context.policyId=0x208f]
  [context.oblPolicy.policyId=0x208f]
  [context.oblPolicy.predicateId=0x7]
  [context.oblPolicy.eventId=0x4]
  [context.oblPolicy.actionId=0x2]
  [context.oblPolicy.preArgDesc=0x2]
  [context.oblPolicy.actArgDesc=0x0]
  [context.oblPolicy.predicateArgs=0x3 0x0 0x0 ]
  [context.oblPolicy.actionArgs=0x2 0x0 0x0 0x0 ]
  [context.evt.eventId=0x8f]
  [context.evt.args=0x20 ]
```

Inject network packet to load a policy ID 0x208f

```
DEBUG (0): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 0, TOS_Node_id= 0, source=0, target=0, request=0.
DEBUG (0): RequestHandler: Pkt: args=1031 : predict ID=7 PolicyID= 208f
DEBUG (0): PolicyRepository-GetPolicy policID=0000
```

Node 0 (AM address 0000) received policy# 208f

```
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 108f eventId=0005 actionId=0001
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 208f eventId=0004 actionId=0002
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (0): PolicyRepositoryP:Load Policy -- pid: 208f, evt: 0004, enabled: 0001
```

Node 0 (AM address 0000) loaded the policy to the ocal repository

```
DEBUG (0): RequestHandler: closest Node ID=2000
DEBUG (0): PolicyRepository-RequestLoadPolicy target=2000 pid=208f
DEBUG (0): PolicyRepository-RequestLoadPolicy send Policy Command
DEBUG (0): PolicyRepository-reqLoadPolicy Task send Policy Command
DEBUG (0): PolicyRepository-GetPolicy policID=208f
DEBUG (0): PolicyRepository-reqLoadPolicy pid=208f policID=208f
DEBUG (0): PolicyRepository-RequestLoadPolicy policy am messgae sent from ID=0000 to Node=2000
```

Node 0 find a closest matching node to policy# 208f, which is node #2 (AM address 2000)

```
DEBUG (2): RequestHandler: Pkt recieved Pkt: Am type= 40, Am Packet add= 8192, TOS_Node_id= 2, source=0, target=8192, request=0.
DEBUG (2): RequestHandler: Pkt: args=1031 : predict ID=7 PolicyID= 208f
DEBUG (2): PolicyRepository-GetPolicy policID=0000
```

Node 2 (AM address 2000) received policy# 208f

```
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 208f eventId=0004 actionId=0002
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- policy list after insertion pid: 0000 eventId=0000 actionId=0000
DEBUG (2): PolicyRepositoryP:Load Policy -- pid: 208f, evt: 0004, enabled: 0001
```

Node 2 (AM address 2000) loaded the policy to the local repository

```
DEBUG (2): RequestHandler: closest Node ID=0000
DEBUG (0): policyAMSend.sendDone- AMsend error number=0
```

Node 2 (AM address 2000) find no closest matching node to policy# 208f,

Figure 51 Loading second policy 208f

8.6 Framework limitations and constraints

During simulation and implementation, the framework exhibited some limitations and constraints which may restrict its features and operations. However, these limitations can be overcome by modifying the network setup or making changes to the system's code. Following are a list of the main limitations and constraints.

Network topology: The network topology plays a major role in getting a node ID. Each node has a limited number of addresses that can be given to a new node joining the network. In some cases where nodes are clustered in a very small area, the parent node can exhaust all its available addresses and reject any new join requests, as illustrated in Figure 52. The limitation is that the new node keeps getting the reject message from the parent node as long as the closest node to it is a parent node whose node repository is full. In such cases, one solution is to move the new node away from that parent node so it can get the new address from another parent node. Another solution is to increase the node repository capacity in the system, which may require a code change in the system to increase the address space, increase the overlay tree levels, or both.

```
(22): sendHelloAckMsg- Node am Hello Ack messgae sent from ID=f000 TO ID=0000
DEBUG (0): nodeRespAMSend.sendDone- AMsend error number=0000
DEBUG (0): RequestHelloAckReceiver- HELLO Msg type=003a
DEBUG (0): RequestHelloAckReceiver- HELLO ACK child_AM_Node_ID=f000
DEBUG (0): NodeRepositoryP:EnableNodeID Node -- nid: 61440
DEBUG (0): NodeRepository node i=0000, address=1000, conf=1
DEBUG (0): NodeRepository node i=0001, address=2000, conf=1
DEBUG (0): NodeRepository node i=0002, address=3000, conf=1
DEBUG (0): NodeRepository node i=0003, address=4000, conf=1
DEBUG (0): NodeRepository node i=0004, address=5000, conf=1
DEBUG (0): NodeRepository node i=0005, address=6000, conf=1
DEBUG (0): NodeRepository node i=0006, address=7000, conf=1
DEBUG (0): NodeRepository node i=0007, address=8000, conf=1
DEBUG (0): NodeRepository node i=0008, address=9000, conf=1
DEBUG (0): NodeRepository node i=0009, address=a000, conf=1
DEBUG (0): NodeRepository node i=000a, address=b000, conf=1
DEBUG (0): NodeRepository node i=000b, address=c000, conf=1
DEBUG (0): NodeRepository node i=000c, address=d000, conf=1
DEBUG (0): NodeRepository node i=000d, address=e000, conf=1
DEBUG (0): NodeRepository node i=000e, address=f000, conf=1
```

Node 0 has a full node repository
Last node was ID 22 (16 Hex)

```
DEBUG (23): send HelloMsg Task
DEBUG (23): sendHelloMsg-Node am Hello messgae sent from ID=0017
```

Node 23 (17 Hex) send Hello message

```
DEBUG (0): HELLO Msg Received from Node =0017
```

Node 0 received the Hello message

```
DEBUG (0): sendHelloRespMsg-Node array is FULL or has not been initialized
```

Node 0 can not give a new address

Figure 52 Network topology limitation

Capacity of repositories: The framework uses repositories to store the node overlay address and policy. An array structure is used to implement policy and node repositories. There are naturally some hardware and software limitations on how large these repositories can be for a sensor node. During simulation, it appears that compilation for TOSSIM has more relaxed rules than compilation for a mote. To determine limitations on policy repository capacity, a number of compilation trials for TOSSIM and a mote were conducted. The program was successfully compiled for TOSSIM simulation and a Micaz mote with a large repository size of 20,000 policies and a 15-node repository capacity. However, the program did not compile for a Micaz mote with a repository size of 1,928 policies and a 15-node repository capacity, as shown in Figure 53.

```
//opt/tinyos-2.1.0/tos/system/BitVectorC.nc(DemoAppC.BitVectorC):78: warning:   non-atomic read
/opt/tinyos-2.1.0/tos/system/BitVectorC.nc(DemoAppC.BitVectorC):83: warning:   non-atomic r/w
../src/core/PolicyRepositoryP.nc:70: error: size of array 'PolicyRepositoryP$policies' is too large
```

Figure 53 Compilation error when policy repository size reached 1928

A number of compilation attempts were conducted to determine the maximum policy repository limit. The experimental results are shown in Table 17, and Figure 54 illustrates the result of the compilation experiments for the policy repository limitation.

Table 17 Policy Repository Maximum Limit Experiment

Policy Repository size	ROM size	RAM size
20	22592	7006
40	22592	7346
60	22592	7686
80	22592	8026
100	22592	8366
200	22600	10066
300	22600	11766
400	22600	13466
1000	22600	23666
1900	22600	38966
1910	22600	39136
1920	22600	39306
1925	22600	39391
1926	22600	39408
1927	22600	39425
1928	**error**	**error**

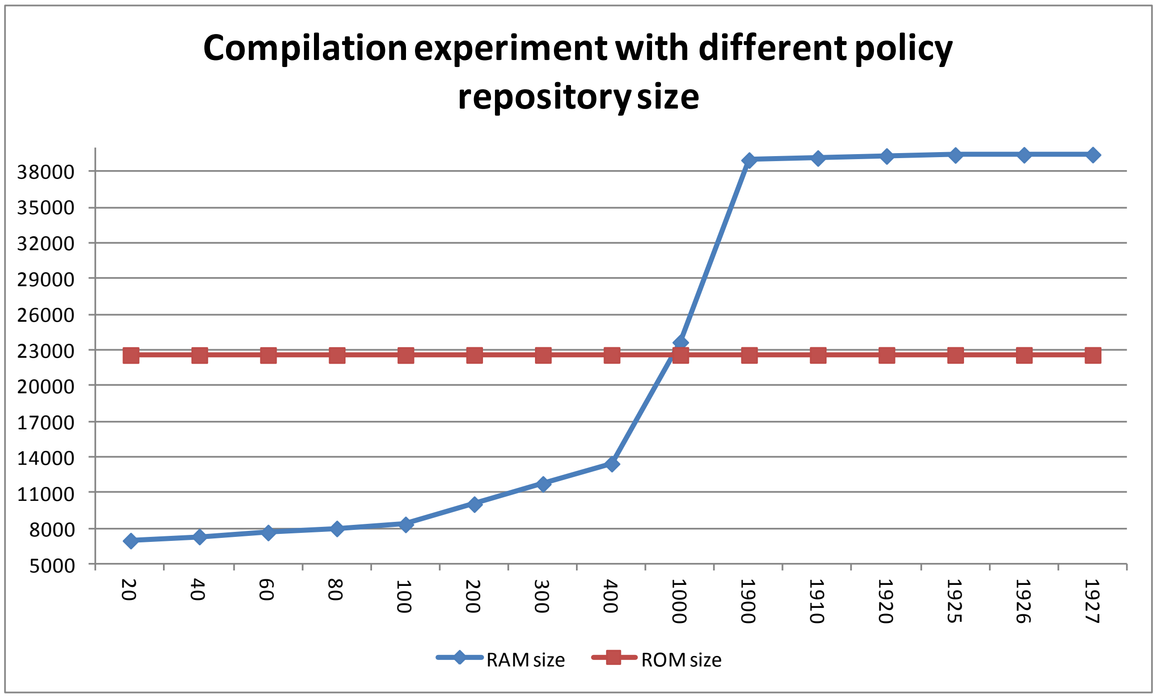


Figure 54 Compilation experiment with different policy repository size

To determine limitations on node repository capacity, a number of compilation trials for TOSSIM and a mote were conducted. The program was successfully compiled for TOSSIM simulation and a Micaz mote with a large repository size of 20,000 nodes and a 20-policy repository capacity. However, the program did not compile for a Micaz mote with a repository size of 10,922 nodes and a 20-policy repository capacity, as shown in Figure 55.

```
/opt/tinyos-2.1.0/tos/system/BitVectorC.nc(DemoAppC.BitVectorC):78: warning:   non-atomic read
/opt/tinyos-2.1.0/tos/system/BitVectorC.nc(DemoAppC.BitVectorC):83: warning:   non-atomic r/w
../src/core/NodeRepositoryP.nc:39: error: size of array 'NodeRepositoryP$nodes' is too large
```

Figure 55 Compilation error when node repository size reached 10922

Several attempts were made to get the maximum node repository limits. The experimental results are shown in Table 18, and Figure 56 illustrates the summary result of the compilation experiments for the policy repository limits.

