UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Mechanisms for Network State Management And Data Capture in MultiNetworks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Ranga Raj

Dissertation Committee:
Nalini Venkatasubramanian, Chair
Sharad Mehrotra
Marco Levorato

2015

DEDICATION

To
My mom and dad, wife and son.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

ABSTRACT OF THE DISSERTATION **viii**

1 INTRODUCTION **1**

2 MINA DB LAYER ENHANCEMENTS **7**

2.1 MINA System Overview 8

2.1.1 MINA Server Architecture 10

2.1.2 Current MINA Data Model 12

2.1.3 Database Access Layer 13

2.2 Redesigned Persistent Data Model for MINA 14

2.2.1 Known Issues with Current MINA Implementation 14

2.2.2 Revised MINA Data Model 15

2.2.3 In-memory Representation for MINA using JUNG 16

2.2.4 Retrofit MINA with revised changes 17

2.2.5 Related Work on Persistence Layer Implementations 19

2.2.6 Database Access Layer Implementation 21

2.2.7 Revised program flow for MINA 22

3 STRESS TESTS ON MINA DB ACCESS LAYER **24**

3.1 Experimental Setup 24

3.2 Overall Test Strategy 25

3.3 Testing using dataset from CRAWDAD 26

3.3.1 Test Methodology using CRAWDAD dataset 27

3.3.2 Test Results from CRAWDAD dataset 28

3.3.3 Interpretation of the results from CRAWDAD datasets 29

3.4 Testing using Random-generated Test Data 29

3.4.1 Test Methodology using Random-generated Data 30

3.4.2 Test Results from Random-generated Data 31

3.4.3 Interpretation of the results from Random-generated Data 37

4 EXTENDING MINA FOR NODE MOBILITY **39**
4.1 The Scenario of a Shanty Town 39
4.1.1 Related Work - Data Gathering for Shanty Towns 40
4.2 Topology for Shanty Town with MINA 41
4.3 Need for Node Mobility 45
4.3.1 Shanty Town Road Network as a Graph 46
4.3.2 Emergency Request as an Event 47
4.3.3 Modeling the Mobility of Heterogeneous MDCs 48
4.3.4 Problem Statement 52
4.4 Path Re-Scheduling Algorithms 54
4.4.1 Summary of assumptions for all heuristics 54
4.4.2 Nearest Walk - A Naive Approach 55
4.4.3 Minimum Deviated Walk - A Local Search Approach 56
4.4.4 Ortho Walk - A Greedy Approach 57
4.5 Extending MINA System Architecture 60
4.5.1 System Architecture for an Emergency Response System in a Shanty Town 60
4.6 Putting MINA and Mobility Planning Together 63
4.6.1 Use Case 1 - Handling Shanty Town Emergency/Disaster Response . 66
4.6.2 Use Case 2 - Location Update of MDC 67
4.6.3 Benefits of Mobility Planning to MINA 68

5 CONCLUSION **69**

Bibliography **73**

A Detailed Test Results **75**
A.1 Test Results from Random-Generated Test Data 76

LIST OF FIGURES

Page

2.1 MINA Overlay Hierarchy . . . 9
2.2 MINA Server Software Stack . . . 10
2.3 MINA Server Component Interactions . . . 11
2.4 MINA Network Information Base UML Diagram . . . 12
2.5 Revised MINA Persistent Data Model . . . 16
2.6 Retrofit MINA Server Architecture . . . 18
2.7 Revised MINA DAO Layer Detail . . . 18
2.8 MINA DAO Implementation . . . 22
2.9 Revised MINA Application Flow . . . 23

3.1 Time to Initialize Network Of Different Sizes . . . 33
3.2 Topology Update Time vs. Node Size (Fixed Update Load) . . . 34
3.3 Topology Update Time vs. Volume of Updates (Fixed Start Network Size) . . . 35
3.4 Random Query Process Time vs. Volume of Updates (Fixed Start Network Size) . . . 36

4.1 Shanty Town Network Topology with MINA . . . 44
4.2 Graph Network for Dharavi . . . 47
4.3 Ortho Walk Run (For a sample input of 5 events at V2, V3, V5, V6, V8 and 3 MDCs at V1, V7 and V4) . . . 59
4.4 System Architecture Overview of Shanty Town Emergency Response . . . 61
4.5 Use Case 1 - Handling Shanty Town Emergency/Disaster Response . . . 64
4.6 Use Case 2 - Location Update of MDC . . . 65

LIST OF TABLES

Page

3.1 Performance of MINA using CRAWDAD's vt/maniac topology traces 28

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my committee chair, Prof. Nalini Venkatasubramanian for her guidance, encouragement and support. I would like to thank my thesis committee members Prof. Sharad Mehrotra and Prof. Marco Levorato for reviewing my work.

I would like to thank my mom and dad for all their prayers and blessings, to my wife and son for all their encouragement and sacrifices for my sake since I started this journey.

I would like to thank Raju Dantuluri Ph.D., Steven Birch (WON+Co.), Tom Tonkin Ph.D. and Ty Duval, my mentors for all their guidance and support.

I would like to thank the PC3 Team of Prof. Nalini Venkatasubramanian, Prof. B. S. Manoj(IIST), Kyle Benson(UCI), Sarath Babu(IIST) and Gaurav Jain(IIST). Thank you all for explicitly allowing me to use our work for a portion of this thesis. I would like to make an explicit mention that Sarath, Gaurav and Manoj put together the Deviated Walk and Nearest Walk that features in this thesis. We use it only as a comparison to my work on Ortho Walk. The entire team contributed to perfecting the solutions we proposed. The time we all spent was enlightening, enjoyable and satisfying. I learned a lot from that experience and I sincerely hope we have many more such opportunities.

I would like to thank Zhijing Qin, Ph.D. who provided me valuable guidance on all aspects of MINA. Your patience is immense and your depth of knowledge on the subject is commendable. I wish you the very best and hope you have a bright future in the industry. I would like to thank my fellow researchers from the DSM group Kyle Benson, Guoxi Wang, Zhu Qiuxi, Phu Nguyen, Qing Han and Nailah Alhassoun for sitting through all my presentations and giving me valuable feedback.

I would like to call out once more to Kyle Benson for patiently walking me through this entire research process. You were my go-to guy for practically anything and everything.

I wish to thank all the faculty who have encouraged me every step of my way here on this journey, in particular, Prof. Michael Carey, Prof. Sharad Mehrotra, Prof. Crista Lopes, Prof. Michael Dillencourt and all others who let me sit through their lectures.

Once more, my thanks to Prof. Nalini Venkatasubramanian for her invaluable feedback and comments through out the course of this project. I am looking forward to doing more work as a part of this group under your guidance.

ABSTRACT OF THE DISSERTATION

Efficient Mechanisms for Network State Management And Data Capture in MultiNetworks

By

Ranga Raj

Master of Science in Computer Science

University of California, Irvine, 2015

Nalini Venkatasubramanian, Chair

Multinetwork INformation Architecture(MINA) is a reflective (self-observing and adapting) middleware approach to manage such dynamic and heterogeneous multi-networks in pervasive environments. MINA depends upon a strong foundation for maintaining the network state at all times. It needs to have a robust platform that performs its functions from small or large topologies alike.

This thesis describes the extensions made to the database access layer of the MINA. Modifications were made to the database schema as well as the database capture layer to capture these periodic or intermittent changes of the network state. We tested the modifications so that MINA scales to handle a large topology. We have also enhanced the runtime component and show that an in-memory representation of the multi-network provides very predictable responses to network path queries. MINA can now use the runtime enhancements made and provide accurate predictions of the network bottlenecks and accurately predict other what-if scenarios.

In order to perform a more realistic test on the code changes, we used publicly available datasets. Node mobility information was captured at single-second intervals observed from an access point. We successfully tested the capability to handle large volumes of changes to the network. We then generated data for larger topologies as well as large batches of changes to network state to see how the modifications handle such stresses. We measured the time taken to persist such large volumes of changes.This gives us an idea of the frequency of changes that MINA can accommodate.

We then show one possible extension of MINA. We propose a suitable deployment for emergency response in a shanty town. We assume that we instrument the entire area of the shanty town with low-cost sensors capable of capturing local parameters like air quality, smoke, temperature etc. Additional network components like mobile nodes and fixed access points or routers perform the task of capturing any relevant sensor data and routing it to a central location (like an Incident Response Center) for further action. We discuss an extension where the mobile nodes are assigned a path on a road network thus directing the mobility. We present efficient techniques to assign a path to these nodes so that the time taken to capture the data from sensor nodes is minimized. We formulate the task as a Dynamic Vehicle Routing Problem with Time Windows constraints which is NP-hard. We present heuristic based algorithms to generate an efficient path for such mobile nodes. We show how such potential extensions can be incorporated into the MINA design framework to solve real-world problems.

Chapter 1

INTRODUCTION

Multi-networks have become increasingly popular in real-world solutions to networking problems. Some authors refer to this as spontaneous networking or multi-hop networks. The term multi-networks usually refers to an implementation where nodes opportunistically make peer-to-peer contacts and share data and resources in an impromptu way[6]. Such a solution takes minimal infrastructure setup for data to be transferred from a source to sink. However, the opportunistic nature of the connections makes the task of managing the network a challenge. The dynamic nature of traffic between nodes and the dynamic nature of connectivity, at times multiple connections as long as the technology allows it, makes the task of managing the Quality of Service(QoS) over the network very challenging.

MINA and its Underlying Philosophy

MINA (Multi-network INformation Architecture) is a reflective (self-observing and adapting) middleware approach to manage dynamic and heterogeneous multi-networks in pervasive environments[18]. It was developed by a team of researchers working under Prof. Venkatasubramanian at UCI. It uses a MySQL database backend to persist state information that is queried periodically to realize the different functions it supports. A key aspect of MINA is that it implements an *Observe-Analyze-Adapt* (OAA) loop. It has the ability to capture the topology and the current network state. It uses this data to analyze performance bottlenecks and if required, provide instructions for alternate routing of packets or adding additional resources to ensure end-user QoS.

MINA uses an overlay network to discover the network topology and actively collects information on all the links associated with each node as a part of the *Observe* process. It persists the information in a MySQL database. The current design of MINA uses this database to read and provide the capability to *Analyze* the network for potential bottlenecks. It uses the *Adapt* paradigm to suggest appropriate network routes that satisfies end-user QoS requirements. MINA integrates with Software Defined Network (SDN) equipment to realize the *Adapt* part of its functionality to adjust the routing and thereby ensure desired throughput.

Database Access Layer Modifications

In this thesis we describe the extensions made to the Database Access layer of the MINA. We designed a new database schema to capture the network topology information that needs to be persisted. By keeping the stored information concise, we provide MINA enough data to generate an in-memory representation of the topology. We implemented the new Database Access layer using Java. We used the Java Universal Network/Graph Framework (JUNGv2) [2] library built by a team from UCI to represent the network topology in-memory. An in-memory representation of the network topology as a graph allows us to provide a faster response to any network path related queries. Our Data Access layer encapsulated all the SQL calls to the database. This allows us to optimize the database queries easily as appropriate.

Stress Testing of MINA DB Access Layer Changes

To validate the performance of the changes, we used datasets from CRAWDAD [9] to get some real-life node mobility data. This allowed us to test the changes to a real network topology. We then hand-generated a very large topology to test with and measured the throughput in terms of volume of network link changes that can be supported as well as measuring the performance on large networks. We show that an in-memory representation of the multi-network provides very predictable responses to network path queries. MINA can now use the runtime enhancements made and provide accurate predictions of the network bottlenecks and accurately predict other what-if scenarios.

Extending MINA - Adding a Mobility Model

We then show a unique application of MINA in the real-world where it plays a significant role - a shanty town's emergency response system. One can build an effective system using different classes of nodes with differing capabilities and on-board resources. The entire region could be instrumented with a large farm of low-cost sensor nodes available to capture data on demand on various parameters. We can incorporate Mobile Data Collectors(MDCs) and fixed networking equipment as necessary, all working with a remote MINA server at a Control Center(CC) and having the ability to analyze any received data. Emergency situations called *events* can be registered anywhere in the shanty town and the nearby sensors would capture the vital details like air quality, smoke, temperature etc. However, the sensors need to wait for an available MDC to pick up and send as quickly as possible to the CC. The QoS requirement of such a topology necessitates providing adequate coverage to capture data from the sensors locally and have sufficient number of MDCs available to transport the data to the server for further analysis within the shortest possible time. In such an emergency response system, the sooner this first-response detail is sent the faster an appropriate relief can be dispatched by the CC.

Completing the shanty town emergency response scenario, providing a reliable response to emergency requests requires two critical parts: (1)Knowledge of the Event Occurrence i.e. location and time and (2)An available MDC needs to travel to the location to capture sensor data within the fastest time and relay the sensor data to the CC. In this thesis we do not address how the CC gets to know that a specific event location needs to be serviced. We assume that the CC receives the coordinates of the *event location* through some external means. We use the term *event expiration* to indicate a time window within which the sensor data needs to be captured. Here, we have addressed the part of supplying the best route among the available MDCs (i.e. assigned mobility to the MDC) in the region to capture the sensor data.

We present efficient techniques to assign a mobility model to the nodes so as to minimize the time taken to capture the data from the sensor nodes. We formulate the mobility model as a Dynamic Vehicle Routing Problem with Time Windows constraints which is NP-hard. We then present heuristics to generate an efficient path for these mobile nodes.

MINA plays an important role in ensuring resilience to network changes. It can advise the network administration staff with recommendations on additional equipment like APs, mobile nodes or routers to ensure that the mobile nodes have an optimal route to the CC even when large parts of the network get disconnected. In the normal course of functioning MINA maintains an accurate picture of the entire deployment based on the information received by the Observe loop.

We conclude this thesis with a discussion on how the MINA's data framework can be extended to accommodate this shanty town scenario or other real-world problems. The CC runs the MINA Tier 1 node and instead of passively recording the node mobility as the mobile nodes move, MINA can assign the mobility path to the mobile nodes and obtain real-time feedback from them as they move.

Contributions in this Thesis

The following contributions are outlined in this thesis:

- We designed and implemented a more robust data management backend for MINA that is capable of handling the dynamic nature of the underlyng infrastructure that large network infrastructures demand.
- We studied the performance of our code changes using a publicly available dataset collection from a WiFi Access Point recording mobile users entering or leaving a region. We present are results here.
- To test the changes further for large volumes of network changes, we created sample test data generating adding or remove network changes at random on a fairly large network topology. We tested the performance of the layer using this data. We present them here.
- We then apply the MINA framework to a more dynamic use case of mobile data collectors operating in a shanty town for emergency response. We provide efficient algorithms for the mobility of the nodes to allow efficient data capture on an overlapping road-network using heterogeneous MDCs. We show how the design of MINA can be extended to this application.

Structure of this Thesis

This thesis is structured as below:

- Chapter 2 discusses the details of the changes to the MINA backend such that it scales to large network updates as well as accommodate a large network topology. We present the software architecture of the DB Access layer here. We also present the schema design and justifications for all these changes.

- Chapter 3 discusses the experiments on the modified MINA Data Access layer. We first present the results of the CRAWDAD dataset used. We then discuss the details of the random-generated dataset and show results indicating that the implementation can handle a large network topology.

- Chapter 4 discusses an application of MINA in the real-world scenario. We discuss the details of a shanty town scenario and present a design where MINA can be extended to take in real-time input for targeting the mobility of the mobile nodes and the existing framework of MINA provides the network management capabilities. We present the mathematical formulation of the shanty town's dynamic event data collection task and finally describe the efficient algorithms for the mobility of the nodes.

- Chapter 5 presents our conclusions.

Chapter 2

MINA DB LAYER ENHANCEMENTS

The primary objectives of MINA are:

1. Efficiently organize heterogeneous devices in multinetworks.
2. Obtain a global and fresh view of the network state as needed.
3. Analyze the network state based on the global view, and get a deeper understanding of the network properties, e.g. resilience, congestion.
4. Adapt or Control the network so that application needs can be met under the network resource constraints.

In order to satisfy these objectives, data is collected and stored by the MINA backend in a MySQL server. In this chapter, we discuss the details of the changes to the MINA backend such that it scales to large network updates as well as accommodate a large network topology. We present the software architecture of the DB Access layer here. We also present a revised schema design and justifications for all these changes.

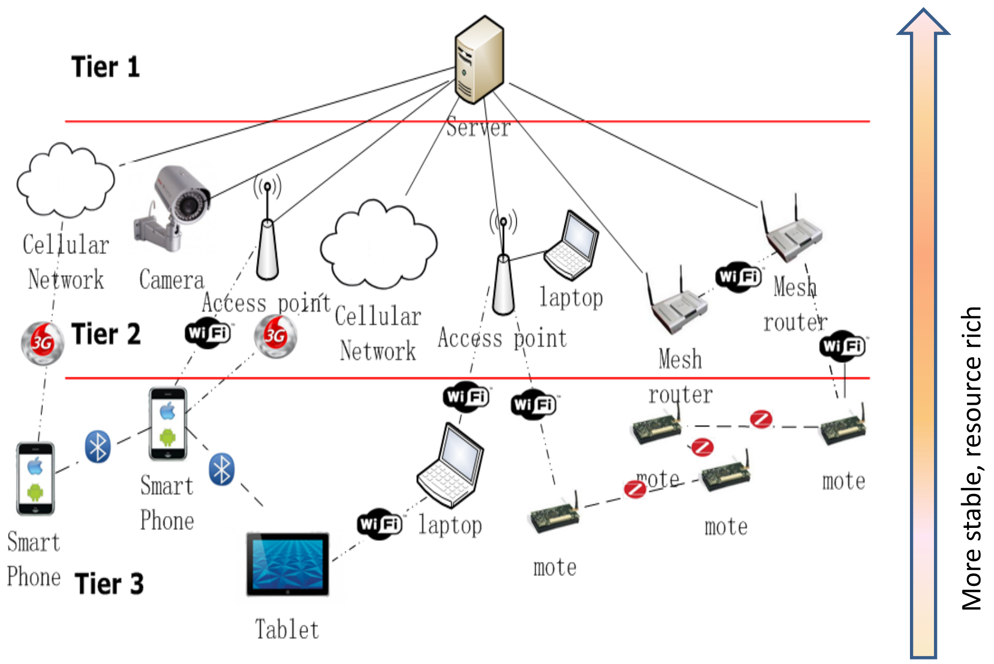
To put this into context, in the next few paragraphs we describe briefly the MINA framework at a high level and the current Data layer design. We follow this up with the enhancements and details of the implementation.

2.1 MINA System Overview

The main goal of the MINA architecture[10] was to create and maintain an overlay network above the existing heterogeneous links, in order to collect the underlying network state information about the devices themselves and the links they are communicating through. In system terms this involves reacting dynamically to instances of nodes joining and leaving the network. A logically centralized server is responsible for collecting and storing this information into a MySQL database. The system is designed using the OAA paradigm (Observe, Analyze, Adapt). It receives network state information from the devices, analyzes them, and possibly issues configuration commands, e.g., to recover from previously detected faults. Using this information, it can coordinate different networks more efficiently or perform management operations on specific network devices.

The first step in MINA is to accomplish the creation of the overlay network. This is an organization of network nodes or devices in a hierarchical manner and is distinct from the underlying network topology. The hierarchy of the overlay is maintained very shallow, just 3-4 levels deep to ensure minimal overhead as well as better scalability and robustness to links/nodes failures due to the node mobility. The hierarchy of the nodes is based on the stability of the network and computing resources the nodes have. Please see Fig. 2.1.

MINA handles scalability in its current design by applying a tree-based *hierarchical overlay model* for the nodes in the network. The highest level, i.e. *Tier 1* is the most stable and



- Exploits the diverse capabilities of the network nodes
- More effectively support node mobility

Figure 2.1: MINA Overlay Hierarchy

is the central server, resource-rich and aggregates information from lower and less stable layers. *Tier 2* comprises of nodes which are stationary and have sufficient CPU and network resources. These nodes aid in the routing of information from the lower layers as well as provide redundant connectivity to the rest of the network to handle large failures. Typically, these nodes are Access Points(APs) or Routers. A third category of nodes, the mobile nodes called *Tier 3* are less capable than Tier 2 in terms of network connection stability. Such nodes may move in and out of range of the APs. They typically have WiFi connectivity to communicate with other nodes which can cause them to move from one AP to another. Sensor nodes or motes are expected to have very limited connectivity and storage capability. MINA classifies them again as Tier 3 nodes which are in-turn connected to the Tier 3 mobile nodes. They have the necessary networking component to talk to the sensors as well as the Tier 2 backend network.

2.1.1 MINA Server Architecture

The server architecture and the interactions between the various components of MINA is described in Fig. 2.2 and Fig. 2.3. More details are available in [10].

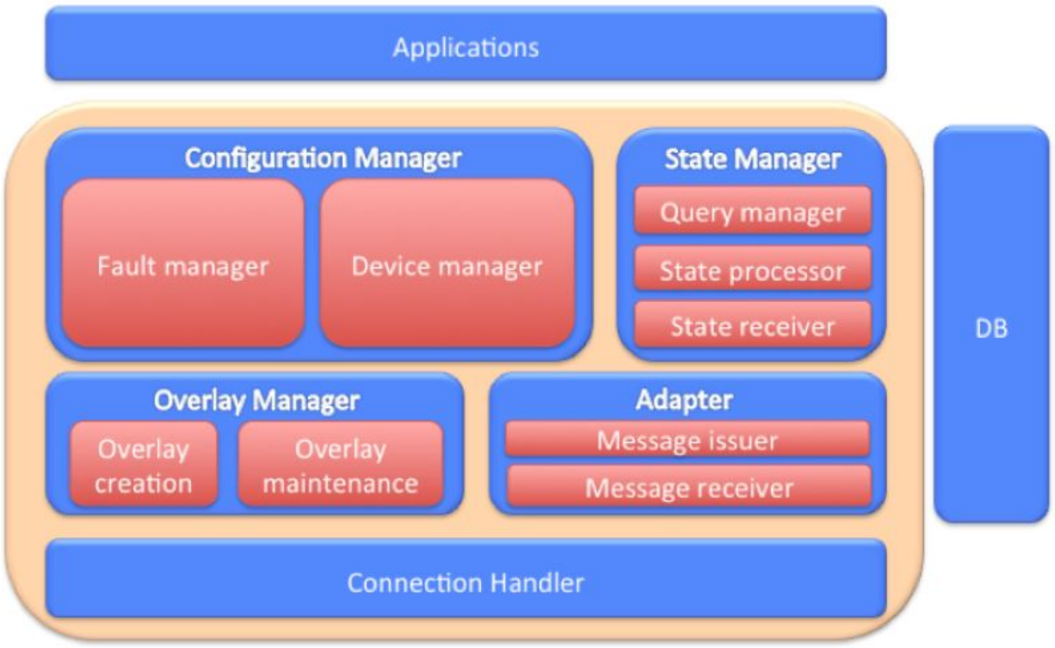


Figure 2.2: MINA Server Software Stack

The *Overlay Manager* in the software stack running on the server side coordinates the overlay network construction and continuously collects the network state information in order to actively monitor the managed environment. Collected state is also persisted into a database, the *NIB* and thus facilitates many analysis tasks e.g., detect possible faults and unexpected behaviors, apply formal reasoning methods for fault detection, etc.

The flow of logic on the server is as below:

- Read a startup configuration file and establish connections with Tier 2 nodes (or dynamic discovery, e.g., broadcast-based discovery). In reality, this is the relatively stable portion of the network and can be easily read from some input to speed up the topology

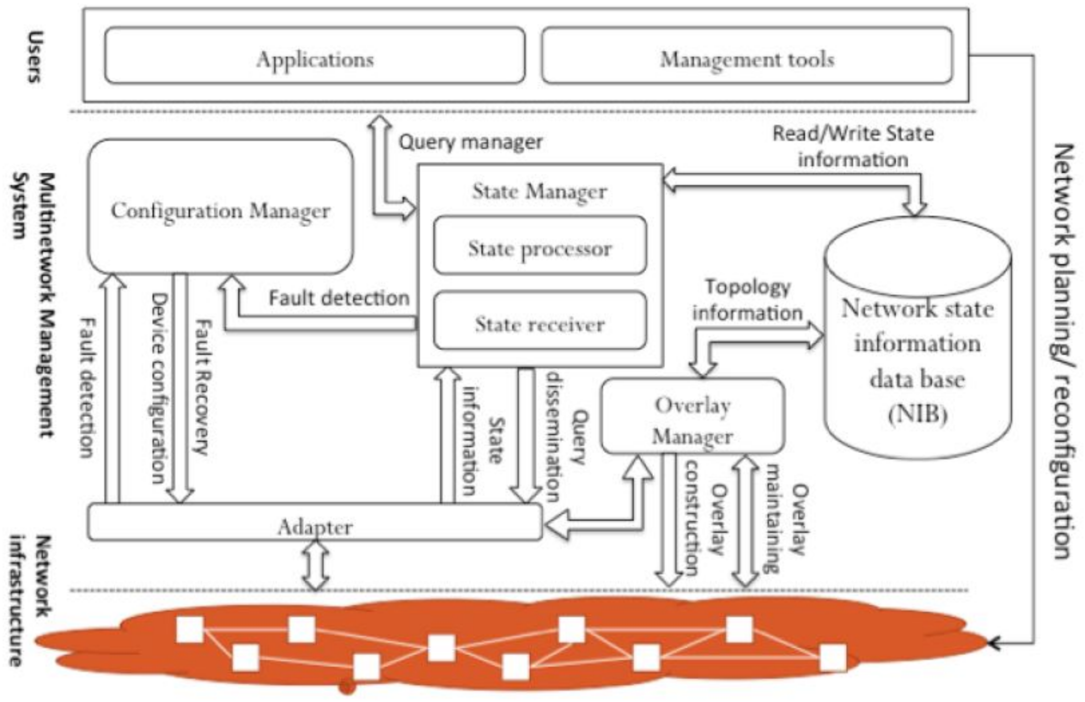


Figure 2.3: MINA Server Component Interactions

discovery.

- Start the overlay construction algorithm.
- Invoke the state information (link/node/application state) collection process by disseminating queries to the clients (Observe).
- Persist collected state information into the database
- Analyze the state (Analyze) and, depending on the results, provide configuration changes to the nodes (Adapt).
- Discover faults (explicit/implicit) in the network (Analyze) and, depending on the discovery and reasoning results, perform fault recovery operations (Adapt).

2.1.2 Current MINA Data Model

We particularly focus on the interactions between the *Network state Information Base (NIB)*, the *Overlay Manager* and *State Manager*. The main goal of this component is to maintain a global picture of the Multinetwork topology. The *NIB* is an Object-oriented representation of the network state. Please see Fig. 2.4. Briefly, the following paragraph describes the NIB.

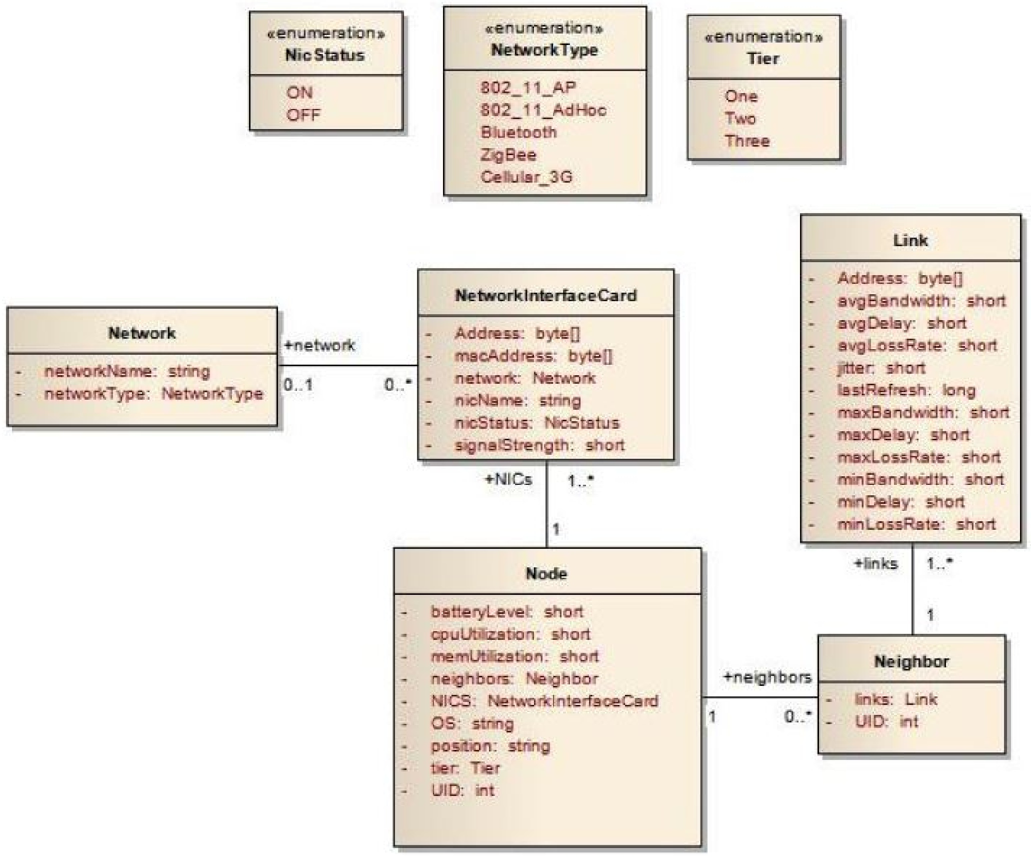


Figure 2.4: MINA Network Information Base UML Diagram

Node is the entry-point class of the NIB. It models a managed device and it is used as starting point for navigating through the entire graph. Every Node is distinguished by a UID (Unique IDentifier). this UID is generated based on the first Network Interface Card (NIC) attached to the Node. The MAC address is a 6-byte address that is unique for each NIC and is set by the manufacturer guaranteeing that each Node has it own unique identifier. The enumeration Tier describes which Tier (in the overlay network) the Node belongs to.