Table 18 Node Repository Maximum Limit Experiment

Node Repository size	ROM size	RAM size
15	22592	7006
30	22592	7051
60	22592	7141
120	22592	7321
240	22592	7681
500	22598	8461
1000	22598	9961
2000	22598	12961
4000	22598	18961
8000	22598	30961
10000	22598	36961
10900	22598	39661
10920	22598	39721
10921	22598	39724
10922	22598	39727
10923	error	error

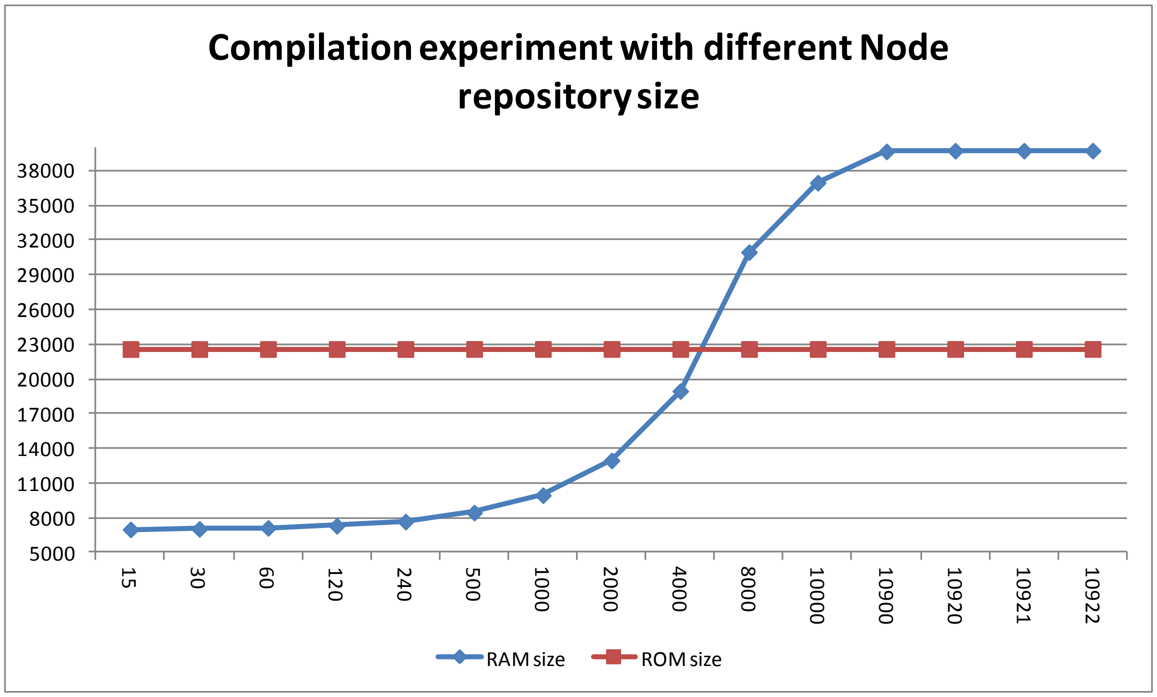


Figure 56 Compilation experiment with different node repository size

Network size: This thesis implementation chose a three-level tree structure, which can accommodate up to 3,616 nodes as illustrated in Table 12. A larger network size would require a change to the overlay address space. Another option is to increase the number of tree levels in the overlay network.

Race condition: This case appears when two or more motes try to acquire the same node ID. The system recovers from this condition by allowing only nodes with unique addresses to be in the overlay tree. The other node has to be restarted but it will not affect the system operation if it stays on, as it becomes a duplicate node in the network.

Unsuccessful acknowledgment of the node ID: This case appears when a node fails to send an acknowledgment of its new address to its parent node or the parent node fails to receive it. The system recovers from this condition by allowing only nodes with unique addresses to be in the overlay tree. Thus, the new node has a valid node ID, but it is not part of the overlay network, and its address will be given to the first new node joining the network.

Chapter 9 Conclusions and Future Work

Policy-based systems exist in various implementation domains, such as data center management, security, privacy, and computer network management. In the future, policy-based systems are expected to play an even more important role in the Internet of Things (IoT), due to their great ability to abstract hardware complexity from a system's users. Policy-based management will help WSNs resolve the challenging issues of governing and controlling embedded devices. For these existing and future implementation domains, there is a need to innovate a new policy-based engine that is lightweight, dynamic, decentralized yet well connected, and capable of handling numbers of policies beyond a device's local physical capacity. Another benefit of such a model is that it will push the most widely used policies onto the device as opposed to leaving them on the gateway node, as it is the case in existing systems.

A new distributed policy framework for WSNs was successfully created and tested. The new framework supports many new features, such as dynamically distributed policies by mathematically calculating the policy key using a hashing algorithm, building an overlay network with a tree structure over a WSN, decentralized policy-based managing which does not rely entirely on a central or local policy repository and yet is well connected and dynamic, just to name a few. Our first objective was to extend the WSN management functionalities beyond conventional policy management systems like Finger/Finger2 by increasing the number of policies that can be individually stored in any sensor node. Section 8.3 shows a simulation case where a node with a policy missing from its local policy repository can still access the missing policy from remote nodes within the WSN. This case confirms that the number of policies available for any sensor node has been increased beyond the sensor's physical capacity to the maximum capacity of the whole WSN.

The overlay network provides information about the topology of the WSN, since new nodes will normally connect to a nearby node, which provides the approximate node location and distance from other nodes. The topology information of the WSN can also be used for administrative purposes, such as using policies to direct the flow of sensing data to a targeted node that is closer to the source node. The overlay network over the WSN also improves distributed policy system dynamism and robustness, which allows nodes to establish P2P connections and find required objects (policies) mathematically without a centralized repository index system. Likewise, under the new system, policies become more accessible, and their availability improves, due to the fact that policies are now dynamically distributed and can be located mathematically within the WSN.

Moreover, if for any reason a hosted node becomes defective, policies can be retrieved from other nodes. Finally, the new policy framework conceals the complexity of administering the policy distribution process from the users by creating a dynamic mechanism for hosting and looking up a required policy within the WSN with minimal user intervention.

Many new algorithms and modified versions of existing algorithms have been implemented in this new framework, particularly those related to hashing and Bloom filter algorithms. The Bloom filter has been widely used in various domains, especially database management systems. Section 5.4 shows that the Bloom filter can help the policy framework check the existence of a policy within the WSN with little computation time, minimal energy, and limited traffic.

While policy-based management enhances the autonomous behavior of WSNs, it adds to the complexity of the debugging process. To meet this challenge, a new tool, Policy IDE, was developed to control the simulation environment for the WSN in conjunction with a graphical user interface and packet injection mechanism. As a result, interactive simulations, granular unit testing, interactive debugging, and execution tracing are feasible for policy-based applications. This augments and streamlines the policy development process in particular, by enabling developers to develop, deploy, and test policies before they are used in production environments and on hardware sensor motes. As discussed in Chapter 8 Validation of TinyPolicy through implementation in TinyOS, these new features come with an expected overhead in program size and performance, compared with conventional policy management systems like Finger/Finger2.

This thesis applies the concept of sharing node resources to achieve the framework objectives. Table 19 shows the contrast between Finger2 (the existing platform) and TinyPolicy (the new framework of this thesis) for implementing a policy-based platform in WSN.

Table 19 Contrast Between TinyPolicy and Finger2

Attribute	TinyPolicy	Finger2
Max. Number of policies	The total capacity of the WSN network	20 per node
Policy Storage	Fully distributed	Local/node
Policy Key	System-generated number	Arbitrary number
Policy Deployment	Mathematically calculated (auto)	Targeted-manual
Network Type	Overlay network	Physical network
Node Deployment	Nodes with similar functionalities are exchangeable	Nodes with similar functionalities are exchangeable if they are pre-loaded with all applicable policies
Node Failure/policy availability	Policies will be available from other nodes	N/A
Node Failure/policy access performance	Relatively slower	N/A
Application domain	Framework can be used for content-based applications	With significant changes, it may be used for content-based applications

Further improvements and enhancements to the framework are possible. Two topics for future research are the following:

TinyPAST: A software component to be built on top of PolicyP2P. It will be responsible for replicating local policies on multiple remote nodes. TinyPAST will increase system persistence and overcome the problem of nodes leaving the network with no prior warning.

TinySCRIBE: Another software component to be built on top of PolicyP2P. It will be responsible for creating, participating, communicating, and maintaining the necessary topics (events) on the local node. With TinySCRIBE, it will be possible to create more complex policy cases in which various events on various remote nodes may collaborate through a series of executing policies to achieve desired results.

References

[1] J. Gutiérrez, J. F. Villa-medina, A. Nieto-garibay, and M. Á. Porta-gándara, "Automated Irrigation System Using a Wireless Sensor Network and GPRS Module," *IEEE Trans. Instrum. Meas.*, vol. 63, no. 1, pp. 166–176, 2014.

[2] N. Matthys and W. Joosen, "Towards policy-based management of sensor networks," *Proc. 3rd Int. Work. Middlew. Sens. networks - MidSens '08*, pp. 13–18, 2008.

[3] W. Han, Z. Fang, and L. T. Yang, "Collaborative Policy Administration," *IEEE Trans. PARALLEL Distrib. Syst.*, vol. 25, no. 2, pp. 498–507, 2014.

[4] J. Strassner, *Policy-based Network Management: Solutions for the Next Generation*. San Francisco: Morgan Kaufmann, 2004, p. 516.

[5] W. Zhang and H. Xu, "A Policy Based Wireless Sensor Network Management Architecture," *2010 Third Int. Conf. Intell. Networks Intell. Syst.*, pp. 552–555, Nov. 2010.

[6] Y. Zhu, S. L. Keoh, M. Sloman, E. Lupu, Y. Zhang, N. Dulay, and N. Pryce, "Finger: An efficient policy system for body sensor networks," in *Mobile Ad Hoc and Sensor Systems, 2008. MASS 2008. 5th IEEE International Conference on*, 2008, pp. 428–433.

[7] S. Misra and A. Jain, "Policy controlled self-configuration in unattended wireless sensor networks," *J. Netw. Comput. Appl.*, vol. 34, no. 5, pp. 1530–1544, Jul. 2011.

[8] J. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," *Proc. 2nd Int. Conf. Embed. networked Sens. Syst.*, 2004.

[9] P. Baronti, P. Pillai, V. Chook, S. Chessa, a Gotta, and Y. Hu, "Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards," *Comput. Commun.*, vol. 30, no. 7, pp. 1655–1695, May 2007.

[10] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith, "Ambient Backscatter : Wireless Communication Out of Thin Air," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 39–50, 2013.

[11] S. Movassaghi, S. Member, M. Abolhasan, and S. Member, "Wireless Body Area Networks : A Survey," *Commun. Surv. Tutorials, IEEE*, vol. pp, no. 99, pp. 1–29, 2014.

[12] T. Naumowicz, R. Freeman, H. Kirk, B. Dean, M. Calsyn, A. Liers, A. Braendle, T. Guilford, and J. Schiller, “Wireless Sensor Network for Habitat Monitoring on Skomer Island,” in *Local Computer Networks (LCN), 2010 IEEE 35th Conference*, 2010, pp. 882–889.

[13] S. Kim, S. Pakzadt, D. Cullert, J. Demmel, G. Fenvest, S. Glasert, and M. Turon, “Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks,” in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, 2007, pp. 254–263.

[14] Y. Zhu, S. L. Keoh, M. Sloman, E. Lupu, N. Dulay, and N. Pryce, “An Efficient Policy System for Body Sensor Networks,” *2008 14th IEEE Int. Conf. Parallel Distrib. Syst.*, vol. 1, pp. 383–390, Dec. 2008.

[15] T. Bourdenas, M. Sloman, and E. C. Lupu, “Self-healing for pervasive computing systems,” *Archit. dependable Syst. VII. Springer Berlin Heidelb.*, vol. VII, pp. 1–25, 2010.

[16] “Finger2IPv6 project.” [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/Finger2IPv6. [Accessed: 02-Feb-2014].

[17] “TOSServ project.” [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSServ. [Accessed: 02-Feb-2014].

[18] D. Agrawal, *Policy Technology for self-managing systems*, 1st ed. IBM Press, 2009.

[19] M. S. Siddiqui and S. H. Ahmed, “Policy-based network management in a machine-to-machine (M2M) network,” *2012 15th Int. Multitopic Conf.*, pp. 387–393, Dec. 2012.

[20] OASIS, “OASIS (Organization for the Advancement of Structured Information Standards).” [Online]. Available: http://www.oasis-open.org/. [Accessed: 02-Feb-2014].

[21] R. Kamal, M. S. Siddiqui, H. Rim, and C. S. Hong, “A policy based management framework for machine to machine networks and services,” in *2011 13th Asia-Pacific Network Operations and Management Symposium*, 2011, pp. 1–4.

[22] J. Lobo, R. Bhatia, and S. Naqvi, “A Policy Description Language,” in *AAAI*, 1999, pp. 291–298.

[23] DMTF, *CIM Query Language Specification*, 1.0.0 ed. DMTF, 2007.

[24] C. Han, R. Kumar, and R. Shea, “A dynamic operating system for sensor nodes,” *Proc. 3rd Int. Conf. Mob. Syst.*, pp. 163–176, 2005.

[25] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[26] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A Policy System for Autonomous Pervasive Environments," *2009 Fifth Int. Conf. Auton. Auton. Syst.*, pp. 330–335, 2009.

[27] M. S. Beigi, S. Calo, and D. Verma, "Policy transformation techniques in policy-based systems management," *Proceedings. Fifth IEEE Int. Work. Policies Distrib. Syst. Networks, 2004. POLICY 2004.*, pp. 13–22, 2004.

[28] S. Srinivasan, V. Sapra, M. Studies, and N. Delhi, "Integration of Rule based and Case based Reasoning System to Support Decision Making," pp. 106–108, 2014.

[29] R. Marin, J. Vivero, P. Leitner, M. Zach, and C. Fahy, "A Distributed Policy Based Solution in a Fault Management Scenario," in *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, 2006, pp. 1–5.

[30] a. Galani, N. Koutsouris, K. Tsagkaris, P. Demestichas, B. Fuentes, C. G. Vazquez, and G. Nguengang, "A policy based framework for governing Future networks," *2012 IEEE Globecom Work.*, pp. 802–806, Dec. 2012.

[31] N. VanderHorn, B. Haan, M. Carvalho, and C. Perez, "Distributed policy learning for the Cognitive Network Management System," *2010 - Milcom 2010 Mil. Commun. Conf.*, pp. 435–440, Oct. 2010.

[32] S. H. Lee, L. Choi, Y. Nah, S. Hong, and J.-A. Jun, "Policy-Based Reprogramming for Wireless Sensor Networks," *2010 13th IEEE Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput. Work.*, pp. 194–203, 2010.

[33] A. Jacquot, J.-P. Chanet, K. M. Hou, X. X. Diao, and J.-J. Li, "A New Approach for Wireless Sensor Network Management: LiveNCM," *2008 New Technol. Mobil. Secur.*, no. Table I, pp. 1–6, Nov. 2008.

[34] M. Ayari, F. Kamoun, and G. Pujolle, "Towards Autonomous Mobile Ad Hoc Networks: A Distributed Policy-Based Management Approach," *2008 Fourth Int. Conf. Wirel. Mob. Commun.*, pp. 201–206, 2008.

[35] V. V. Thanh, H. N. Chan, B. P. Viet, and T. N. Huu, "A survey of routing using DHTs over Wireless Sensor Networks," *Technology*, no. Icita, pp. 978–981, 2009.

[36] S. Ratnasamy, S. Shenker, I. Icsi, B. Karp, D. Estrin, and U. C. B. Eecs, "Data-Centric Storage in Sensornets with GHT , A Geographic Hash Table," *Mob. Networks Appl.*, 2003.

[37] M. Ali and Z. a. Uzmi, "CSN: a network protocol for serving dynamic queries in large-scale wireless sensor networks," *Proceedings. Second Annu. Conf. Commun. Networks Serv. Res. 2004.*, pp. 165–174, 2004.

[38] M. Caesar, M. Castro, and E. Nightingale, "Virtual ring routing: network routing inspired by DHTs," *ACM SIGCOMM*, 2006.

[39] O. Landsiedel, K. a. Lehmann, and K. Wehrle, "T-DHT: Topology-based Distributed Hash Tables," *Fifth IEEE Int. Conf. Peer-to-Peer Comput. (P2P 2005)*, no. 2, pp. 143–144, 2005.

[40] F. Araujo, L. Rodrigues, J. Kaiser, and C. Mitidieri, "CHR: A Distributed Hash Table for Wireless Ad Hoc Networks," *25th IEEE Int. Conf. Distrib. Comput. Syst. Work.*, no. June 2005, pp. 407–413.

[41] A. A.-B. Al-Mamou and H. Labiod, "ScatterPastry: An Overlay Routing Using a DHT over Wireless Sensor Networks," *2007 Int. Conf. Intell. Pervasive Comput. (IPC 2007)*, pp. 274–279, Oct. 2007.

[42] G. Al Sukkar, H. Afifi, and S.-M. Senouci, "Lightweight and Distributed Algorithms for Efficient Data-Centric Storage in Sensor Networks," *2007 Fourth Annu. Int. Conf. Mob. Ubiquitous Syst. Netw. Serv.*, pp. 1–3, 2007.

[43] A. Rowstron and P. Druschel, "Pastry : Scalable , decentralized object location and routing for large-scale peer-to-peer systems," *Design*, no. November 2001, 1892.

[44] G. Gutiérrez, B. Mejías, P. Van Roy, D. Velasco, and J. Torres, "WSN and P2P: A Self-Managing Marriage," *2008 Second IEEE Int. Conf. Self-Adaptive Self-Organizing Syst. Work.*, pp. 198–201, Oct. 2008.

[45] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[46] B. F. Content-delivery-as-a-service, Y. Jin, Y. Wen, W. Zhang, and S. Member, "Content Routing and Lookup Schemes using Global," vol. 8, no. 1, pp. 268–278, 2014.

[47] C. Jardak and J. Riihij, "Analyzing the Optimal Use of Bloom Filters in Wireless Sensor Networks Storing Replicas," pp. 1–6, 2009.

[48] A. Kirsch and M. Mitzenmacher, "Less Hashing , Same Performance : Building a Better Bloom Filter," pp. 187–218, 2008.

[49] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of Bloom filters," *Inf. Process. Lett.*, vol. 108, no. 4, pp. 210–213, Oct. 2008.

[50] M. V. Ramakrishna and J. Zobel, “Performance in practice of string hashing functions,” in *Proc. of the Fifth International Conference on Database Systems for Advanced Applications*, 1997, pp. 215–223.

[51] B. Jenkins, “Dr. Dobb’s Journal.” [Online]. Available: http://www.drdobbs.com/database/algorithm-alley/184410284?pgno=5. [Accessed: 02-Feb-2014].

[52] B. Jenkins, “Bob Jenkins.” [Online]. Available: http://burtleburtle.net/bob/hash/doobs.html. [Accessed: 02-Feb-2014].

[53] P. Levis and D. Gay, *TinyOS Programming*. Cambridge: Cambridge University Press, 2009.

[54] B. L. Titzer, D. K. Lee, J. Palsberg, and A. Background, “Avrora : Scalable Sensor Network Simulation with Precise Timing,” pp. 477–482, 2005.

[55] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A Survey and Comparison of Peer-to-Peer Overlay Network Schemes,” no. March, 2004.

[56] E. Egea-López, J. Vales-Alonso, A. S. Martínez-Sala, P. Pavón-Mariño, and J. García-Haro, “Simulation tools for wireless sensor networks,” in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS05).*, 2005, p. 24.

[57] S. Kim, J. Chung, and S. Member, “Message Complexity Analysis of Mobile Ad Hoc Network Address Autoconfiguration Protocols,” vol. 7, no. 3, pp. 358–371, 2008.

[58] P. Levis, N. Lee, M. Welsh, and D. Culler, “TOSSIM : Accurate and Scalable Simulation of Entire TinyOS Applications.”

[59] D. Moss, N. Wilmot, and P. Levis, “BoX-MACs : Exploiting Physical and Link Layer Boundaries in Low-Power Networking,” *Comput. Syst. Lab. Stanford Univ.*, 2008.

[60] B. A. I. Housani, B. Mutrib, and H. Jaradi, “The Linux Review - Ubuntu Desktop Edition - version 8 . 10,” in *Current Trends in Information Technology (CTIT), 2009 International Conference on the*, 2009, pp. 1–6.

[61] I. Trends, “Java IDE,” *Computer (Long. Beach. Calif).*, vol. 38, no. 7, pp. 16–18, 2005.

[62] G. Lindstrom, “Programming with Python,” *IT Prof.*, vol. 7, no. 5, pp. 10–16, Sep. 2005.

[63] S. Nellen, “Eclipse Plugin for TinyOS Debugging,” Distributed Computing Group-Computer Engineering and Networks Laboratory (TIK), 2009.