This can be typically 1, 2 or 3. Each Node has a 1-to-N association with *Neighbor* and *NIC*. *NetworkInterfaceCard* models one network interface card available locally for this Node. It is characterized by an address (IP address), a macAddress, and other attributes. The enumeration nicStatus refers to the actual status of the network interface (it can be either ON or OFF). Each *NetworkInterfaceCard* is associated with a *Network*. *Network* represents a network available in the managed Multinetwork environment. *Neighbor* is a one-hop distance device reachable from this Node. Each neighbor has a 1-to-N association with *Link*. *Link* models a physical link with a *Neighbor* in order to collect network information. Each Link is characterized by an identifier and some metrics that model the channel from the network perspective (e.g., bandwidth, delay, and lossRate).

2.1.3 Database Access Layer

The *NIB* is implemented on a MySQL database. In order to access the data within the MySQL database and access it or manipulate as a java object, the current design uses a Database Access Layer called Persistence Manager. This cooperates with the State Manager component of the server to persist the state information in the NIB. Java Persistence API (JPA) [7] technology has been used here to map these java objects at runtime into equivalent artifacts in the MySQL database.

The Java Persistence API, sometimes referred to as JPA, is a Java programming language framework managing relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition. A persistence entity is a lightweight Java Class whose state is typically persisted to a table in a relational database. Instances of such an entity correspond to individual rows in the table. Entities typically have relationships with other entities, and these relationships are expressed through object/relational metadata. Object/relational metadata is specified directly in the entity class file by using Java Annotations (e.g., @Entity), or in a separate XML descriptor file distributed with the application. JPA, however, is only the definition of a common Java standard interface aimed to handle the Object Relational Mapping (ORM). The current implementation of MINA uses Hibernate[16], which supplies an open-source implementation of the Java Persistence API.

2.2 Redesigned Persistent Data Model for MINA

2.2.1 Known Issues with Current MINA Implementation

The current version of MINA has been tested for a relatively small deployment of less than 50 *Tier 3* nodes, 2 *Tier 2* nodes and one *Tier 1* node. While it validated the concept very well, tests revealed that the Observe and Analyze phase of MINA were severely impacted by the database design.

A downside of the JPA implementation using Hibernate used in MINA for the persistence layer is that it encapsulated the database layer totally. As a result, there is no opportunity at the database level to intercept the query and send a more optimal SQL (suited for a Relational Database) for execution to the database. In addition, practical usage of Hibernate requires careful instantiations of dependent objects (lazy instantiations). If not coded properly, queries can fetch a lot more data than required rendering the system inefficient.

Current version of MINA uses an in-memory representation of the information consisting of chained linked lists of Neighbors and Child nodes for reference. Any node movement in the physical topology caused changes to the membership of these linked lists. Since these objects were linked to the Object-relational mappings, a number of calls were wasted to the database.

As a result, the team noticed a very slow performance when it came to the Observe and

Analyze phase of MINA even for very small topologies.

In this thesis, we revisit the existing class model of MINA for the persistence layer and make it more efficient for implementation on a relational database. We eliminated some java classes and replaced others with queries that we executed at runtime.

2.2.2 Revised MINA Data Model

Fig. 2.5 presents the modified Persistent Data Model. The current design of the NIB persisted the Neighbor information separately and associated a Link entity to each such entry. In this design, we have effectively combined these two entities to just a new Link entity.

The Link entity represents an association between two Node entities. Physically, this represents a connection between two Nodes and is qualified further as an association between the MAC addresses of the two NIC cards that form this link. The Link entity is further described with details such as: bandwidth, delay, jitter and lossrate.

The distinct advantage with this design is that the changing of a link association between any two Nodes can be easily represented by changing just one entity. Maintaining a list of Neighbors to a give nodeUID can be a simple query on the Link entity to fetch all the nodes associated with the nodeUID of the NICs. The other benefit is that no further maintenance of the Link entity is required if the physical links are added or removed.

The revised schema matches the data model shown in Fig. 2.5.

These changes along with the Software architecture changes described in the next section helped improve the performance and ensured the database access layer performs satisfactorily.

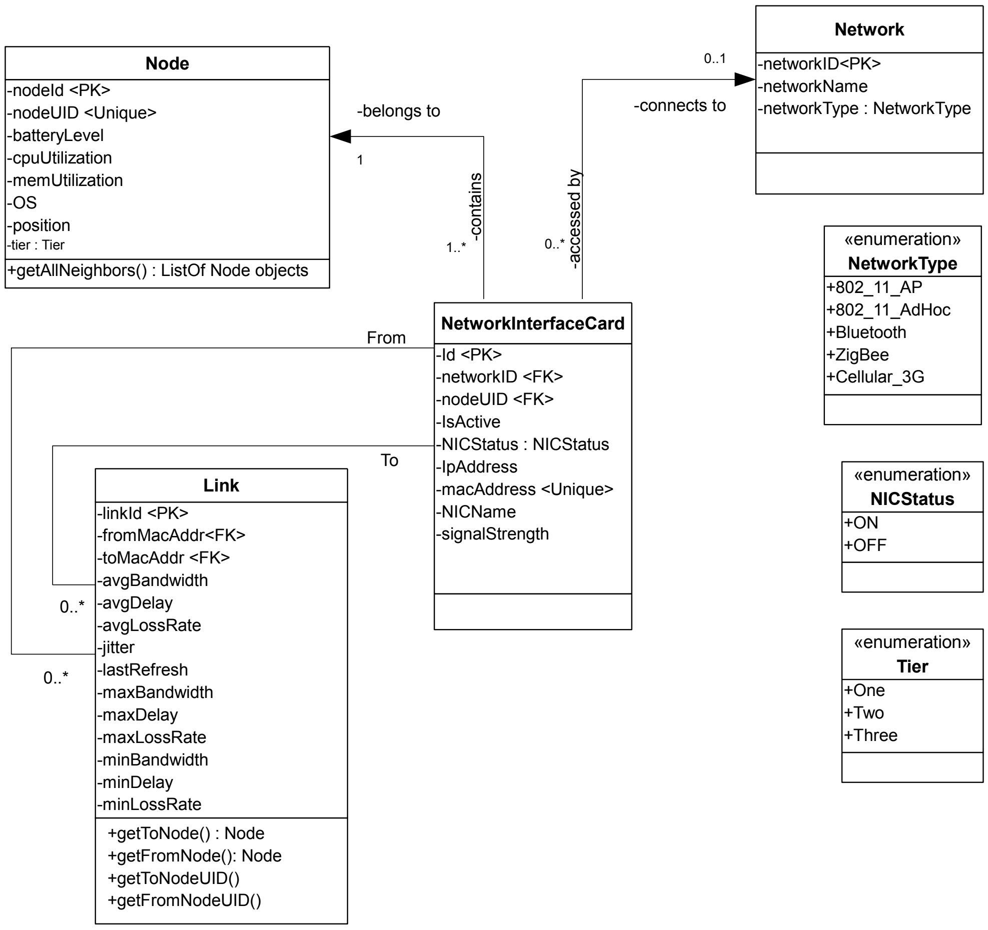


Figure 2.5: Revised MINA Persistent Data Model

2.2.3 In-memory Representation for MINA using JUNG

In the beginning of this subsection, we discussed the known issues with the current version of MINA. One of them was that for each operation in the Analyze phase or even in the Observe phase when topology change had to be modified, the program made repeated calls to the MySQL database.

In this section, we introduce a major change. We propose the use of an in-memory structure that represents the topology of the real network. The effort taken to maintain such a structure is paid back by the availability of graph functions on this structure. However, care must be taken to make sure that the persistent data is consistent with the in-memory graph.

On this effort, we considered a number of options including using graph databases. We settled on JUNG [2] which was a graph package developed by researchers at UCI. We felt it provided the desired functionality in a small footprint that we needed for MINA. We describe JUNG very briefly here.

JUNG the Java Universal Network/Graph Framework

JUNG is a Java library and allows easy representation and visualization of data that can be represented as a graph or network. JUNG-based applications can leverage the built-in capabilities of the graph operations. The JUNG architecture allows the representations of entities and their relations, such as directed and un-directed graphs, graphs with parallel edges and others. This was just what MINA needed. It provides a mechanism for annotating graphs, entities, and relations with metadata for visualization purposes. The current distribution of JUNG (v2.0.1) includes implementations of a number of algorithms from graph theory, data mining, and social network analysis, such as routines for clustering, decomposition, optimization, random graph generation, statistical analysis, and calculation of network distances, flows, and importance measures (centrality, PageRank etc.).

JUNG's visualization framework can be used for interactive exploration of the network if required. It is distributed as an open-source library.

2.2.4 Retrofit MINA with revised changes

Having identified the pieces that needed to be rewritten, added or changed, our next effort was to find the most optimal way to change the code of MINA. At this point, we have chosen to make a simple retrofit for MINA to improve its performance using the design changes above. To put this change into perspective, we show these changes in Fig. 2.6.

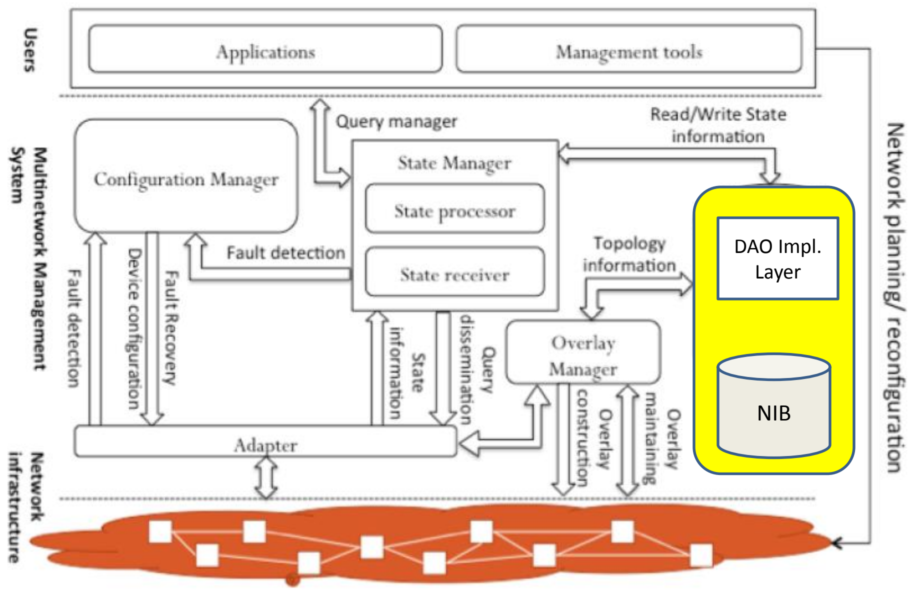


Figure 2.6: Retrofit MINA Server Architecture

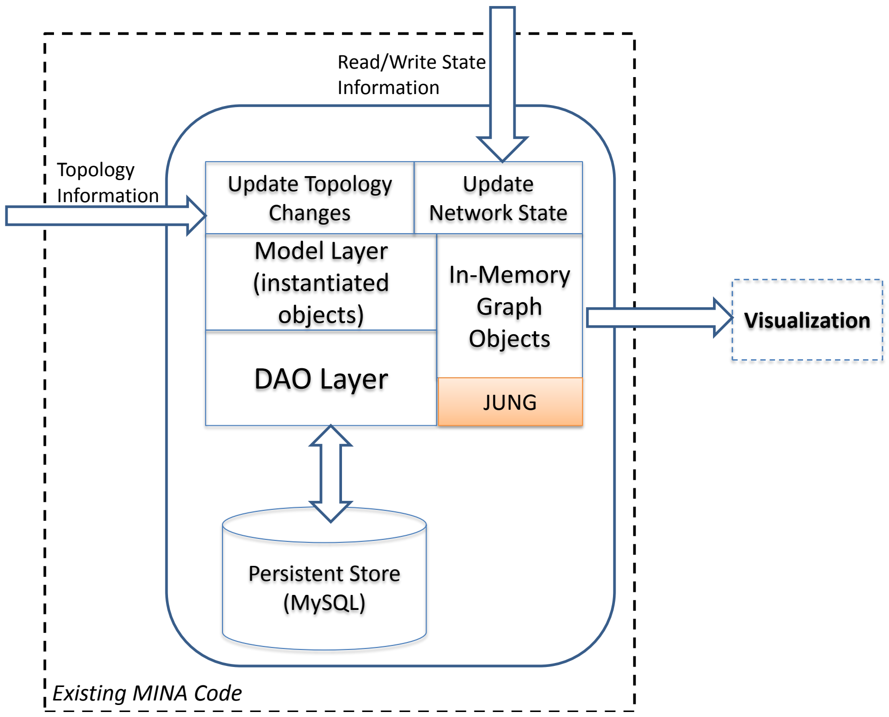


Figure 2.7: Revised MINA DAO Layer Detail

The purpose of the DAO Implementation layer is to encapsulate all the database accesses to the NIB. By using the JUNG library, we instantiate the In-memory graph object to keep the network representation during the program execution. Any object instance that is fetched from or written to the persistent store is in a Model Layer. Thus the current MINA code that processes the topology changes from the Overlay Manager or captures the network state from the State Manager manipulates the objects instantiated from the Model layer or the in-memory graph object. We show this detail in Fig. 2.7.

2.2.5 Related Work on Persistence Layer Implementations

A typical business application, such as an Account Management system needs some mechanism to appropriately register credit and debit transactions on the different accounts it manages over time. Each of these transactions happen in the context of a *session*. When the session has ended and the computation has been completed, the application needs some way to persist its current state. Most systems utilize an underlying database management system to persist state.

Persistence Layer is the software layer that makes it easier for a program to persist its state in the underlying database. The database can be relational or non-relational and each have their own specific ways to start a *session*. In order to perform the operations with the stored data, one needs a way to handle the *metadata* of the persistent information, *queries and expressions* to fetch the information stored and some form of support for *Transactions* which ensures atomicity of database operations as appropriate.

Monolithic application code design traditionally interspersed persistent storage access logic along with regular business logic. With the emergence of modular programming and object-oriented programming, the benefits of encapsulation and re-usability gained significance. A paradigm called *design patterns* came into prominence in the software design.

Design patterns are conceptual styles of code that can be typically applied to many object-oriented programming languages and allow standardization of the the program code. As a direct benefit of this approach, programs were more easily maintainable and once less error-prone. One such pattern in the discussion of a persistence layer is the *Data Access Object* paradigm. Programming best practice documentation of most groups usually spell out the details of such a layer.

Persistence Layer can be implemented using many different technologies today. Each technology offers the programmer varying amount of flexibility and encapsulation of underlying database accesses. Hibernate [16] is one such layer that was quite popular among Java programmers. It was designed to let Java programmers deal with the Java objects and leave the contract of persisting the information to the underlying framework. Hibernate used compiler directives and descriptor files to generate the necessary bindings in the code to implement the persistence logic. The Java Persistence API (JPA) provided a framework for such persistence layers to be built and included in the design of business applications.

Toplink is another persistence layer implementation and delivers a standards based enterprise Java solution for all relational and XML persistence needs based on high performance and scalability. It is currently available as Oracle Toplink.

Each of these technologies comes with its own merits and demerits and learning curves. The choice of persistence layer is more often made by the larger development team so as to leverage synergy during the development process.

We chose to pick neither but implemented just a generic Data Access Layer pattern using basic JDBC and SQL logic but ensured that we use the Data Access Object java implementation pattern.

2.2.6 Database Access Layer Implementation

In order to implement the Data Access Layer, we made use of a simple java pattern called: the Data Access Object pattern. The DAO pattern is used to separate low level data access logic or operations, in our case the JDBC and SQL logic from high level business services. This requires 3 pieces:

- Model Object or Value Object which is a simple Plain Old Java Object (POJO) containing getter and setter methods to store data retrieved using DAOImpl class defined below.
- Data Access Object(DAO) Interface where we define the standard operations to be performed on a model object(s).
- Data Access Object Impl(DAOImpl) concrete class which implements the DAO interface, a class which is responsible to get data from a data source which can be database or any other storage mechanism.

Based on these principles, for each object in the persistence layer, we implemented the DAO layer. This is described in 2.8. By instantiating the corresponding DAOImpl classes, we obtain access to the model objects and all the relevant methods. This allows us to encapsulate all the SQL logic within the DAOImpl classes and optimization or modifications of any DB related queries can be confined to a very small part of the code.

While this is not a perfect pattern, it is quite adequate for the demands of this project and hence considered.

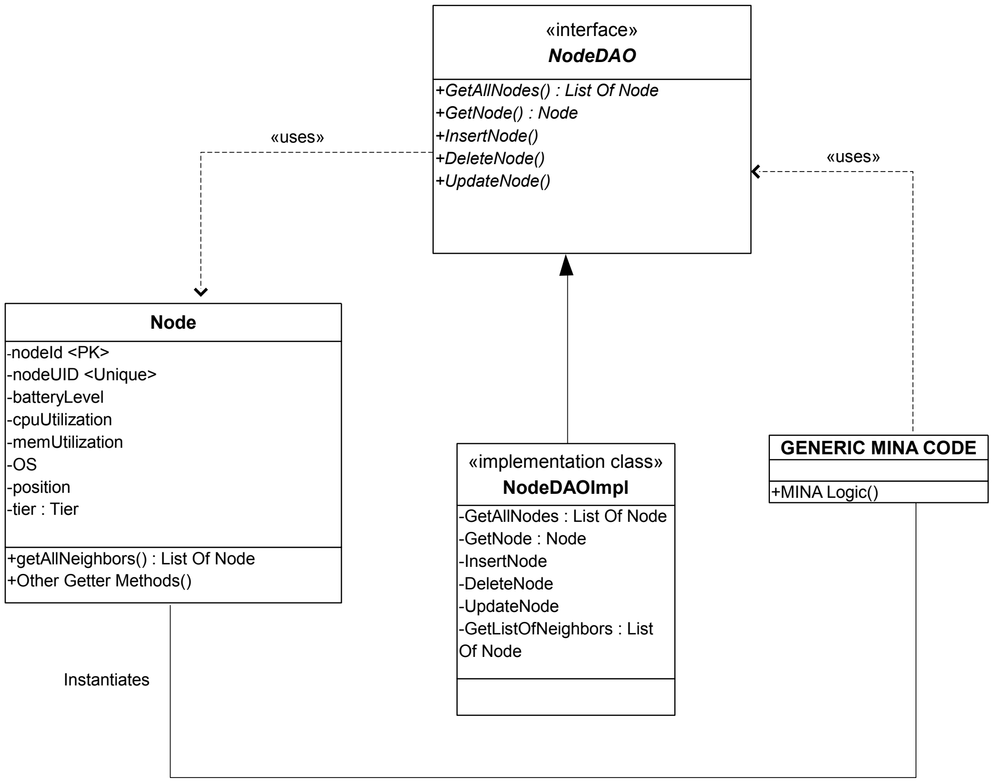


Figure 2.8: MINA DAO Implementation

2.2.7 Revised program flow for MINA

As a consequence of the code modifications made, the program flow for the revised MINA is described in Fig. 2.9. At the start of the MINA program, we expect that any persistent data about the previously known stable network state is first loaded from the NIB into a runtime in-memory graph object. Thereafter, MINA goes into a listen-mode where the network state is fed into the Update Network State function. This piece forwards the updates to the in-memory state of the network as well as to the database. On the same lines, the Update Topology Changes function listens to the topology changes and forwards the changes to both, the graph and the database.

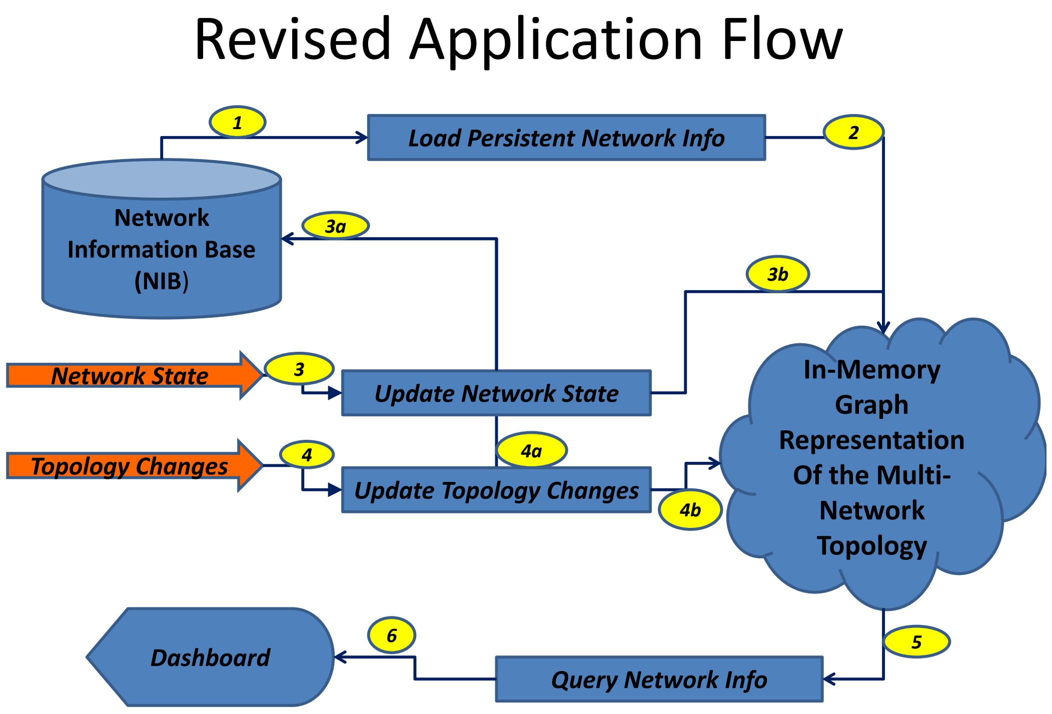


Figure 2.9: Revised MINA Application Flow

The ultimate objective is that we have a new *Query Network Information* capability that provides services related to queries on the network state which in-turn realizes this functionality from the in-memory graph.

Typical functions provided by this layer include:

- Fetch all current neighbors of a given node
- Obtain the path causing the least delay between two nodes
- Obtain the overall delay along the fastest path between two nodes

These functions are expected to improve the response of the system in the Observe and Analyze phases.

Chapter 3

STRESS TESTS ON MINA DB ACCESS LAYER

3.1 Experimental Setup

The revised MINA Database Access Layer is written and compiled using JDK 1.7. It uses JUNG v. 2.0.1 and the database is a single-instance MySQL 5.6 server. We chose to test the access layer separately as a part of this thesis. The codebase was executed on a Windows 7(64bit) OS running on an Intel Quad-core Pentium i7-2600 CPU@3.4Ghz processor and 8GB main memory.

Testing the MINA software with a real network input at realtime is difficult. Therefore, we built a test program that is capable of putting a varying amount of load on the Database Access Layer of MINA and as a result stress the database platform.

Our ideal setup would have been a stream input for these network state or topology changes. Controlling the number of changes sent via stream and measuring the time taken to process these changes would have been a good runtime test. However, we did not have any such stream of data that we could control the input rate.

We have therefore chosen to use a filestream as an input for all our tests and measured the time taken to process inputs of varying sizes. We measured the time taken for the system to initialize a previously stored persistent state of the network to get some ballpark estimate of how long MINA will take to initialize its network state.

We measured the time taken for an All-pairs shortest path on the graph during the tests periodically. At the end of each set of network updates processed through the system, we measured the time taken for a random query for a shortest path between any two nodes. Our intention was to give us an indication of degradation in performance with larger volumes if any.

We had the ability to initialize the database and repeat any specific run as many times as we needed.

3.2 Overall Test Strategy

As mentioned earlier, we could not test the revised code changes using a real stream input. In its placed, we assembled test files with data enough to run the same load atleast 25 times. A file input would test the robustness of the Database Access Layer just as well if not more while offering the ability to adjust the input load.

In order to test with some reasonable network load, we searched through the internet for public repositories for data that we could legitimately use for node mobility. We found a source in CRAWDAD [9]. In the coming subsections, we describe this test and the results.

After making sure that the database layer accepts the sustained load that the CRAWDAD dataset provided, we proceeded to stress test the system even further. We generated test data for networks of varying sizes and applied topology changes to such a network. Our intention was to stress the system with a very large load of topology changes and measure the time taken to absorb a fixed number of changes. Our motivation was to see how the database layer behaved with increasing load.

At the end of each set of topology changes, we run a query as mentioned before to validate that the performance of typical queries from Observe phase or Analysis phase did not degrade in any specific way. In order let the system reach a steady state before taking the results, we ran each test for atleast 25 times. The following sections describe these tests in greater detail.

3.3 Testing using dataset from CRAWDAD

We chose the topology traces collected during MANIAC Challenge 2009 and posted on [9]. According the website, the data comprises routing and topology traces collected during the Mobile Ad hoc Networks Interoperability And Cooperation (MANIAC) Challenge, held on March 8, 2009 in conjunction with IEEE PerCom 2009. The datasets used show the topology changes at each time instant. Two datasets were made available with 3 runs in each.

Each of the three runs of the competition lasted around 20 minutes. A total of 21 network nodes participated in the tests with IP addresses of the form 10.10.0.x, where x (the fourth octet) is in the set 21, 22, 23, 24, 25, 50-65. By comparing the image capture each second, we could extract the topology changes between one second and another and generated our input test data from this dataset accordingly. This formatted input was then passed to our our program that consumes these topology updates.

3.3.1 Test Methodology using CRAWDAD dataset

The topology datasets from [9] was effectively an image in the form of a 2-dimensional matrix of all possible links at the instant it was captured. The very first set provided a list of nodes as well as the initial state of the network. We therefore converted that data in the very first snapshot received to a series of Node additions and link additions.

The test data also provided multiple snapshots. In fact, over 1200 in each case at supposedly intervals of 1 second. Therefore, it is imperative that we process all these changes before the next snapshot arrives. Each snapshot contains a number of add links. However, the test data did show superfluous add links where a link to be added was already existing and therefore nothing needed to be done. On the same lines, the test data showed a number of remove links pointing to links that never existed.

We chose to measure the time taken for the first snapshot to be processed separately. This indicates the very first state where it takes in a baseline of the nodes and links. The work accomplished in this phase is:

- Create the Node and NIC card instance
- Add a link whenever there is a 1-hop link as provided by the snapshot between two nodes.
- Synchronize the data in the MySQL database and the in-memory cloud

In the next part of the run, we processed each snapshot received as a topology change from the snapshot received in the previous update. This allowed us to identify any link that existed before as a 1-hop link is now a non-existent link or a multi-hop link. This translated to a remove link message to be processed by the topology update. Similarly, a multi-hop or non-existent link changing to a 1-hop link was processed as an add link message.

We measured the time taken to process each update snapshot and averaged it over all the snapshots processed to give us an average processing time. The expectation was that these snapshots need to be consumed in the shortest possible time.

Finally, at the end of each snapshot, we measured the time it takes the system to complete a query between any two random nodes in the network. The program generates a pair of nodes from the set of nodes loaded into the in-memory graph. Then compute the All-pairs shortest path for the graph. Then compute the shortest path if one exists between the two nodes. Our expectation was that this query must return a response as quickly as possible.

3.3.2 Test Results from CRAWDAD dataset

The results of testing with the CRAWDAD dataset [9] is presented in the Table 3.1.

Table 3.1: Performance of MINA using CRAWDAD's vt/maniac topology traces

Dataset	2007-top1	2007-top2	2007-top3	2009-top1	2009-top2	2009-top3
Initial State Capture						
Nodes Added	21	21	21	16	16	16
Links Added	101	84	62	72	78	70
Time to Capture(ms)	751.36	532.95	275.25	381.04	253.04	316.71
Topology Update Capture						
Number of Topology Updates	1320	1260	1260	1219	1215	1243
Total Links Added	3475	3032	3382	3119	4143	3545
Total Links Removed	3390	2897	3231	3021	4035	3454
Average Time for Topology Update(ms)	13.86	12.01	12.67	11.63	14.87	12.66
Max Time for Random Query(ms)	4.96	4.57	6.84	4.69	4.64	4.576

3.3.3 Interpretation of the results from CRAWDAD datasets

The performance of the Data Access Layer with the CRAWDAD dataset was quite encouraging. We are aware that the size of the topology was not large enough for us to feel comfortable about the robustness of the platform. However, the number of topology updates that needed to be processed in a given snapshot seemed realistic enough for us to test with this dataset and document the findings. In addition, it gave us the opportunity to test whether our logic accurately identified the links to remove them as appropriate for the test to simulate such conditions.