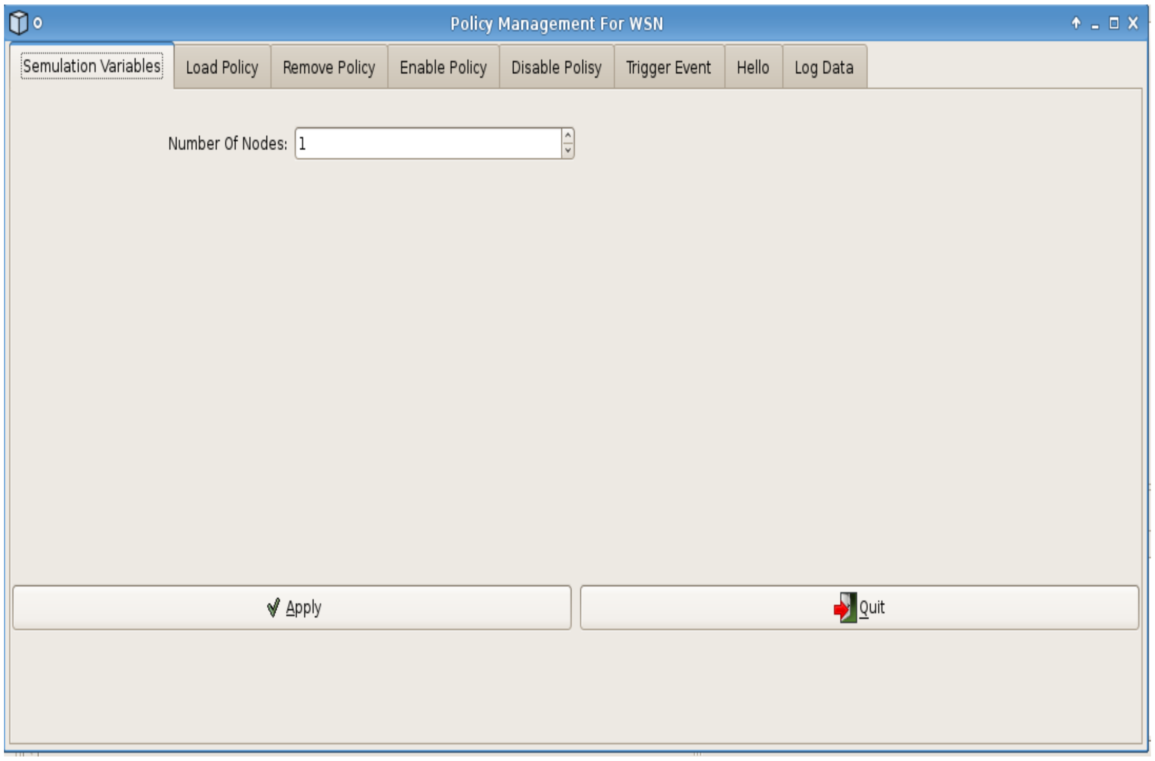
[64] M. Ilavsk and R. Jakˇ, “Interactive Evolution of Graphical User Interface with Gtk Toolkit,” in *Cognitive Infocommunications (CogInfoCom), 2011 2nd International Conference on*, 2011, pp. 1–6.

[65] N. Qwasmi, D. Smullen, and R. Liscano, “Integrated Development Environment for Debugging Policy-based Applications in Wireless Sensor Networks,” *Procedia Comput. Sci.*, vol. 21, no. 2013, pp. 225–233, 2013.

Appendixes

Appendix A Policy management tool (Policy IDE) interface

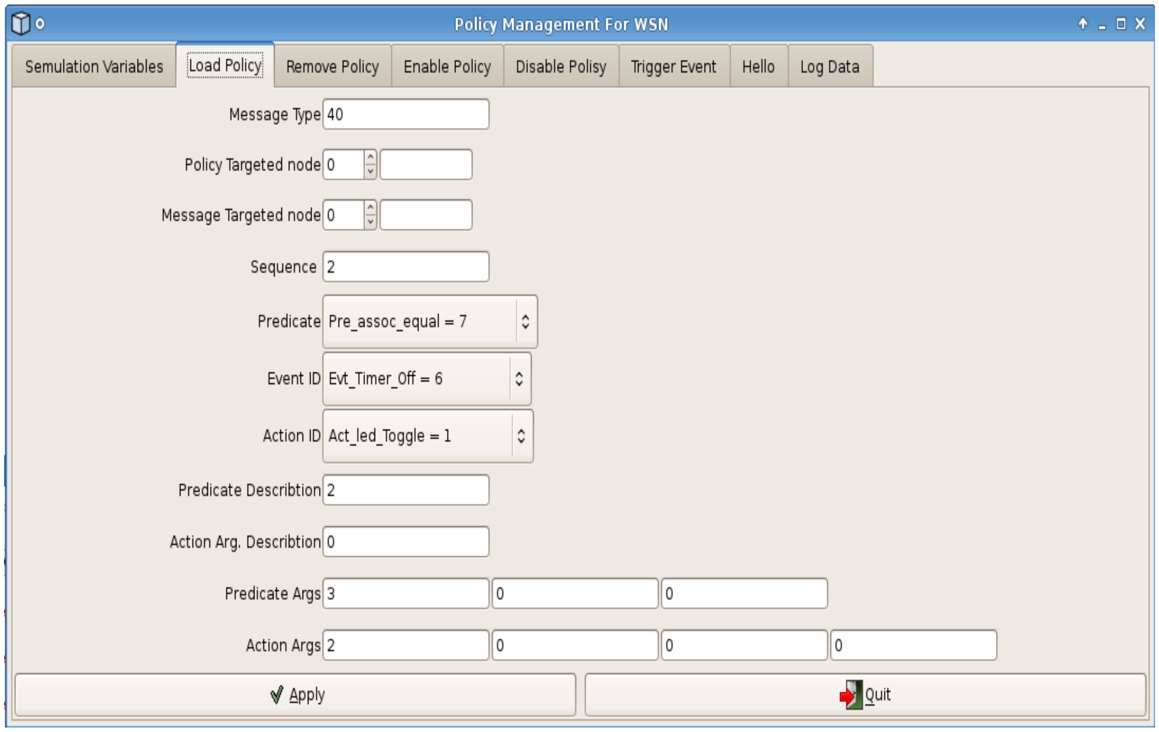
The policy management tool's GUI consists of eight tabs as shown in Figure A.1. Each tab is designed to perform a specific task. The first tab is the Simulation Variables tab. In this tab, the user enters the simulation variables, which currently control only the number of nodes required for the simulation. At present, this screen only controls the number of nodes, but it is possible to include more simulation parameters, such as noise and links, which control the network topology. The simulation starts after entering the number of nodes and pressing the apply button. From then on, the number of nodes cannot be modified, and so the apply button disappears from the screen. If the number of nodes needs to be changed, the simulation must be restarted.



Number Of Nodes: 1

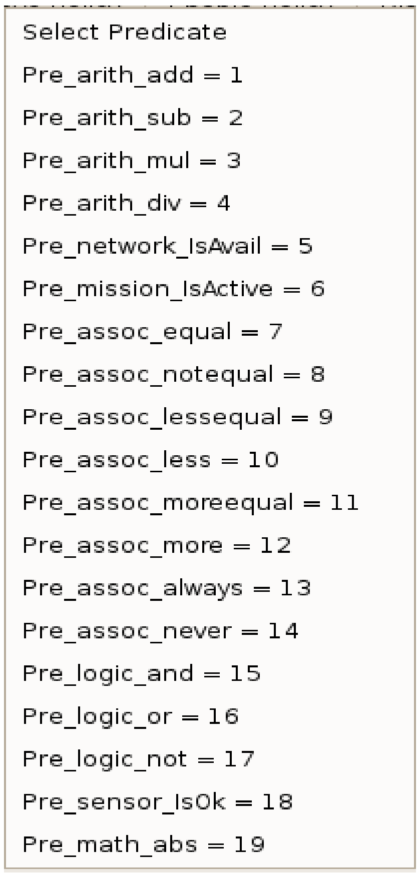
Figure A.1 PMT-Simulation Variables

The second tab is the Load Policy tab. Here, the user can create a policy and load it into a specific mote based on the PolicyP2P algorithm. In this tab, the user provides all required parameters to create a policy. The message type is a predefined number for the network packet type designated to perform specific tasks. The Load Policy task uses message type 40. The Policy Targeted node and Message Targeted node fields are for providing the mote network address. As shown in Figure A.2, there are two fields for both the Policy Targeted node and the Message Targeted node. The reason is that each mote has two different addresses: The first is the network physical address and the second is the overlay network address (AM address). The intention here is to give users more flexibility by using either of the two addresses. The Sequence field is a numeric field representing the value of the policy sequence field in the policy key as shown in Figure 19. The Predicate field is a selection menu for predefined predicates: equals, less than, greater than, etc. The Predicate is used to validate the condition in the policy by comparing the parameter in the policy with the value provided by the triggered event. The output of the menu selection is a numerical representation of the selected operation. A list of available predicates is shown in Figure A.3. The Event ID field is a selection menu for a preset list of available Event IDs. The output of the menu selection is a numerical representation of the selected event. A list of available events is shown in Figure A.4. The Action ID field is a selection menu for a preset list of available action IDs. The output of the menu selection is a numerical representation of the selected action. A list of available actions is shown in Figure A.5. The Predicate Description field is a numeric field representing the order of parameters in the condition statement. The Action Arg. Description field is a numeric field to control the parameter for the required action. Predicate Args and Action Args provide arguments for the predicate and action inputs respectively.