Some significant take aways that we took was that the Data Access Layer was indeed capable of handling topology updates atleast for such a small topology. The other significant finding was that initial setup was likely to be slow when the program instantiates the in-memory graph from the persistent store. We then thought that this is an item that we should investigate with larger networks.

3.4 Testing using Random-generated Test Data

Based on preliminary results of testing the current MINA system, one of the key observations was a slowness in the Observe phase and Analysis phase. The reason was narrowed down to the data access layer design that made repeated calls to the database for any changes to existing topology. We realize that this repeated call would not normally have made a very big difference but the issue was that the Hibernate layer introduced a number of wasteful calls to the database. As a result, no SQL optimization was possible in the existing design.

With the revised database schema and a Data Access Layer that is built using the DAO Pattern, we now have an opportunity to validate the performance of the code with large volumes of input. The key aspects we need to cover are:

1. Time taken for a network topology in terms of nodes and links take to initialize from a persistent store

2. Given a certain volume of topology changes in terms of number of links added or dropped take to be processed, verify if increasing the size of the network has any impact.

3. Given a certain size of a network to start with, observe how increasing the volume of topology changes affects the time taken to process them. This is very significant as this will tell us how many such changes can the MINA system handle for a given size of the network.

3.4.1 Test Methodology using Random-generated Data

In order to generate this test data, we pre-populated our database with a given number of Nodes and Links. This allowed us to test for the time taken to instantiate a given network. Over repeated runs, we tested with different sizes of the network and measured the time taken to read these configurations from the database and instantiate a graph with equivalent nodes and edges. In the process of creating links between any two nodes on this test network, we did not choose to implement any specific topology here. Our reasoning was that this is a stress test and therefore as long as we tested with a large enough data size, these random links should not matter. Therefore we picked any two random distinct nodes and created a link between them subject to a limit of the number of links we needed approximately to set up the starting configuration. For example, we populated a starting point with 250 nodes + 1000 links, then with 500 nodes + 2000 links, ... 2000 nodes + 10,000 links.

In the next set of tests, we created batches of topology changes that we can expect to get periodically from the underlying network. We tested a small fixed amount of 100 links added between two nodes in the network at random + 50 links between any two nodes picked at random to be removed if a link existed. This data mimics the information that a Tier 2 node in MINA would get indicating a group of its child nodes that may have moved from one location to another and as a result, present link adds and deletes accordingly. Due to the choice of picking any two nodes in random, sometimes, we may generate data such that a topology message is received indicating a link being added but in reality that message was already received through some other source and therefore a number of such messages are superfluous. On the same lines, we have had instances where the test data 'removed' a link between two nodes but there never was one in the first place. Again, such messages were ignored.

We generated two sets of test data. First, by fixing the load of topology updates, measured the change in performance if any with different network sizes. Second, by fixing the starting configuration of the network, tested with varying amount of topology updates. The following subsections summarize our results and findings.

3.4.2 Test Results from Random-generated Data

The following paragraphs present the results observed using the random-generated test data. As mentioned before, we were more particular about testing with volume rather than the semantics of the network itself. As a result, we tested with loads that may not have any relation to any network we may see in reality. However, at the same time, we tested the Database Access Layer in normal realistic ranges of topology size and thus ascertained that the code does perform well under all types of stress conditions.

Studying the Time to Initialize MINA from persistent storage Info

The first observation we made was to measure the time taken to read data from the MySQL database and create an in-memory startup is proportional to the number of nodes in the persistent store. The significance of this measurement is that we expect MINA to have atleast all the Tier 2 nodes persistent in the repository. Tier 1 and Tier 2 nodes are relatively stable and do not enter or leave the network unless there is an outage. Therefore, if for example, the MINA system was ever shutdown the previous night, the next day, one could simply instantiate the process to read from this persistent store. However, this would mean a slightly longer startup time theoretically and we attempted to quantify this or put a ballpark on this phase. The variation of startup time with different node sizes is presented in Fig. 3.1. We see that even a large graph like 2000 nodes and 8000 links, the initialization process should take under a second.

Studying the affect of Node Network Size for Topology Updates

Our next task was to observe the effect of changing node network size and observing the time it takes for a fixed amount of topology update load. We tested this with 3 different sets of loads. 100 Link Adds, 1000 Link Adds and 2000 Link Adds. The plot shown in Fig. 3.2 shows the performance of the logic to absorb these topology updates. For a given load, there is little if not any variation of the processing time across all the network sizes. There is definitely an increase in processing time with increase in load which is acceptable and that is studied in the next subsection.

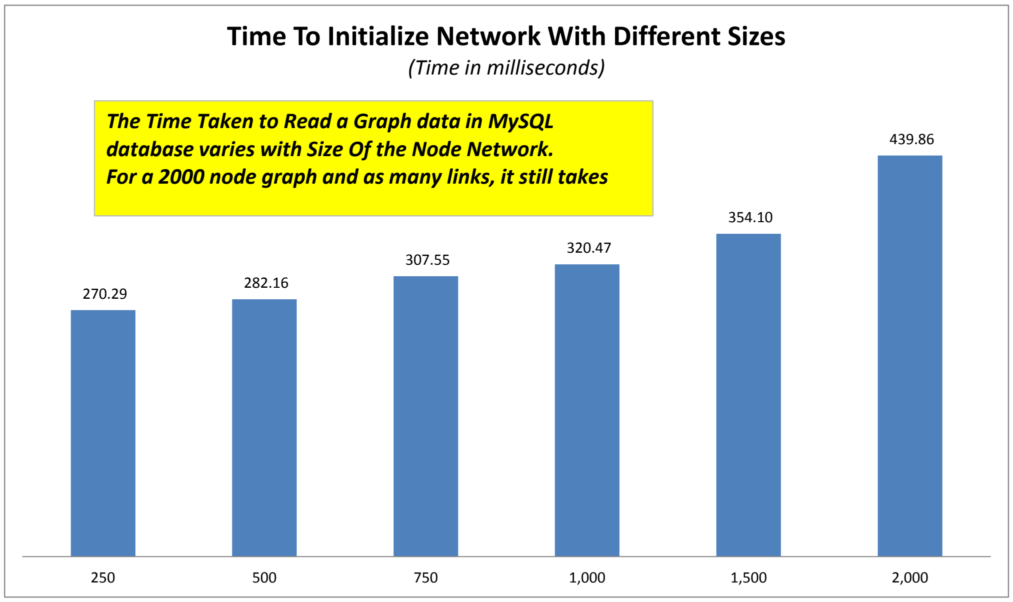


Figure 3.1: Time to Initialize Network Of Different Sizes

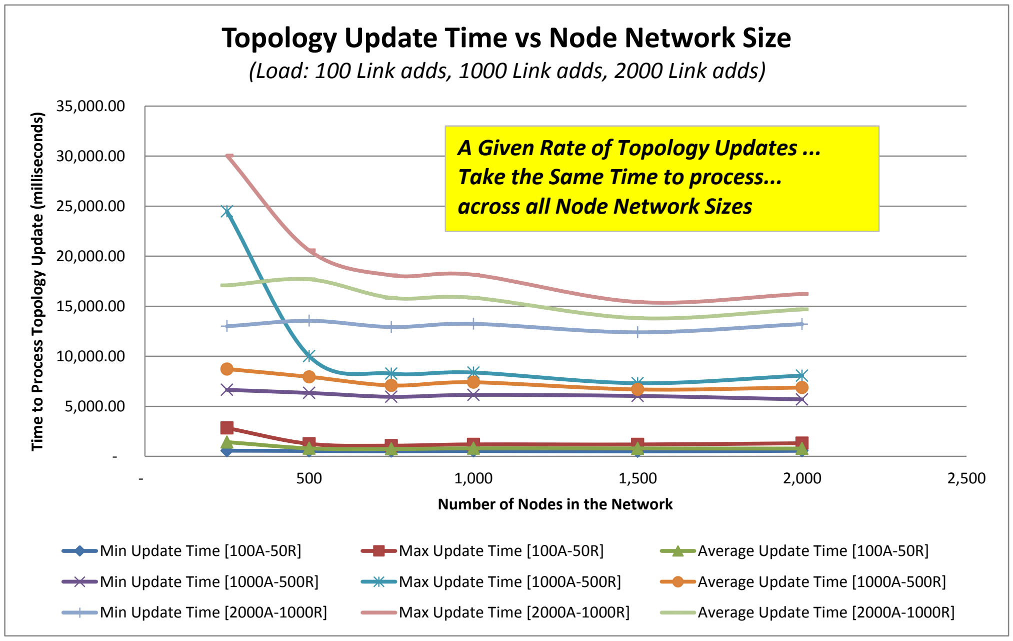


Figure 3.2: Topology Update Time vs. Node Size (Fixed Update Load)

Studying the Time Taken for Topology Updates with varying Load

We plot this metric in the Fig. 3.3. In order to test this aspect, we started each run with a fairly large initial state of the network. This is one of the aspects that we wanted MINA to support, i.e. ability to manage a large network represented in the in-memory graph object as well as in the persistent store. We picked 2000 nodes with 8000 links between nodes as the starting point which we considered as fairly large for a MINA deployment. Our observation was that the time to process the update was linearly increasing with increase in load.

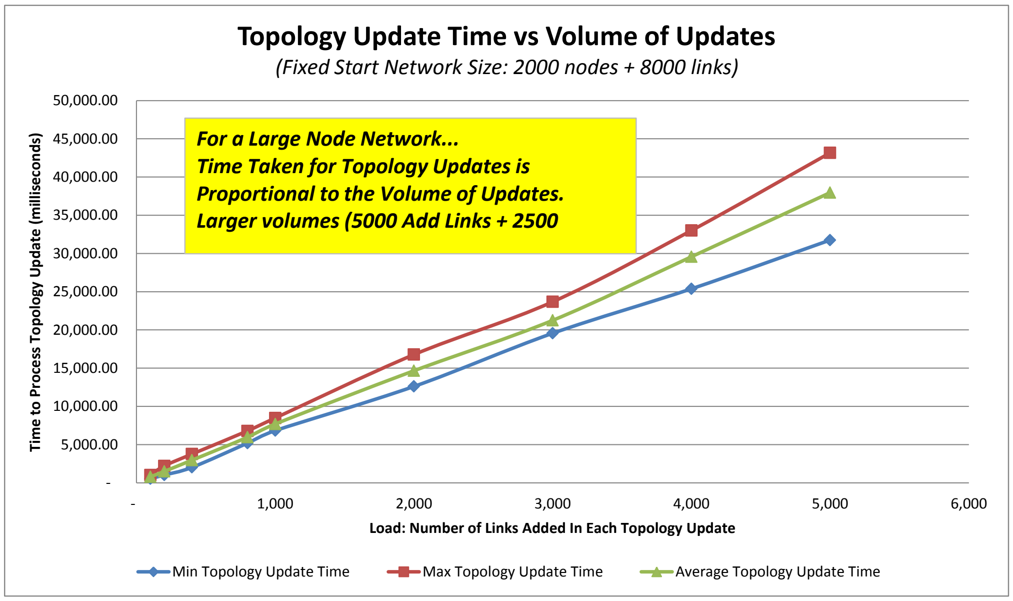


Figure 3.3: Topology Update Time vs. Volume of Updates (Fixed Start Network Size)

Studying the Time Taken for a Random Shortest Path Query with varying Load

Determining the shortest path between any two nodes in a given graph network is perhaps the most common query that MINA's Analyze function puts on the underlying Data Access Layer. In order to efficiently return a response, the system should be able to return the query results ion short order of time. Our observation was that for a reasonably large node network starting point, the query process time does seem to rise sharply after crossing say 3000 to 4000 link changes in a batch. See Fig. 3.4.

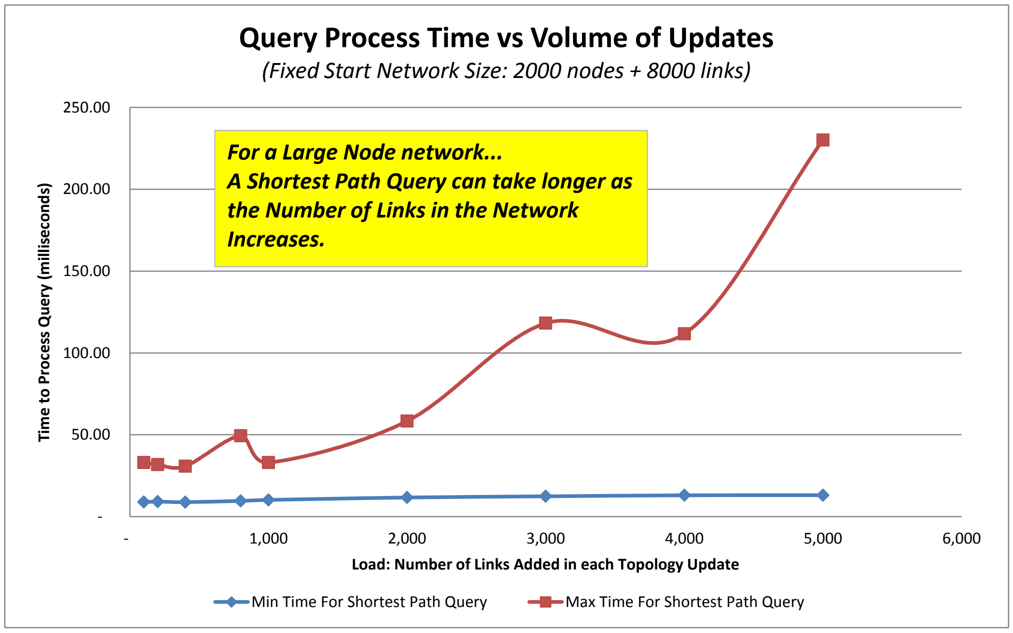


Figure 3.4: Random Query Process Time vs. Volume of Updates (Fixed Start Network Size)

3.4.3 Interpretation of the results from Random-generated Data

Node Network Size vs. Initialization Time

On closer inspection of the Time to Initialize Network Of Different Sizes in Fig. 3.1, there are two significant activities here.

- Loop through all the nodes and links from the repository
- For each value returned in the previous step, create a node and attach it to the graph or create a link and attach it to the graph

The complexity of this process is $\mathcal{O}(N+E)$ where N is the number of nodes in the persistent store and E is the number of edges/links in the persistent store.

Node Network Size vs. Topology Updates

Observing the plots for the Topology Update Time vs. Node Network Size, as mentioned before we were not very sure how the logic would behave. This was certainly a surprise that there was no significant change in the time for the topology updates with changing network size. Our interpretation of this phenomenon is that the JUNG framework is optimized to provide a constant-time lookup and constant-time insert/delete/update of the links in spite of the node network. This is clearly evident in Fig. 3.2 towards the larger network sizes.

One very interesting anomaly was consistently observed and did throw a doubt on our data collected. We noticed that when the node network size was just 250, the maximum time taken for the topology update was consistently much higher than the rest of the node network sizes. We ran these tests repeatedly at this node network size and this strange behavior is attributed to MySQL being slow in the first 30 seconds of its run. There are a number of references in the internet on the relative sluggishness of MySQL database for the first 30 seconds. A discussion regarding this is out of scope of this thesis. However, there is one aspect that must be considered. MySQL uses buffered query results. Initial queries are likely to be slow on the database. Under smaller nodes, the setup time for the network was too short and therefore the early measurements for the topology update must have got skewed by this.

Topology Update Time vs. Number Of Updates

For a reasonably large node network starting point, the topology updates take longer and rises linearly. This is confirmed from Fig. 3.3. We did not expect this to be any different. Our only purpose was to make sure that the increase was linear and not anything else. This can be explained again by the token that the JUNG library does provide constant-time access to a insert/update/delete operations on the graph network and so does the database in this case.

Query Process Time vs. Number Of Updates

We refer to the Fig. 3.4. The Shortest Path query in JUNG is supposedly implemented using Dijkstra's algorithm which has a complexity of $\mathcal{O}(E + log(N)$ where N is the number of nodes in the graph object and E is the number of edges/links. However, if there is sufficient memory on the server running the software, the time to return a shortest path query may not be significant. On a reasonably large graph starting with 2000 nodes and 8000 links, we still seemed to finish the shortest path query is less than 500 seconds. This shows that the strategy of using an in-memory graph can support pretty large networks and help MINA with this extension.

Chapter 4

EXTENDING MINA FOR NODE MOBILITY

In this chapter, we show a unique application of MINA in the real-world where it will play a major role in network management. We consider the scenario of a shanty town's emergency response system.

4.1 The Scenario of a Shanty Town

Shanty towns are a result of rapid urbanization in less economically developed countries [4]. They are characterized by informal construction with no building codes [15] and are often illegal and self-built using basic materials. They tend to be totally unplanned and rapidly growing. As a result, such settlements have high population density and lack basic infrastructure services like sanitation, electricity, water and public transport. The population density in such places can touch a million per sq. mile. The size of the vehicles that can navigate the streets in the town can be limited by the width of the roads. Limitations may include narrow alleyways, overhead overhanging obstructions or illegally connected overhead electric wires. The life and challenges in a typical shanty town can be found in [3].

Shanty towns lack a reliable system for an emergency incident response and relief dispatch. The lack of uniform road access across the region makes the job of attending to emergencies or providing disaster relief a major challenge. Such a response in shanty towns largely depends upon (1) the nature of the emergency, (2) the road access to the location that needs the relief, and (3) the type of relief resources available on hand. Early gathering of information on the ground conditions plays an important role in the effectiveness of the response that would follow. A quick-response system using the right tools and coordination with volunteers, municipalities, locals and emergency relief groups will create a better opportunity to provide relief to the inhabitants. Therefore, we propose an approach of deploying volunteers as mobile agents, directed by a central Control Center (CC), to travel along a specified path on the road network, capture the emergency information quickly and transmit the data to the central location to provide situational awareness for appropriate response.

4.1.1 Related Work - Data Gathering for Shanty Towns

Due to a lack of any planned infrastructure, social scientists opine that NGOs and local service organizations are usually the first point of contact for the residents of a shanty town seeking relief from unplanned urban emergencies like fire, smoke, gas, medical etc. They register the request in their system and coordinate with local agencies who provide relief to the requestor.

In our survey of past work, [14, 24] start with an extensive use of sensors to capture such data at the scene. However, the lack of good data infrastructure leads to the data staying uncollected and rendered useless within a short period of time. The data remains on the sensors until some opportunistic mobile nodes come within range. The data exchange takes place from the sensors to the mobile nodes. [12] describes a number of these commonly used data exchange strategies in this problem space.

Providing a reliable response to emergency requests in a shanty town requires the emergency event data to be captured within a finite amount of time. We use the term *event expiration* to indicate such a time window. Many commentators on shanty town such as in [21] have concluded that residents in shanty towns have realized the potential of being active, organized citizens in their neighborhoods and for this reason we consider a volunteer-driven approach to operate the Mobile Data Collectors.

4.2 Topology for Shanty Town with MINA

An effective system can be built using different classes of nodes with differing capabilities and on-board resources. The entire region could be instrumented with a large farm of low-cost sensor nodes that can capture data on demand on various parameters. We can incorporate MDCs and fixed networking equipment as necessary, all working with a remote MINA server at a Control Center(CC) and having the ability to analyze any received data.

Emergency situations called *events* can be registered anywhere in the shanty town and the nearby sensors would capture the vital details like air quality, smoke, temperature etc. However, the sensors need to wait for an available MDC to pick up and send as quickly as possible to the CC. The QoS requirement of such a topology necessitates providing adequate coverage to capture data from the sensors locally and have sufficient number of MDCs available to transport the data to the server for further analysis within the shortest possible time. In such an emergency response system, the sooner this first-response detail is sent the faster an appropriate relief can be dispatched by the CC. [23] started the initial thinking on minimizing this delay by considering a suitable mobility model for the MDCs. We extend this idea by formally tackling the task of assigning a mobility model to the MDCs.

A shanty town emergency response could consist of training a team of local volunteers who know the terrain well enough to act as Mobile Data Collectors (MDCs), gathering information from event locations on-demand. A practical observation in a shanty town is that relief depends upon the access to the event location. With development so unplanned in a shanty town, some locations are accessible by a four-wheeler such as a car or a taxi while others can only be navigated by a two-wheeler like a bicycle or scooter. Some others can only be reached on foot. Therefore, we need heterogeneous MDCs with capabilities that can navigate the different types of access paths. The volunteer MDC operators can be distributed anywhere in the shanty town. Over the course of time, additional volunteers can be signed up in specific hotspots in the shanty town to improve their availability.

Operating and coordinating the MDCs and the greater emergency response effort presents several challenges. First, the MDC and the CC need to have a common understanding of the road network. Modeling such a road network and representing the roads with the right characteristics can only be as accurate as the best available data source (e.g. OpenStreetMap). Temporary blockages due to construction, fire spread, or other reasons may not be discovered until an MDC attempts to cross them and so solutions must have the ability to quickly adapt to such dynamic events.

In a shanty town emergency system, it is critical to ensure that data captured at an event source be communicated to the sink (the CC) with least amount of delay. However, we cannot assume having a guaranteed network connectivity in such places to make this happen. On this topic, past work [14] suggests incorporating static nodes in the proximity of the mobile nodes to relay the data to the sink. We understand that cellular network coverage is ubiquitous even in shanty towns today. However, the construction material used in shanty towns can cause a lot of *dead zones* within the shanty towns for cellular network coverage also thereby making it unreliable for transporting critical data.

We therefore suggest using a set of fixed nodes operating as WiFi Access Points or Routers so that an MDC would be in the range of more than one AP at any point in time. When the MDCs encounter *dead zones*, we have to depend upon the MDC to use some form of store-and-forward or DTN technique and minimize the time to send to the CC. Our suggested network architecture is presented in Fig. 4.1.

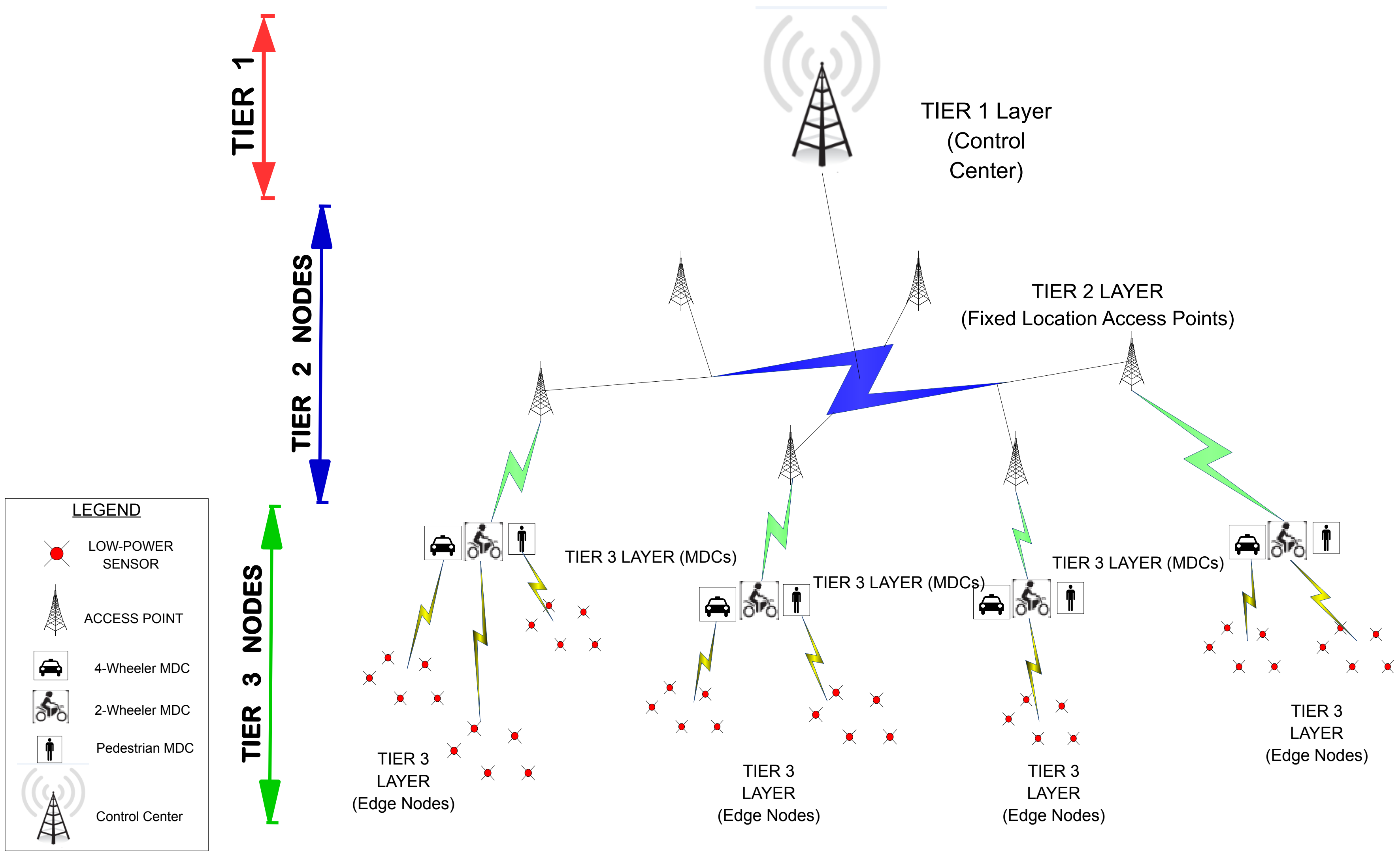


Figure 4.1: Shanty Town Network Topology with MINA

The complexity of such a network requires some form of network management. We propose using MINA. MINA (described in detail in Chapter 2) classifies the nodes in the network into 3 different tiers. Sensor nodes are classified as Tier 3 - Edge nodes. The mobile nodes i.e. MDCs are classified as Tier 3 - Mobile nodes. The Access Points are classified as Tier 2 nodes and finally CC is the Tier 1 node or MINA server. We recommend having redundant connections between nodes in each layer so as to minimize the chances of ever losing connectivity between the CC and the MDCs.

4.3 Need for Node Mobility

The topology shown in Fig. 4.1 is tailor-made for MINA and there should be no concern on its applicability. However, MINA as it stands today has no part to play in the mobility of the mobile nodes in the topology. When nodes move opportunistically, they do cause new links to sensor nodes and drop others along the way and MINA knows about this movement only after the fact.

The primary objective of our enhancement to MINA is to intelligently route the MDCs in the shanty town given that we have some knowledge of where the mobile node should be headed. As the nodes move, MINA captures the state changes appropriately using its OAA loop. As the mobile nodes move, MINA allows a location update of these nodes and now we have a real-time picture of the network.

In order to determine the mobility model, we present here our modeling of the road network. The following paragraphs and subsections discusses the modeling, the problem statement and the mobility algorithms in detail.

4.3.1 Shanty Town Road Network as a Graph

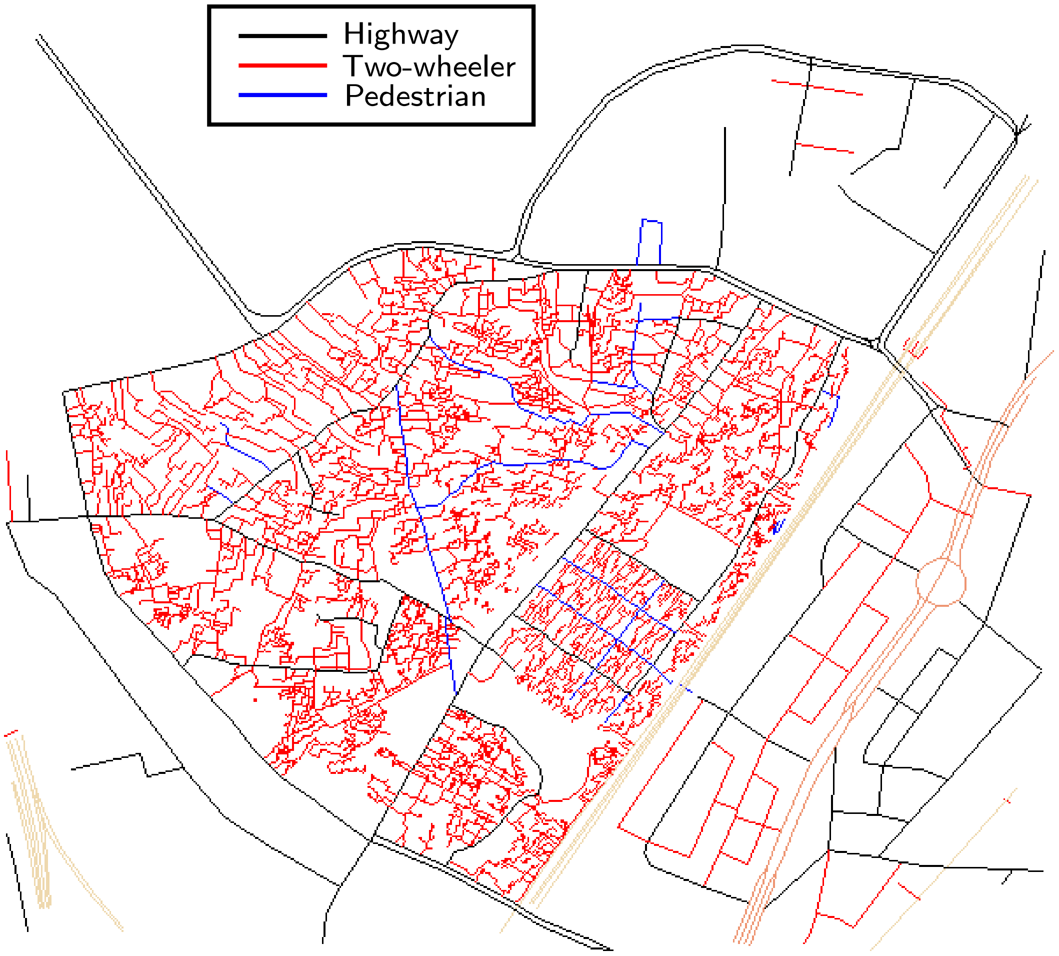
In order to plan a path for each MDC, we need an accurate representation of the shanty town and the different artifacts used to formally describe the access paths. We model the entire space as a graph with junctions and road-ends modeled as vertices on the graph and the accesses linking these junctions to one another as edges. For example, *openstreetmap.org* [1] provides a fairly detailed XML representation of any mapped region in the world in an OpenStreetMap (OSM) [1] format. There may be other sources of this information. The edge weights should be proportional to the length of the access link. We therefore have a graph $G(V, E)$ as a generic representation of the road network.