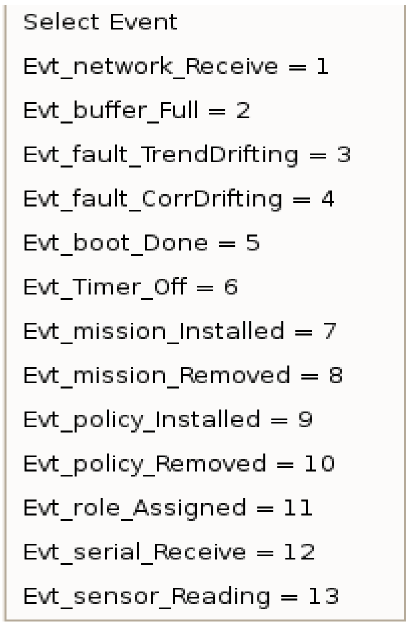
Message Type	40			
Policy Targeted node	0			
Message Targeted node	0			
Sequence	2			
Predicate	Pre_assoc_equal = 7			
Event ID	Evt_Timer_Off = 6			
Action ID	Act_led_Toggle = 1			
Predicate Describtion	2			
Action Arg. Describtion	0			
Predicate Args	3	0	0	
Action Args	2	0	0	0

Figure A.2 PMT-Load Policy



Select Predicate
Pre_arith_add = 1
Pre_arith_sub = 2
Pre_arith_mul = 3
Pre_arith_div = 4
Pre_network_IsAvail = 5
Pre_mission_IsActive = 6
Pre_assoc_equal = 7
Pre_assoc_notequal = 8
Pre_assoc_lessequal = 9
Pre_assoc_less = 10
Pre_assoc_moreequal = 11
Pre_assoc_more = 12
Pre_assoc_always = 13
Pre_assoc_never = 14
Pre_logic_and = 15
Pre_logic_or = 16
Pre_logic_not = 17
Pre_sensor_IsOk = 18
Pre_math_abs = 19

Figure A.3 Predicate list



Select Event
Evt_network_Receive = 1
Evt_buffer_Full = 2
Evt_fault_TrendDrifting = 3
Evt_fault_CorrDrifting = 4
Evt_boot_Done = 5
Evt_Timer_Off = 6
Evt_mission_Installed = 7
Evt_mission_Removed = 8
Evt_policy_Installed = 9
Evt_policy_Removed = 10
Evt_role_Assigned = 11
Evt_serial_Receive = 12
Evt_sensor_Reading = 13

Figure A.4 Event list

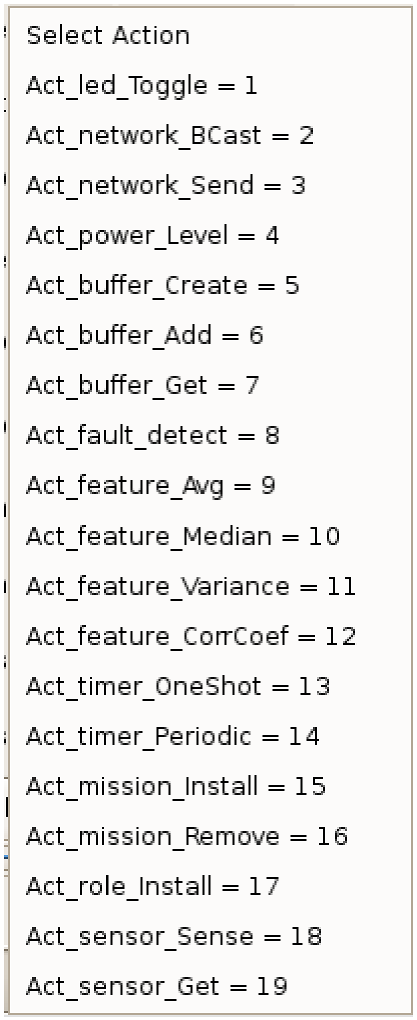
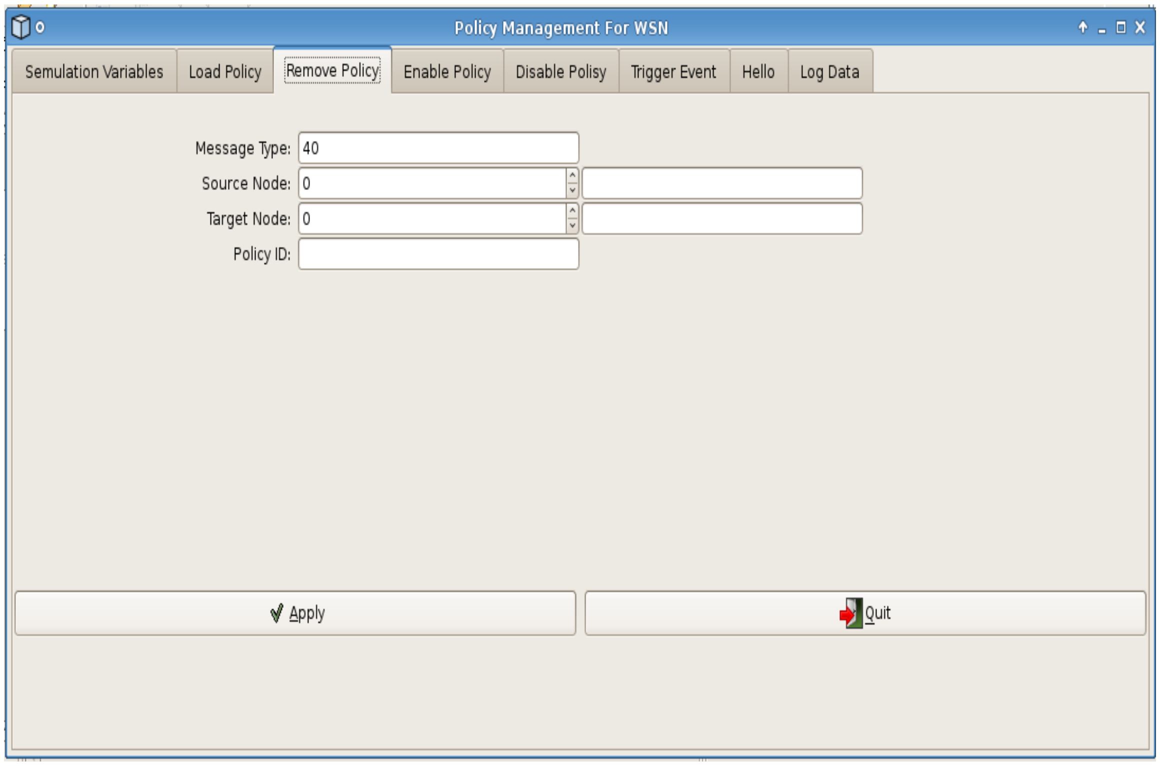


Figure A.5 Action list

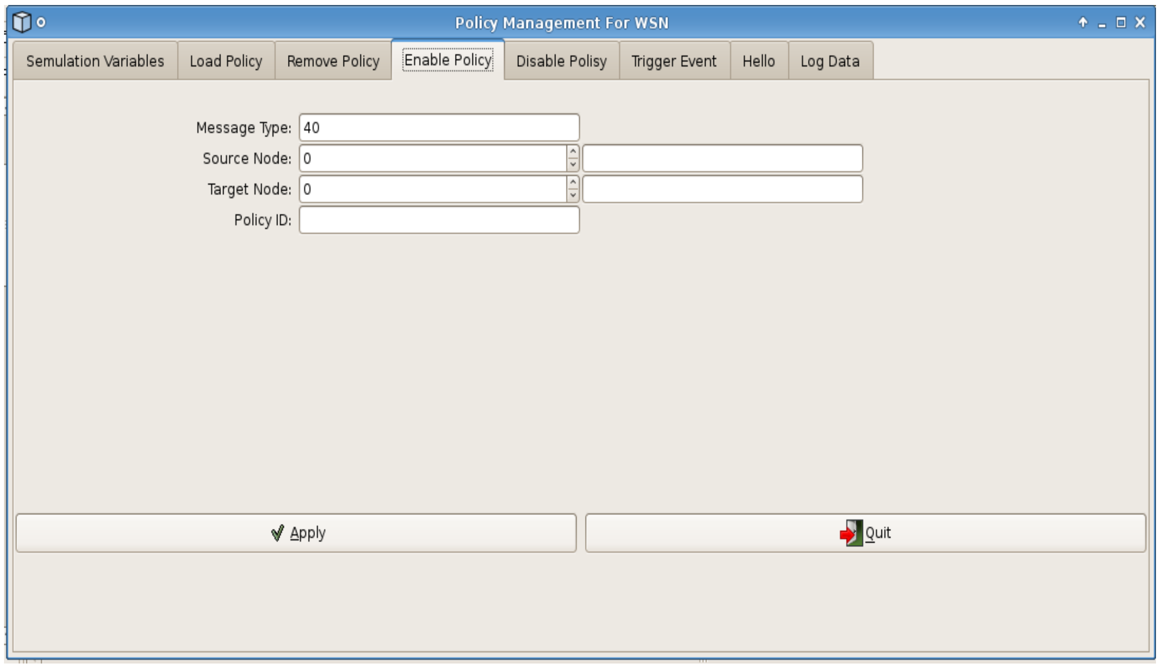
The third tab is the Remove Policy tab, as shown in Figure A.6. Here, the user can remove (delete) a policy from any targeted mote. In this tab, the user provides all required parameters to remove a policy. The Message Type is a preset value for the network packet type designated to perform specific tasks. The Remove Policy task uses network message type 40. The Source Node and Target Node fields are for providing a mote network address. As shown in Figure A.6, there are two fields for both the source node and target node. The reason is that each mote has two different addresses: The first is the network physical address and the second is the overlay network address (AM address). The intention here is to give users more flexibility by using either of the two addresses. The Policy ID field is a numeric field representing the hashed value of a policy key as shown in Figure 19.



Message Type: 40
Source Node: 0
Target Node: 0
Policy ID:

Figure A.6 PMT-Remove Policy

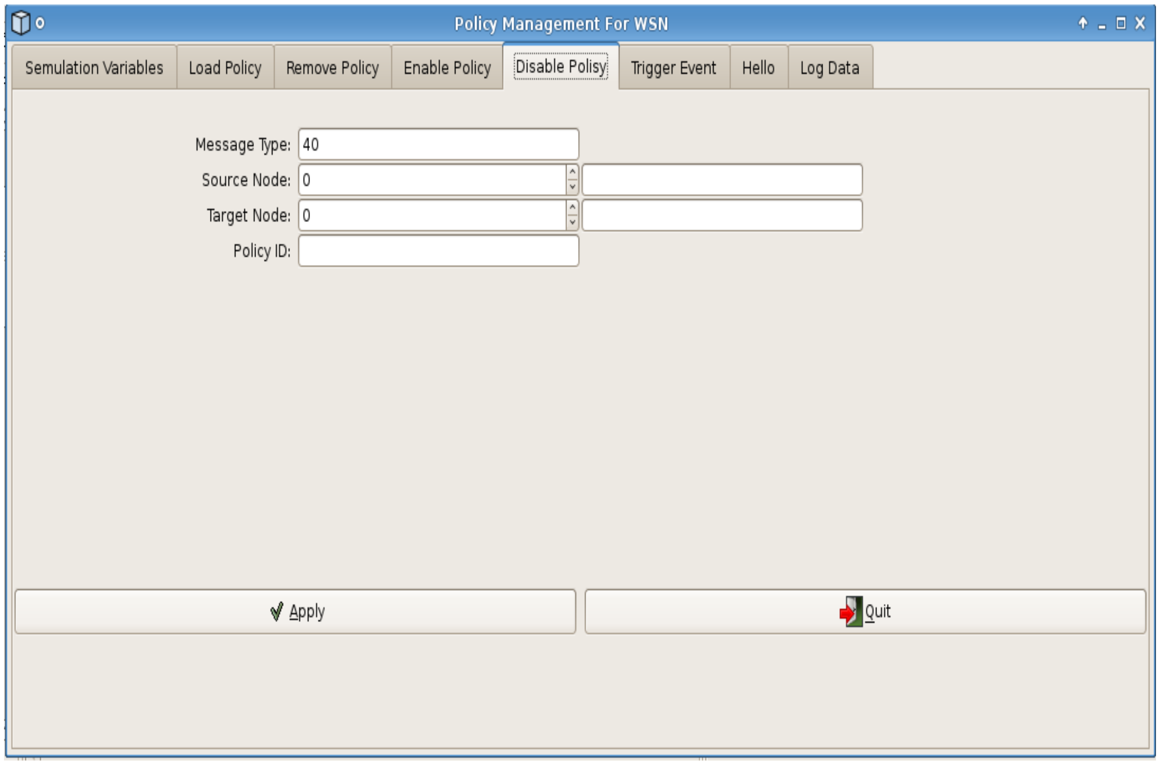
The fourth tab is the Enable Policy tab. Here, the user can enable a policy in any targeted mote. In this tab, the user provides all required parameters to enable a policy. The enable policy task uses network message type 40. The Source and Target fields are for providing a mote network address. As shown in Figure A.7, the Source Node and Target Node each have two fields, so the user can provide either the network physical address or the overlay network address (AM address). The Policy ID field is a numeric field representing the hashed value of a policy key as shown in Figure 19.



Message Type: 40
Source Node: 0
Target Node: 0
Policy ID:

Figure A.7 PMT-Enable Policy

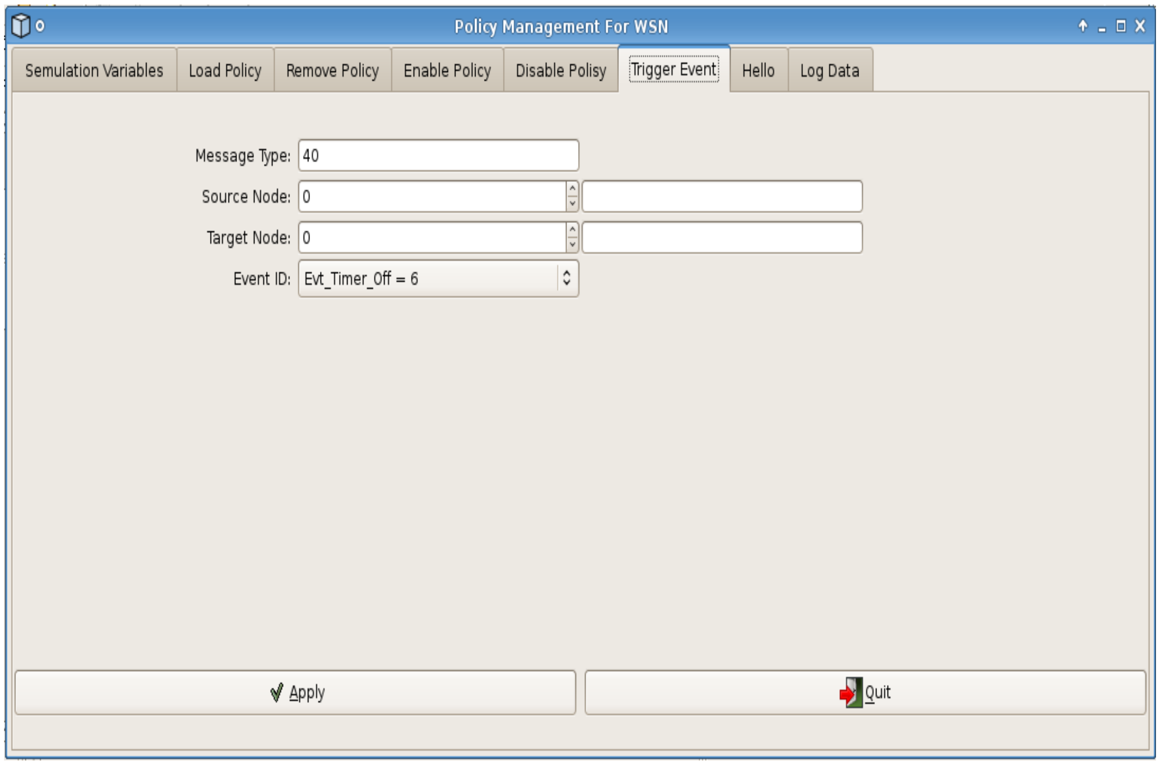
The fifth tab is the Disable Policy tab. Here, the user can disable a policy in any targeted mote. In this tab, the user provides all required parameters to disable a policy. The Disable Policy task uses network message type 40. The Source and Target fields are for providing a mote network address. As shown in Figure A.8, the Source Node and Target Node each have two fields, so the user can provide either the network physical address or the overlay network address (AM address). The Policy ID field is a numeric field representing the hashed value of a policy key as shown in Figure 19.



Message Type: 40
Source Node: 0
Target Node: 0
Policy ID:

Figure A.8 PMT-Disable Policy

The sixth tab is the Trigger Event policy tab. Here, the user can trigger an event in any targeted mote. In this tab, the user provides all required parameters to trigger an event. The Trigger Event task uses network message type 40. The source and target fields are for providing a mote network address. As shown in Figure A.9, the Source Node and Target Node each have two fields, so the user can provide either the network physical address or the overlay network address (AM address). The Event ID field is a selection menu for a preset list of available Event IDs. The output of the menu selection is a numerical representation of the selected event. A list of available events is shown in Figure A.4.



Message Type: 40

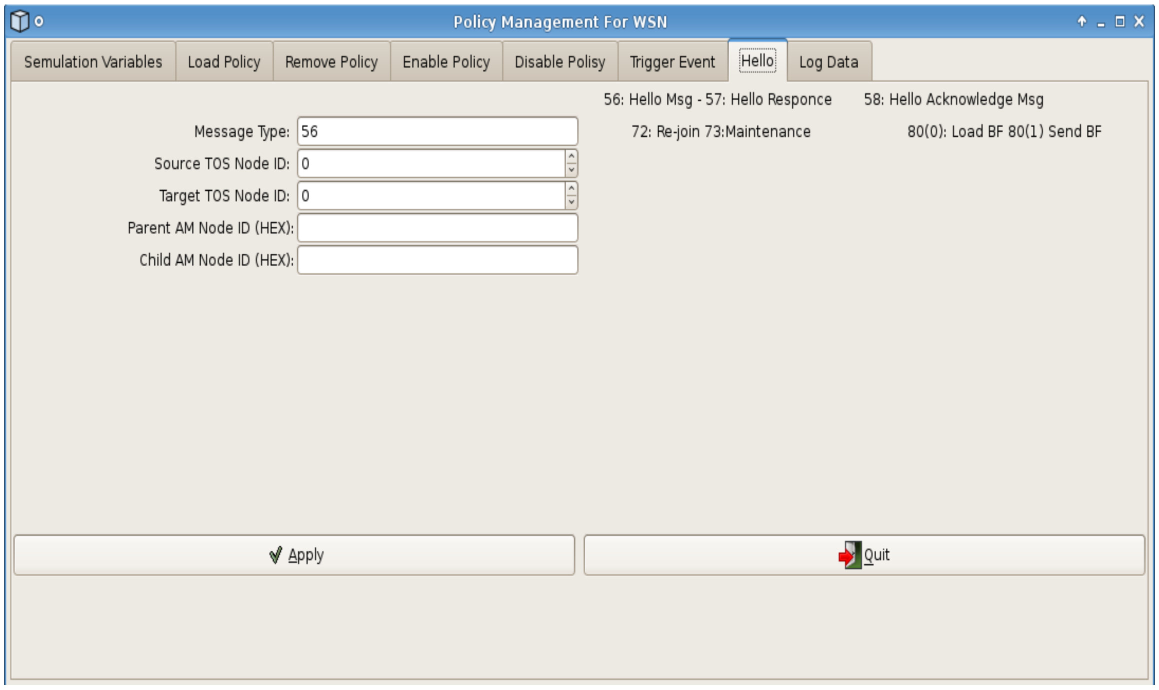
Source Node: 0

Target Node: 0

Event ID: Evt_Timer_Off = 6

Figure A.9 PMT-Trigger Event

The seventh tab is the Overlay Network Messages (Hello) tab. In this tab, the user can inject various overlay network messages, such as hello, re-join, BLOOM_FILTER, and maintenance messages into any targeted mote. As shown in Figure A.10, the user provides all required parameters to inject overlay network messages. Overlay network messages use various predefined message types, such as 56 for hello messages, 57 for Hello-Response messages, 58 for Hello-Acknowledgment messages, 72 for Re-Join messages, 73 for maintenance messages, and 80 for BLOOM_FILTER messages. The Source TOS Node ID and Target TOS Node ID fields are for providing a mote network physical address. The Parent AM Node ID and Child AM Node ID fields are numeric fields to provide the overlay network addresses for the source and destination motes.

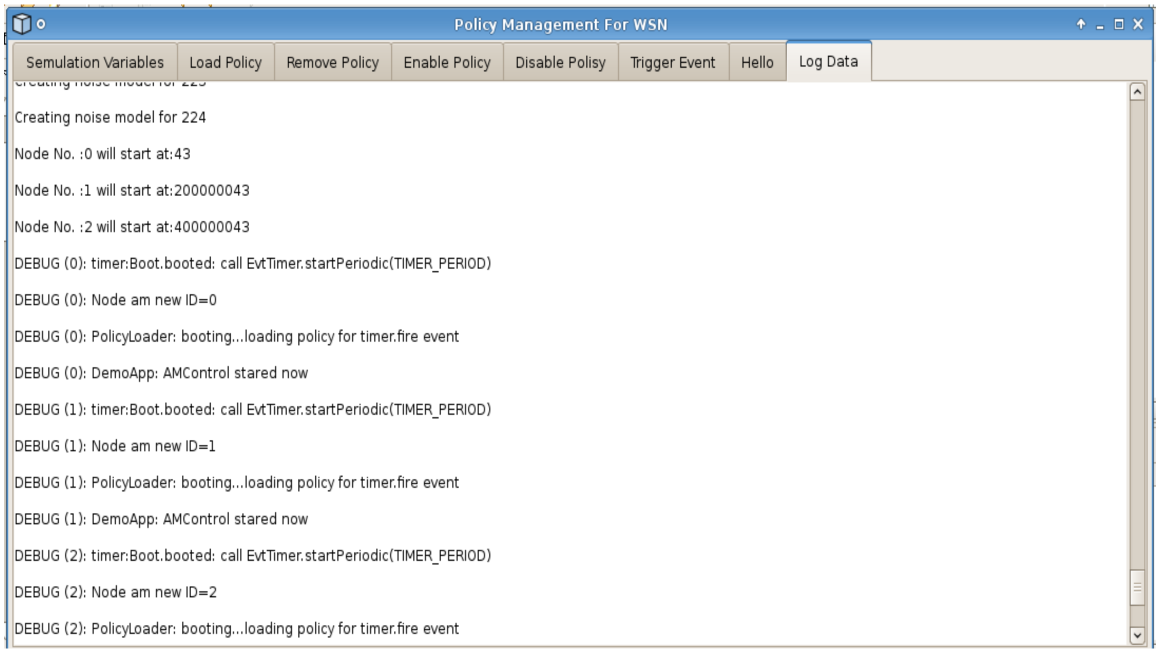


56: Hello Msg - 57: Hello Responce 58: Hello Acknowledge Msg
72: Re-join 73:Maintenance 80(0): Load BF 80(1) Send BF

Message Type: 56
Source TOS Node ID: 0
Target TOS Node ID: 0
Parent AM Node ID (HEX):
Child AM Node ID (HEX):

Figure A.10 PMT-Overlay Network Messages

The last tab is for log data. In this tab, the user can display all testing and debugging data provided by the mote through the TOSSIM environment. This text widget is linked to a text file at compilation time. The text file is updated by the TOSSIM software through a dedicated communication link with the motes.



```
Creating noise model for 224
Node No. :0 will start at:43
Node No. :1 will start at:200000043
Node No. :2 will start at:400000043
DEBUG (0): timer:Boot.booted: call EvtTimer.startPeriodic(TIMER_PERIOD)
DEBUG (0): Node am new ID=0
DEBUG (0): PolicyLoader: booting...loading policy for timer.fire event
DEBUG (0): DemoApp: AMControl stared now
DEBUG (1): timer:Boot.booted: call EvtTimer.startPeriodic(TIMER_PERIOD)
DEBUG (1): Node am new ID=1
DEBUG (1): PolicyLoader: booting...loading policy for timer.fire event
DEBUG (1): DemoApp: AMControl stared now
DEBUG (2): timer:Boot.booted: call EvtTimer.startPeriodic(TIMER_PERIOD)
DEBUG (2): Node am new ID=2
DEBUG (2): PolicyLoader: booting...loading policy for timer.fire event
```

Figure A.11 PMT-Log Data

Appendix B Mathematical Model Data

Network formation performance

The Network formation data table shows various network sizes ranging from 2 nodes to 200 nodes. Each node has a leaf table with a capacity of 16 entries, which means that each parent node can have a maximum of 16 children. Data from our analysis shows the number of messages, the number of bytes, and the time (in seconds) of all required messages for network formation; these are Hello, Response, Acknowledgment, and Re-join.