The shanty town can have roads of different types. For the sake of a generic model, we consider three different road networks using the set of edges: *(i) H-Type* with edges where a four-wheeler can reach, *(ii) T-Type* with edges where a two-wheeler (e.g. scooter or moped) can access, and *(iii) P-Type* with edges where pedestrians can walk. A pedestrian-mounted MDC can traverse an edge of any type while the two-wheeler MDC can traverse just the T-Type and H-Type edges. Thus, we have three different graphs each having the same set of vertices for convenience but different sets of edges, such as: $G_H(V, E_H)$, $G_T(V, E_T)$ and $G_P(V, E_P)$. We can see then that:

$$E_H \in E_T \in E_P$$

Fig. 4.2 shows the picture of the graph network.

Figure 4.2: Graph Network for Dharavi



4.3.2 Emergency Request as an Event

A typical emergency request would be a message phoned in to the Control Center (CC) or some other external communication channel that the CC provides. We model a request for emergency assistance received by the CC as an event. An event could apply to any location in the shanty town. For simplicity, in order to pinpoint the location for emergency response, we translate the event to the nearest vertex location. Thus an *event* is represented as $e(v_e, ts_e, exp_e)$ and is described in terms of the following attributes:

- v_e The nearest junction on the road network, which is considered as the event location

- ts_e The timestamp when the event has occurred
- exp_e The expiration time of the event

The *Expiration Time* of the event determines a window within which the event data needs to be captured. Any event data captured after this duration is of limited value and hence considered useless for us. Expiration Time for individual events could be different depending upon the nature of the event. However, our model can flexibly accommodate this as needed.

We assume that the events are discrete and non-deterministic. In our analysis, we assume, say M events during an observation window. Therefore we model the distribution of the *Event Arrival Time* for the events as a uniformly random thereby generating the requisite number of events.

4.3.3 Modeling the Mobility of Heterogeneous MDCs

In our model, the Control Center (CC) provides the sequence of event locations and the fastest traversal path based on global knowledge of the road network, its condition, and of other events occurring in the shanty town.

An MDC Walk: We define the sequence of event locations visited by (or planned for) an MDC as an MDC Walk or more succinctly as a *walk*. This term is used very liberally in the following discussion. Therefore, we write:

$$W_x = \{v_1, v_2, \ldots, v_n\}, \text{where } v_i \;\epsilon\; V,\ \forall i \in [1, n]$$

Representation of an MDC: We model an MDC x using the following attributes and represent it by $M_x(g_x, v_{x0}, W_x)$ where:

- g_x is the graph network that the MDC x can traverse which can be an instance of the three graphs $G_H(V, E_H)$, $G_T(V, E_T)$ and $G_P(V, E_P)$.
- v_{x0} is either the next vertex the MDC x is headed to visit in the current path or its last known location (e.g. a normal rest location of the MDC). This is significant because any changes to the trajectory will be specified only from this location ($v_{x0} \epsilon V$).
- W_x is a walk assigned to the MDC x starting from the reference location v_{x0}.

When the CC receives a new event, it computes an efficient walk to handle this new event as well as the previously scheduled set of events. The CC then updates the corresponding MDC with a new walk W_x.

Generic Cost Model Design: We consider two aspects here. (1) Travel related costs and (2) Data Capture Constraints. Travel-related costs are typically in terms of distance traveled or time elapsed during the travel or for completing a service. Data Capture Constraints are associated with missing the capture of an event as it represents an opportunity missed to serve a customer.

Travel Related Costs:

Travel Time: We write the Travel Time of an MDC x between two locations v_a and v_b as:

$$TT_x(v_a, v_b, t) = \frac{d_x(v_a, v_b)}{s_x(v_a, v_b, t)} + (h_x(v_a, v_b) * \alpha) + \beta \tag{4.1}$$

Here $d_x(v_a, v_b)$ is the shortest distance between v_a and v_b on the graph that the MDC x is associated with, $s_x(v_a, v_b, t)$ is the average speed along the edges from v_a to v_b that the MDC x travels starting at v_a at time t, t is the time at which the MDC departs v_a, $h_x(v_a, v_b)$ is the number of hops along the shortest distance on the graph of MDC x between the vertices v_a and v_b, α is a delay at each junction along the path that accounts for time negotiating an intersection, and β is the service time at an event location for gathering data. This generic representation allows us to model that the travel time can change depending upon the traffic conditions at t. In our analysis, we do not consider the traffic conditions.

Expected Time of Arrival: The Expected Time of Arrival of the MDC x from v_a to v_b is computed in terms of the ETA at the starting vertex $v_a$at time t as:

$$ETA_x(v_b) = ETA_x(v_a) + TT_x(v_a, v_b, t) \tag{4.2}$$

$v_a \in W_x$ and $v_b \in W_x$ at time t.

Total Distance: A useful measure of the cost incurred by operating an MDC is the Total Distance traveled during an entire response effort. If we have K MDCs operating in the neighborhood, we have

$$TD = \sum_{j=1}^{K} \sum_{i=1}^{m(j)-1} d(v_i, v_{i+1}) \tag{4.3}$$

Here *m(j)* is the number of events successfully visited by an MDC j and $v_i \in W_j$.

Data Capture Constraints:

Miss Penalty: We define the Miss Penalty of the event at location v_a as the penalty assessed when an MDC arrives at an event location after the event has expired OR when no MDC arrives at all within the observation window.

$$MP(v_a) = \begin{cases} 0 & \text{if } ETA_x(v_a) \leq EXP_x(v_a) \\ 1 & \text{if } ETA_x(v_a) > EXP_x(v_a) \end{cases} \tag{4.4}$$

$v_a \in W_x$ at the end of the observation window, $EXP_x(v_a)$ is the expiration time of the event at location v_a on the walk of MDC x. When the MP at the corresponding event location is evaluated as 1, we consider that the event has been missed.

Total Missed: A measure of the total number of the missed events gives an indication of how efficient the path plan is in terms of data capture. In all our discussions, we assume that an event location is visited by only a single MDC. Therefore, we compute the number of missed events as

$$TM = \sum_{j=1}^{M} MP(j) \tag{4.5}$$

Thus over a period of time, we account for a set of M events covered by all K MDCs as:

$$M = TM + \sum_{j=1}^{K} m(j) \tag{4.6}$$

Here $m(j)$ is the number of events successfully visited by an MDC j for all the vertices in the walk W_j.

Percentage of Event Data Captured: This measure is an indicator of how effective a scheme is. We compute it as:

$$\frac{(1 - TM)}{M} * 100 \tag{4.7}$$

Average Time of Event Data Capture: Another useful indicator to show how responsive a scheme is to measure the time elapsed between the event request and when an event data is captured averaged over a period of time across the set of M events and using K MDCs as:

$$\frac{\sum_{j=1}^{K} \sum_{v_i \in W_j} (ARR_j(v_i) - e_i)}{M - TM} \tag{4.8}$$

$ARR_j(v_i)$ is the arrival time of the MDC x at event location v_i such that $v_i \in V$. e_i is the time the event request was received for location v_i.

4.3.4 Problem Statement

The above formal definitions leads us to the following overall objective:

Assuming that K MDCs are available in a shanty town emergency response effort and M independent events occur random non-deterministically distributed in time and space in the shanty town, our ideal objective is to generate a path plan for these K MDCs to handle all these M random events without exceeding their event expiration times. Informally we try to primarily maximize the Percentage of Events Captured before expiration and secondarily minimize the Average Capture Time of an Event.

Stated more formally,

GIVEN:

An MDC x defined as $M_x(g_x, v_{x0}, W_x)$.

$$\begin{aligned} W_x &= \{v_1, v_2, \ldots, v_n\}\,, \text{where } v_i \in V \\ EXP_x &= \{e_1, e_2, \ldots, e_n\} \end{aligned}$$

EXP_x is the set of expiration times $EXP(v_i)$ of each of the event locations in W_x. There are K MDCs and M events to capture. $TT(v_a, v_b)$ is the travel time of an MDC on the shortest path between v_a and v_b and is measured in seconds. $ARR_x(v_{xn})$ is the arrival time of the MDC x at v_{xn}. Find an ideal set of k walks W^{Ideal} such that

$$W^{Ideal} = \{W_1, W_2, \ldots, \ldots, W_K\} \text{ and} \tag{4.9}$$

$$\begin{aligned} &\text{minimize} && (4.8) \\ &\text{subject to} && ARR_x(v_{xj}) + TT_x(v_j, v_{j+1}) \leq e_{j+1}, \qquad (4.10) \\ & && \forall j \in [0, n-1] \text{ and } \forall x \in [1, K] \end{aligned}$$

Here, v_{x0} is the current location of the MDC x and $v_{x0} = v_0$. Clearly, Equation 4.10 cannot always be satisfied as some events may necessarily be missed due to a lack of available MDCs. Therefore, we approximate this solution as best as possible with the heuristic algorithms described in Section 4.4.

Problem Reduction: This problem falls into the broad category of Vehicle Routing Problem with Time Window (VRPTW). Traditional Vehicle Routing Problems involve a deterministic set of customer locations and a set of vehicles with finite capacity to serve them. However, the MDC should accommodate new emergency requests as and when they come in. We thus have a Dynamic VRPTW problem. This is known to be a subclass of Combinatorial Optimization Problems, which are NP-hard, and requires searching among $O(m!)$ combinations. Like many such problems, heuristic-based solutions can be quite effective.

Multiple-TSP with Time Windows (mTSPTW) [5] and Vehicle Routing Problems [17, 19] have been studied extensively in the operations research community. In [13], Laporte describes the exact algorithms to solve relatively small problems. When the demand changes over time, i.e., handling non-deterministic events as we are, the classic solutions become even more expensive. Spliet, in [20], describes that one of the solutions is to come up with a master schedule and apply minimal changes to accommodate new demand. This might cause the resulting schedule to be sub-optimal but it may yield acceptable results, i.e., customers are served within their time window constraints but the overall distance covered by the service may not be optimal. Moreover, TSP or mTSP, by definition, are constrained to avoid visiting the same location more than once on a tour and they typically do not handle dynamic events. Similar work can be found in [8] and [22].

4.4 Path Re-Scheduling Algorithms

In the previous section we reduced our problem to a Dynamic Vehicle Rescheduling problem for MDCs in the shanty town. We therefore solve this problem using heuristics that we describe here. In reality, we may only generate a W^{eff} (a subset of all the vertices to be visited by the walks assigned to each of the k MDCs) and this could be only a subset of an W^{Ideal} (an ideal solution which is again a set of vertices for the k MDCs that will be hard to find), i.e.,

$$\{v_i \in W_k,\ \forall\ W_k \in W^{eff}\} \subseteq \{v_j \in W_x^{Ideal},\ \forall\ W_x^{Ideal} \in W^{Ideal}\} \tag{4.11}$$

The remaining event locations are discarded as the algorithm does not provide a route to the locations before the event data expires.

In this section we propose two different heuristic algorithms, the *Minimum Deviated Walk* and the *Ortho Walk* to handle such dynamic events and provide a schedule or walk for each MDC. In order to compare the performance of this, we first describe a naive approach to solve the problem, which we call the *Nearest Walk*.

4.4.1 Summary of assumptions for all heuristics

MDC Current Position: We assume that the Control Center keeps track of the position of each MDC and therefore knows the next routable vertex (road junction) on an MDC's path. This is the earliest vertex at which a route for the MDC can be changed, which we call the *CurrentPosition.*

MDC Current Time: The time of arrival of the MDC at the *CurrentPosition* is called the *CurrentTime.* For stationary MDCs this is simply the current time.

All-Pairs Shortest Path Lookup: We assume the road network is modeled as a graph G with v vertices and e edges. There is a one-time cost to obtain the TimeToTravel between all vertex pairs by the shortest path available. The Floyd-Warshall (or similar) algorithm can yield this in $\mathcal{O}(v^3)$. If we keep this information in persistent storage we can perform a constant time lookup.

4.4.2 Nearest Walk - A Naive Approach

The core principle here is to append the new event to the walk of one of the MDCs that will be closest to the event location at the end of its respective schedule. In other words, we measure the shortest distance between the new event and the last event location scheduled for an MDC. Then determine the MDC which is closest w.r.t. travel time via the road access. The new event location is appended to the walk of the corresponding MDC. We therefore call this algorithm the *Nearest Walk.*

Analysis of the Nearest Walk Algorithm

We now analyze the complexity of this algorithm for a single event that needs to be accommodated among the schedules of k MDCs. The algorithm identifies the final destination in the current walk (last stop) for each of the k MDCs using $\mathcal{O}(k)$ constant-time operations. Using the All-pairs Shortest Path Lookup we compute the ExpectedTimeOfArrival of the MDCs to service the new event from that last stop in the existing walk. For all k MDCs, this is $\mathcal{O}(k)$ constant-time lookups. We pick the MDC that could reach the new event location earliest, using at most k comparisons. Thus for each new event, we execute a complexity of just $\mathcal{O}(k)$ to augment the walk of one of the MDCs. The value of k is typically a very small number in comparison to the number of vertices or edges in the graph. The algorithm is polynomial in k for each iteration of an event and therefore expected to run fast.