Network formation data

		Hello Message			Response Message			Acknowledgment Message		
Number of Node	Number of Leaf Node	Number of Messages	Number of Bytes	Time (s)	Number of Messages	Number of Bytes	Time (s)	Number of Messages	Number of Bytes	Time (s)
2	16	1	2	0.000064	1	6	0.000192	1	4	0.000128
5	16	4	8	0.000256	10	60	0.00192	4	16	0.000512
10	16	9	18	0.000576	45	270	0.00864	9	36	0.001152
20	16	19	38	0.001216	190	1140	0.03648	19	76	0.002432
30	16	29	58	0.001856	435	2610	0.08352	29	116	0.003712
40	16	39	78	0.002496	780	4680	0.14976	39	156	0.004992
50	16	49	98	0.003136	1225	7350	0.2352	49	196	0.006272
60	16	59	118	0.003776	1770	10620	0.33984	59	236	0.007552

70	16	69	138	0.004416	2415	14490	0.46368	69	276	0.008832
80	16	79	158	0.005056	3160	18960	0.60672	79	316	0.010112
90	16	89	178	0.005696	4005	24030	0.76896	89	356	0.011392
100	16	99	198	0.006336	4950	29700	0.9504	99	396	0.012672
110	16	109	218	0.006976	5995	35970	1.15104	109	436	0.013952
120	16	119	238	0.007616	7140	42840	1.37088	119	476	0.015232
130	16	129	258	0.008256	8385	50310	1.60992	129	516	0.016512
140	16	139	278	0.008896	9730	58380	1.86816	139	556	0.017792
150	16	149	298	0.009536	11175	67050	2.1456	149	596	0.019072
160	16	159	318	0.010176	12720	76320	2.44224	159	636	0.020352
170	16	169	338	0.010816	14365	86190	2.75808	169	676	0.021632
180	16	179	358	0.011456	16110	96660	3.09312	179	716	0.022912
190	16	189	378	0.012096	17955	107730	3.44736	189	756	0.024192
200	16	199	398	0.012736	19900	119400	3.8208	199	796	0.025472

Number of Node	Number of Leaf Node	Re-join Message			Total network formation messages		
		Number of Messages	Number of Bytes	Time (s)	Number of Messages	Number of Bytes	Time (s)
2	16	2	4	0.000128	5	16	0.000512
5	16	5	10	0.00032	23	94	0.003008
10	16	10	20	0.00064	73	344	0.011008
20	16	20	40	0.00128	248	1294	0.041408
30	16	30	60	0.00192	523	2844	0.091008
40	16	40	80	0.00256	898	4994	0.159808
50	16	50	100	0.0032	1373	7744	0.247808
60	16	60	120	0.00384	1948	11094	0.355008
70	16	70	140	0.00448	2623	15044	0.481408

80	16	80	160	0.00512	3398	19594	0.627008
90	16	90	180	0.00576	4273	24744	0.791808
100	16	100	200	0.0064	5248	30494	0.975808
110	16	110	220	0.00704	6323	36844	1.179008
120	16	120	240	0.00768	7498	43794	1.401408
130	16	130	260	0.00832	8773	51344	1.643008
140	16	140	280	0.00896	10148	59494	1.903808
150	16	150	300	0.0096	11623	68244	2.183808
160	16	160	320	0.01024	13198	77594	2.483008
170	16	170	340	0.01088	14873	87544	2.801408
180	16	180	360	0.01152	16648	98094	3.139008
190	16	190	380	0.01216	18523	109244	3.495808
200	16	200	400	0.0128	20498	120994	3.871808

Policy loading performance

The table shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node has a local policy repository with a capacity of 20 entries, which means that each node can have a maximum of 20 policies in its memory. Analysis data shows the number of messages, the number of bytes, and the time (in seconds) of all required messages (Get and Response) to load policies into the network for P2P algorithm usage and into the local node for local node usage. The table shows the minimum, maximum, and average performance of each category.

Policy loading performance data

Nodes	Policy repository size	Network total policies	Number of messages required to load policies into network			Number of bytes required to load policies into network			Time required to load policies into network (s)		
			Min	Max	Average	Min	Max	Average	Min	Max	Average
2	20	40	40	120	80	1160	3480	2320	0.03712	0.11136	0.07424
5	20	100	100	300	200	2900	8700	5800	0.0928	0.2784	0.1856
10	20	200	200	600	400	5800	17400	11600	0.1856	0.5568	0.3712
20	20	400	400	1200	800	11600	34800	23200	0.3712	1.1136	0.7424
30	20	600	600	1800	1200	17400	52200	34800	0.5568	1.6704	1.1136
40	20	800	800	2400	1600	23200	69600	46400	0.7424	2.2272	1.4848
50	20	1000	1000	3000	2000	29000	87000	58000	0.928	2.784	1.856
60	20	1200	1200	3600	2400	34800	104400	69600	1.1136	3.3408	2.2272
70	20	1400	1400	4200	2800	40600	121800	81200	1.2992	3.8976	2.5984
80	20	1600	1600	4800	3200	46400	139200	92800	1.4848	4.4544	2.9696
90	20	1800	1800	5400	3600	52200	156600	104400	1.6704	5.0112	3.3408
100	20	2000	2000	6000	4000	58000	174000	116000	1.856	5.568	3.712
110	20	2200	2200	6600	4400	63800	191400	127600	2.0416	6.1248	4.0832
120	20	2400	2400	7200	4800	69600	208800	139200	2.2272	6.6816	4.4544
130	20	2600	2600	7800	5200	75400	226200	150800	2.4128	7.2384	4.8256
140	20	2800	2800	8400	5600	81200	243600	162400	2.5984	7.7952	5.1968
150	20	3000	3000	9000	6000	87000	261000	174000	2.784	8.352	5.568
160	20	3200	3200	9600	6400	92800	278400	185600	2.9696	8.9088	5.9392
170	20	3400	3400	10200	6800	98600	295800	197200	3.1552	9.4656	6.3104
180	20	3600	3600	10800	7200	104400	313200	208800	3.3408	10.0224	6.6816
190	20	3800	3800	11400	7600	110200	330600	220400	3.5264	10.5792	7.0528
200	20	4000	4000	12000	8000	116000	348000	232000	3.712	11.136	7.424

Nodes	Policy repository size	Network total policies	Number of messages required to load policies into local node (Get and Response)			Number of bytes required to load policies into local node			Time required to load policies into local node (s)		
			Min	Max	Average	Min	Max	Average	Min	Max	Average
2	20	40	80	160	120	2320	4640	3480	0.07424	0.14848	0.11136
5	20	100	200	400	300	5800	11600	8700	0.1856	0.3712	0.2784
10	20	200	400	800	600	11600	23200	17400	0.3712	0.7424	0.5568
20	20	400	800	1600	1200	23200	46400	34800	0.7424	1.4848	1.1136
30	20	600	1200	2400	1800	34800	69600	52200	1.1136	2.2272	1.6704
40	20	800	1600	3200	2400	46400	92800	69600	1.4848	2.9696	2.2272
50	20	1000	2000	4000	3000	58000	116000	87000	1.856	3.712	2.784
60	20	1200	2400	4800	3600	69600	139200	104400	2.2272	4.4544	3.3408
70	20	1400	2800	5600	4200	81200	162400	121800	2.5984	5.1968	3.8976
80	20	1600	3200	6400	4800	92800	185600	139200	2.9696	5.9392	4.4544
90	20	1800	3600	7200	5400	104400	208800	156600	3.3408	6.6816	5.0112
100	20	2000	4000	8000	6000	116000	232000	174000	3.712	7.424	5.568
110	20	2200	4400	8800	6600	127600	255200	191400	4.0832	8.1664	6.1248
120	20	2400	4800	9600	7200	139200	278400	208800	4.4544	8.9088	6.6816
130	20	2600	5200	10400	7800	150800	301600	226200	4.8256	9.6512	7.2384
140	20	2800	5600	11200	8400	162400	324800	243600	5.1968	10.3936	7.7952
150	20	3000	6000	12000	9000	174000	348000	261000	5.568	11.136	8.352
160	20	3200	6400	12800	9600	185600	371200	278400	5.9392	11.8784	8.9088
170	20	3400	6800	13600	10200	197200	394400	295800	6.3104	12.6208	9.4656
180	20	3600	7200	14400	10800	208800	417600	313200	6.6816	13.3632	10.0224
190	20	3800	7600	15200	11400	220400	440800	330600	7.0528	14.1056	10.5792
200	20	4000	8000	16000	12000	232000	464000	348000	7.424	14.848	11.136

Nodes	Policy repository size	Network total policies	Total number of messages required to load policies (net and local)			Total number of bytes required to load policies (net and local)			Total time required to load policies (net and local) (s)		
			Min	Max	Average	Min	Max	Average	Min	Max	Average
2	20	40	120	280	200	3480	8120	5800	0.11136	0.25984	0.1856
5	20	100	300	700	500	8700	20300	14500	0.2784	0.6496	0.464
10	20	200	600	1400	1000	17400	40600	29000	0.5568	1.2992	0.928
20	20	400	1200	2800	2000	34800	81200	58000	1.1136	2.5984	1.856
30	20	600	1800	4200	3000	52200	121800	87000	1.6704	3.8976	2.784
40	20	800	2400	5600	4000	69600	162400	116000	2.2272	5.1968	3.712
50	20	1000	3000	7000	5000	87000	203000	145000	2.784	6.496	4.64
60	20	1200	3600	8400	6000	104400	243600	174000	3.3408	7.7952	5.568
70	20	1400	4200	9800	7000	121800	284200	203000	3.8976	9.0944	6.496
80	20	1600	4800	11200	8000	139200	324800	232000	4.4544	10.3936	7.424
90	20	1800	5400	12600	9000	156600	365400	261000	5.0112	11.6928	8.352
100	20	2000	6000	14000	10000	174000	406000	290000	5.568	12.992	9.28
110	20	2200	6600	15400	11000	191400	446600	319000	6.1248	14.2912	10.208
120	20	2400	7200	16800	12000	208800	487200	348000	6.6816	15.5904	11.136
130	20	2600	7800	18200	13000	226200	527800	377000	7.2384	16.8896	12.064
140	20	2800	8400	19600	14000	243600	568400	406000	7.7952	18.1888	12.992
150	20	3000	9000	21000	15000	261000	609000	435000	8.352	19.488	13.92
160	20	3200	9600	22400	16000	278400	649600	464000	8.9088	20.7872	14.848
170	20	3400	10200	23800	17000	295800	690200	493000	9.4656	22.0864	15.776
180	20	3600	10800	25200	18000	313200	730800	522000	10.0224	23.3856	16.704
190	20	3800	11400	26600	19000	330600	771400	551000	10.5792	24.6848	17.632
200	20	4000	12000	28000	20000	348000	812000	580000	11.136	25.984	18.56

Bloom filter performance

The table shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node has a local policy repository with a capacity of 20 entries, which means that each node can have a maximum of 20 policies in its memory. Analysis data shows network total number of policies, number of messages, number of bytes, and time (in seconds) required to look up policies. The table shows the minimum, maximum, and average performance of each category. Finally, the table shows the amount of time saved by using Bloom filter, assuming that the rate of missing policies is 30%.

Bloom filter data

Nodes	Policy repository size	Network total policies	Number of messages required to look up a policy			Number of bytes required to look up a policy			Time required to look up policies (s)			Time saved by using Bloom filter with policy missing rate of 30% (s)		
			Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
2	20	40	40	120	80	1160	3480	2320	0.04	0.11	0.07	0.01	0.03	0.02
5	20	100	100	300	200	2900	8700	5800	0.09	0.28	0.19	0.03	0.08	0.06
10	20	200	200	600	400	5800	17400	11600	0.19	0.56	0.37	0.06	0.17	0.11
20	20	400	400	1200	800	11600	34800	23200	0.37	1.11	0.74	0.11	0.33	0.22
30	20	600	600	1800	1200	17400	52200	34800	0.56	1.67	1.11	0.17	0.50	0.33
40	20	800	800	2400	1600	23200	69600	46400	0.74	2.23	1.48	0.22	0.67	0.45
50	20	1000	1000	3000	2000	29000	87000	58000	0.93	2.78	1.86	0.28	0.84	0.56
60	20	1200	1200	3600	2400	34800	104400	69600	1.11	3.34	2.23	0.33	1.00	0.67
70	20	1400	1400	4200	2800	40600	121800	81200	1.30	3.90	2.60	0.39	1.17	0.78
80	20	1600	1600	4800	3200	46400	139200	92800	1.48	4.45	2.97	0.45	1.34	0.89
90	20	1800	1800	5400	3600	52200	156600	104400	1.67	5.01	3.34	0.50	1.50	1.00

100	20	2000	2000	6000	4000	58000	174000	116000	1.86	5.57	3.71	0.56	1.67	1.11
110	20	2200	2200	6600	4400	63800	191400	127600	2.04	6.12	4.08	0.61	1.84	1.22
120	20	2400	2400	7200	4800	69600	208800	139200	2.23	6.68	4.45	0.67	2.00	1.34
130	20	2600	2600	7800	5200	75400	226200	150800	2.41	7.24	4.83	0.72	2.17	1.45
140	20	2800	2800	8400	5600	81200	243600	162400	2.60	7.80	5.20	0.78	2.34	1.56
150	20	3000	3000	9000	6000	87000	261000	174000	2.78	8.35	5.57	0.84	2.51	1.67
160	20	3200	3200	9600	6400	92800	278400	185600	2.97	8.91	5.94	0.89	2.67	1.78
170	20	3400	3400	10200	6800	98600	295800	197200	3.16	9.47	6.31	0.95	2.84	1.89
180	20	3600	3600	10800	7200	104400	313200	208800	3.34	10.02	6.68	1.00	3.01	2.00
190	20	3800	3800	11400	7600	110200	330600	220400	3.53	10.58	7.05	1.06	3.17	2.12
200	20	4000	4000	12000	8000	116000	348000	232000	3.71	11.14	7.42	1.11	3.34	2.23

Central policy repository performance

In a system with a central policy repository, the Root node functions as the only policy repository in the network. Since there is no central policy repository system to evaluate, the TinyPolicy system was modified to resemble central repository system operation. The analysis data table shows various network sizes ranging from 2 nodes to 200 nodes with a 3-level overlay tree structure. Each node has a local policy repository with a capacity of 20 entries, which means that each node can have a maximum of 20 policies in its memory. Analysis data shows central policy repository size (Root), network total number of policies, number of messages, number of bytes, and time (in seconds) required to load policies into the local node repository using the central repository approach. Finally, the table shows the amount of time (in seconds) required to load the same number of policies using the distributed policy repository approach.

Performance data: Central vs. distributed repository approach

Nodes	Policy repository size	Network total policies	Number of messages required to load policies into local node (Get and Response)	Number of bytes required to load policies into local node	Time required to load policies into local node (s)	Average Time required to load policies into node's local repository using distributed policy repository (s)
2	40	40	80	2320	0.07424	0.1856
5	100	100	200	5800	0.1856	0.464
10	200	200	400	11600	0.3712	0.928
20	400	400	800	23200	0.7424	1.856
30	600	600	1200	34800	1.1136	2.784
40	800	800	1600	46400	1.4848	3.712
50	1000	1000	2000	58000	1.856	4.64
60	1200	1200	2400	69600	2.2272	5.568

70	1400	1400	2800	81200	2.5984	6.496
80	1600	1600	3200	92800	2.9696	7.424
90	1800	1800	3600	104400	3.3408	8.352
100	2000	2000	4000	116000	3.712	9.28
110	2200	2200	4400	127600	4.0832	10.208
120	2400	2400	4800	139200	4.4544	11.136
130	2600	2600	5200	150800	4.8256	12.064
140	2800	2800	5600	162400	5.1968	12.992
150	3000	3000	6000	174000	5.568	13.92
160	3200	3200	6400	185600	5.9392	14.848
170	3400	3400	6800	197200	6.3104	15.776
180	3600	3600	7200	208800	6.6816	16.704
190	3800	3800	7600	220400	7.0528	17.632
200	4000	4000	8000	232000	7.424	18.56

Formula (12) Re-join Response Message

	Re-join Message			Re-join Response Message		
Number of Node	Number of Messages	Number of Bytes	Time (second)	Number of Messages	Number of Bytes	Time (second)
2	2	4	0.000128	2	12	0.000384
3	3	6	0.000192	3	18	0.000576
4	4	8	0.000256	4	24	0.000768
5	5	10	0.00032	5	30	0.00096
6	6	12	0.000384	6	36	0.001152
7	7	14	0.000448	7	42	0.001344

8	8	16	0.000512	8	48	0.001536
9	9	18	0.000576	9	54	0.001728
10	10	20	0.00064	10	60	0.00192
11	11	22	0.000704	11	66	0.002112
12	12	24	0.000768	12	72	0.002304
13	13	26	0.000832	13	78	0.002496
14	14	28	0.000896	14	84	0.002688
15	15	30	0.00096	15	90	0.00288
16	16	32	0.001024	16	96	0.003072
17	17	34	0.001088	16	96	0.003072
18	18	36	0.001152	16	96	0.003072
19	19	38	0.001216	16	96	0.003072
20	20	40	0.00128	16	96	0.003072
21	21	42	0.001344	16	96	0.003072
22	22	44	0.001408	16	96	0.003072
23	23	46	0.001472	16	96	0.003072
24	24	48	0.001536	16	96	0.003072
25	25	50	0.0016	16	96	0.003072

Formula (13) Maintenance messages

node	policy repository size	network total policies	Network tree levels	Number of Maintenance messages
2	20	40	3	60
5	20	100	3	240
10	20	200	3	540

20	20	400	3	1140
30	20	600	3	1740
40	20	800	3	2340
50	20	1000	3	2940
60	20	1200	3	3540
70	20	1400	3	4140
80	20	1600	3	4740
90	20	1800	3	5340
100	20	2000	3	5940
110	20	2200	3	6540
120	20	2400	3	7140
130	20	2600	3	7740
140	20	2800	3	8340
150	20	3000	3	8940
160	20	3200	3	9540
170	20	3400	3	10140
180	20	3600	3	10740
190	20	3800	3	11340
200	20	4000	3	11940

Formula (15) administrative messages

node	policy repository size	network total policies	Network tree levels	Total Number of administrative messages
2	20	40	3	40

5	20	100	3	100
10	20	200	3	200
20	20	400	3	400
30	20	600	3	600
40	20	800	3	800
50	20	1000	3	1000
60	20	1200	3	1200
70	20	1400	3	1400
80	20	1600	3	1600
90	20	1800	3	1800
100	20	2000	3	2000
110	20	2200	3	2200
120	20	2400	3	2400
130	20	2600	3	2600
140	20	2800	3	2800
150	20	3000	3	3000
160	20	3200	3	3200
170	20	3400	3	3400
180	20	3600	3	3600
190	20	3800	3	3800
200	20	4000	3	4000

Appendix C Publications

- Nidal Qwasmi, Daniel Smullen, Ramiro Liscano, "Integrated development environment for debugging policy-based applications in wireless sensor networks," in proc., *4th Int. Conf. Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2013)*, October 21-24, 2013, Niagara Falls, Ontario, Canada.
- Nidal Qwasmi, Khalil El-Khatib, Ramiro Liscano, Julie Thorpe (2013), "Privacy Policy Negotiation Architecture for Pervasive Computing Environments," ThinkMind digital Library 55-60, in proc., *3rd Int. Conf. Advanced Collaborative Networks, Systems, and Applications (COLLA 2013)*, Nice, France, July 21 - 26, 2013, ISBN: 978-1-61208-287-5.
- Nidal Qwasmi, Ramiro Liscano, "Bloom filter Supporting Distributed Policy-Based Management in Wireless Sensor Networks," *Procedia Computer Science*, Volume 19, 2013, Pages 248-255, ISSN 1877-0509.
- Poster presentation at *Consortium for Software Engineering Research (CSER) 2012* titled "Framework For Distributed Policy-Based Management in Wireless Sensor Network," November 4, 2012, Markham, Ontario, Canada.
- Nidal Qwasmi, Ramiro Liscano, "Framework for Distributed Policy-Based Management in Wireless Sensor Networks to Support Autonomic Behavior," *3rd Int. Conf. Ambient Systems, Networks and Technologies (ANT-2012), Procedia Computer Science*, Volume 10, 2012, Pages 232-239.
- Nidal Qwasmi, Ramiro Liscano, "Distributed Policy-Based Management for Wireless Sensor Networks," *Procedia Computer Science*, Volume 10, 2012, Pages 1208-1212, ISSN 1877-0509.
- Nidal Qwasmi, "Policy-Based Management in Wireless Sensor Networks to Support Autonomic Behavior," presentation of paper, *UOIT 3rd Annual Graduate Student Research Conference*, April 2012.
- Nidal Qwasmi, Fayyaz Ahmed, Ramiro Liscano, "Simulation of DDOS Attacks on P2P Networks," *Proc. 2011 IEEE Int. Conf. High Performance Computing and Communications (HPCC '11)*, 610-614, Banff, Canada. September 2-4, 2011.
- Mukhtaj S Barhm, Nidal Qwasmi, Faisal Z Qureshi et al., "Negotiating Privacy Preferences in Video Surveillance Systems," *Proc. 24th Int. Conf. Industrial Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE 2011)*, 511-521, 2011.
- As a contribution back to the WSN research community, our research was used as a basis for other open source projects, such as [15] and [16], which inspired many other researchers abroad.