4.4.3 Minimum Deviated Walk - A Local Search Approach

This is a local search algorithm where we attempt to find the best place to insert a new event location. The motivation of this algorithm is to leave the relative order of the events in the existing schedule intact. When we deal with multiple MDCs, we pick the MDC that pushes the very last event on its walk the least amount in time, giving it the name *Minimum Deviated Walk*.

In order to achieve this goal, we consider first the walk of each MDC that we may have from a previous iteration of the algorithm. Knowing the location of the new event allows us to compute the shortest TimeToTravel between any one of these vertices on an MDC's walk and this event location.

We then try to insert the new event location after this vertex. This deviation from the previous path effectively extends it by adding the segments from (1) the closest vertex along the path to the new event location and (2) the new location to the next event location that was in the original plan for that MDC. This results in the MDC reaching the subsequent event locations later than originally scheduled. Therefore, we must then check each of these pre-scheduled events on the MDC to determine if they will be missed (reached after event expiration). If the deviation causes any of the events to be missed, we do not include the new event. Instead, we defer consideration of the event until after the first event location that would be missed if including the new event. Such a deferred event will now have another opportunity to get scheduled at a later time.

Analysis of the Minimum Deviated Walk

This algorithm is executed at every event occurrence as it needs to insert the new event location in one of the MDC schedules. We assume that we have k MDCs and m unexpired events. For each MDC, the algorithm computes the minimum additional time required to take a detour from the MDC's current walk at each possible insertion point. If there are m unexpired events across all the MDCs, the algorithm tests $\mathcal{O}(m)$ insertion points in the MDCs' walks. If the new event is reachable at the insert location within the specified time, DW checks for the possibility of missing any of the later events that were in the original schedule, which also requires $\mathcal{O}(m)$ comparisons at most. Lastly, the DW must at least check each of the k MDCs, including the idle ones, for possible assignment of the new event. Therefore, this algorithm requires $\mathcal{O}(m+k)$ computations to find out the appropriate position for the new event. Note that there exists the possibility of deferring an event, which requires additional executions of DW at a later time. If we assume the expected number of additional executions is α, we have that DW's complexity for handling one new event is actually $\mathcal{O}((m+k)(1+\alpha))$.

4.4.4 Ortho Walk - A Greedy Approach

We now describe a solution that uses a greedy approach for determining an efficient walk. The Ortho Walk considers all pending events to be visited by the MDCs along with the new event. This is not biased in any way towards the pre-scheduled events and evaluates the best path based on the *CurrentPosition* of the MDCs and the set of event locations pending (i.e. to be visited by some MDC).

We keep an array of event locations to keep track of the remaining event locations still under consideration, called *UnexploredEvents*. We use the *CurrentPostion* and *CurrentTime* values for each MDC as defined earlier. We record the time of arrival at these current positions for each of the k MDCs and call them $CurrentTime_x$. We start our search among the k start locations and find the earliest event location among the *UnexploredEvents* that can be reached by any of the MDCs. As we narrow down each transition, we mark that vertex as *visited* and therefore remove it from the *UnmarkedEvents*.

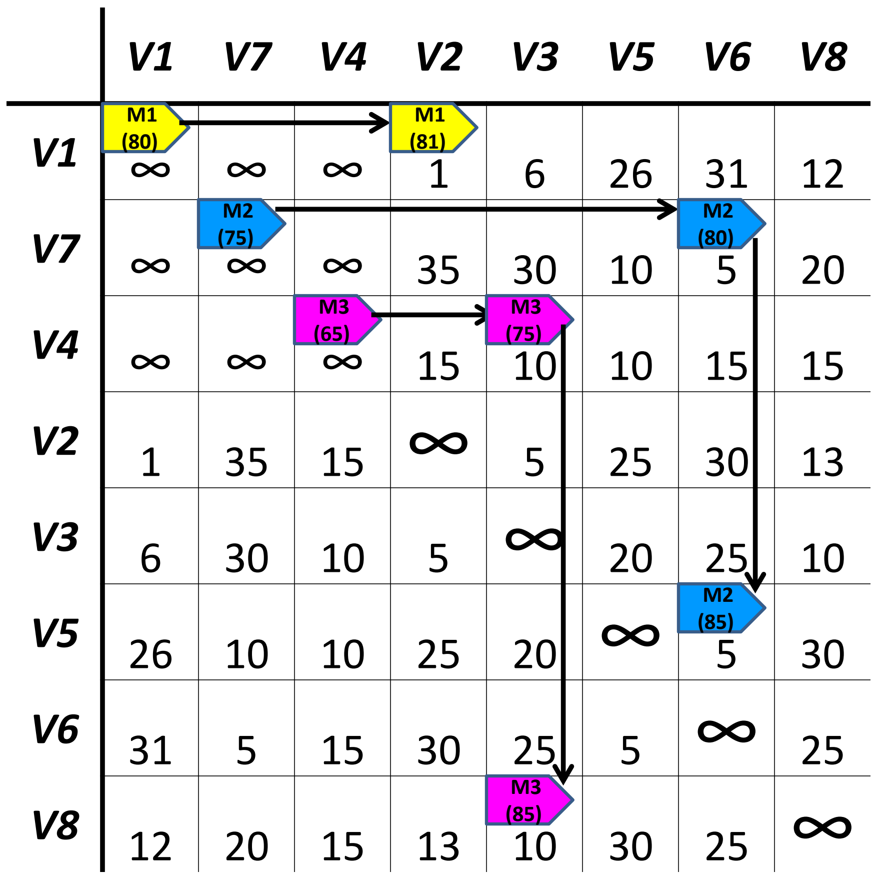
The algorithm ends with the iteration where (1) there are no elements in the *UnmarkedEvents* or (2) there was no movement possible between and MDC at its corresponding current position to an unmarked event location without expiring the event. Such events will typically get missed unless a new event gets added causing a re-shuffling of schedules thereby making room for this event into one of the schedules. The algorithm results in an effective path for each of the k MDCs.

It can be easily observed that this algorithm will keep the MDC close to the current location and inherently prefers the next event location within its current vicinity. Therefore, if there are fewer events in a remote part of the shanty town and there are no MDCs in the vicinity, such events may never get scheduled unless there is free capacity among the MDCs to be dispatched to the distant location.

Visualizing this Algorithm

We show a sample run generating the walks for 3 MDCs with 5 events to be visited. The *CurrentLocation* and *CurrentTime* of each MDC is indicated.

We build a *TravelTime* matrix with rows and columns representing the locations of *UnmarkedEvents* and the k MDCs. The cells represent the MDC's travel time to these locations along the shortest path. We mark the distance from each vertex to itself as ∞ as well as the MDC start locations as we do not consider them possible destinations. For the sake of simplifying the discussion, we assume that all MDCs are of the same type. Consider this matrix as a grid where each MDC is placed on the cell corresponding to its *CurrentLocation*.



	V1	V7	V4	V2	V3	V5	V6	V8
V1	∞	∞	∞	1	6	26	31	12
V7	∞	∞	∞	35	30	10	5	20
V4	∞	∞	∞	15	10	10	15	15
V2	1	35	15	∞	5	25	30	13
V3	6	30	10	5	∞	20	25	10
V5	26	10	10	25	20	∞	5	30
V6	31	5	15	30	25	5	∞	25
V8	12	20	15	13	10	30	25	∞

Figure 4.3: Ortho Walk Run (For a sample input of 5 events at V2, V3, V5, V6, V8 and 3 MDCs at V1, V7 and V4)

For MDC x, find the earliest ETA_x corresponding to the unmarked locations in the current row. Repeat this for each of the k MDCs and choose the earliest ETA_x among all of them that does not expire an event. Mark this location as visited and update MDC_x's *CurrentLocation* and *CurrentTime*. In the next iteration, MDC_x's reference location has changed and it considers events along its current column instead, alternating between rows and columns for each event visited. Fig. 4.3 shows the schedule returned by the algorithm. The path traced by each MDC on the matrix changes in right angles, which was our inspiration for naming the algorithm *Ortho Walk*. This process continues until the algorithm terminates.

Analysis of Ortho Walk Algorithm

The OW heuristic function executes every time the system registers a new event. Let us assume there are m pending events yet to be serviced by k MDCs. In the first iteration, the algorithm determines the shortest TimeToTravel between the starting points of each of the k MDCs and the m event locations, which takes mk constant time lookups. Choosing the lowest TimeToTravel would add an additional k comparisons. Therefore, the running time for one iteration, which obtains a schedule for one of the m event locations, is $\mathcal{O}(mk)$. That leaves us $m-1$ event locations to schedule in the next iteration. Therefore, for all m event locations, we evaluate the complexity as $\mathcal{O}(mk^2)$. Note that as both m and k are much smaller in comparison to v or e we have reduced the complexity to a manageable degree.

4.5 Extending MINA System Architecture

In the previous subsections here in this chapter, we discussed in detail, our work on providing the ability to direct the mobile nodes in a shanty town to maximize the data capture for emergency response. We will now place this in the context of MINA architecture in system terms. To do this we first discuss the system architecture we envisioned for the shanty town emergency response. We then show the link between the two and see how the two enhance the value of one another's capabilities.

4.5.1 System Architecture for an Emergency Response System in a Shanty Town

Our conceptual view of an emergency response system for shanty towns follows a distributed system middleware deployment paradigm. Fig. 4.4 presents an overview of the system

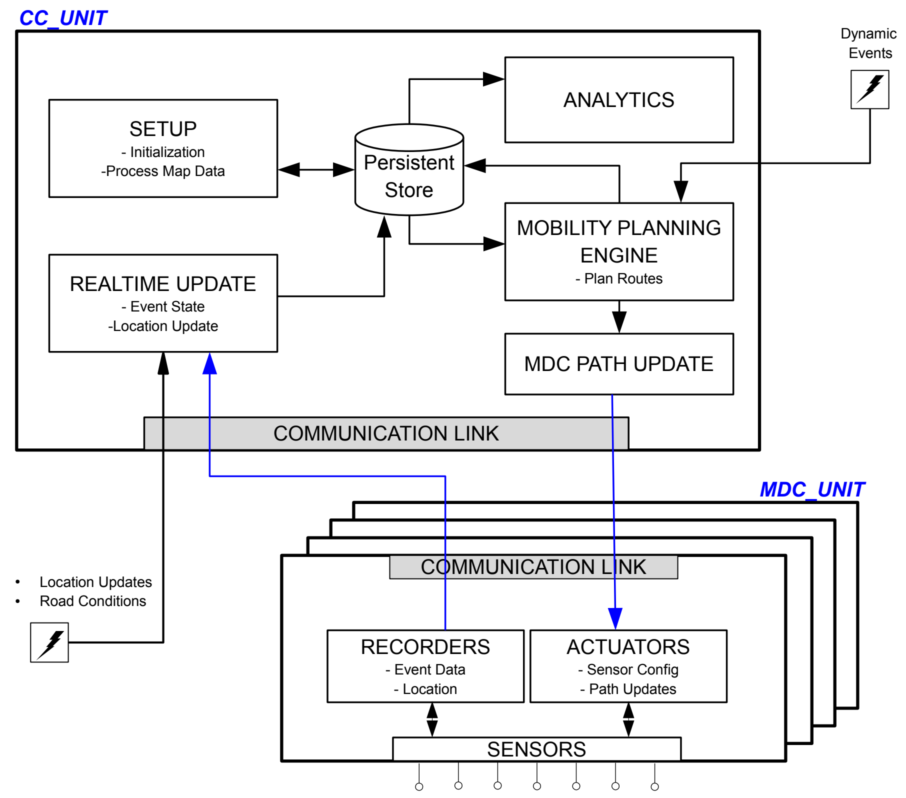


Figure 4.4: System Architecture Overview of Shanty Town Emergency Response

architecture of such a deployment. We see two middleware components: *Control Center Component* (CC_Unit) and an *MDC Component* (MDC_Unit). The CC_Unit does the bulk of the computation and this allows the MDC_Unit to be lightweight and consume a minimal amount of CPU or energy resources. The MDC_Unit is replicated on each MDC and communicates with the CC_Unit via a communication link, e.g. satellite network, WiFi or any other capability that ensures a connectivity to the CC. We can alternatively use a store-and-forward or Delay Tolerant Network (DTN) based technique to opportunistically send the collected data to the CC using any other available connectivity. We analyzed this aspect in a related work [11].

These components are described in greater detail below.

Control Center Component

The Control Center Component (CC_Unit) has 5 main modules. The Setup Module, the Mobility Planning Engine Module, the Analytics Module, the MDC Path Update Module and the Realtime Update Module. Fig. 4.4 depicts the interactions between these modules.

We assume that an input data of some kind is available that depicts the roads and junctions, their characteristics such as distance between junctions and the access type to indicate whether they are navigable by a car, scooter or a pedestrian. We showed this in Fig. 4.2. The Setup Module performs a one-time task of formatting the input data into a graph. When the road conditions change, some access paths may become unusable and therefore some portions of the graph need to be updated. This may trigger additional computations of shortest distance between vertices if that is being saved locally. The Setup Module is also responsible for storing the map data into some form of persistent storage. This module processes the data to produce as many graphs as there are road types with some access paths being present in more than one graph. Depending upon the type of MDC, we assign the appropriate graph network for further computations.

An important part of the CC_Unit is the Analytics Module which analyzes and presents the data collected. The Mobility Planning Engine (MPE) records the events received from an external source which triggers the rescheduling of the MDCs to accommodate the new event. MPE is the heart of the CC_Unit. The MPE module determines among the heterogeneous MDCs, the one that can reach the event location soonest. It forwards an outbound message to the MDCs with an updated route through the MDC Path Update Module. The Realtime Update Module receives the event data collected by the MDCs and save it in the persistent store. In addition, it handles any road change conditions sent from an external source triggering an update of the graph data from the Setup Module.

The MDC Component

The MDC Component (MDC_Unit) consists of two main modules. The Recorder Module captures the event data from the sensors and sends it to the CC_Unit via a communication link. The Actuators Module handles two diverse tasks. One is to update the path plan provided by the CC_Unit and the other is to control the sensors if need be. The MDC_Unit uses the communication link to communicate with the CC. The capability of a periodic update allows the CC to make a determination if the MDC is running behind schedule and therefore reroute other MDCs to event locations that may in the danger of not being serviced before event expiry.

4.6 Putting MINA and Mobility Planning Together

The topology for a shanty town shown in Fig. 4.1 introduces a layer of Access Points and Routers inside the shanty town area. As a result, the MDCs have more opportunities to stay connected and communicate with the Control Center. Messages can be routed to/from the Control Center more promptly. The introduction of MINA to manage such a topology provides very unique capabilities on this infrastructure.

We present two such use case scenarios here where MINA makes this a robust solution for efficient response to emergency/disaster scenarios.

Figure 4.5: Use Case 1 - Handling Shanty Town Emergency/Disaster Response

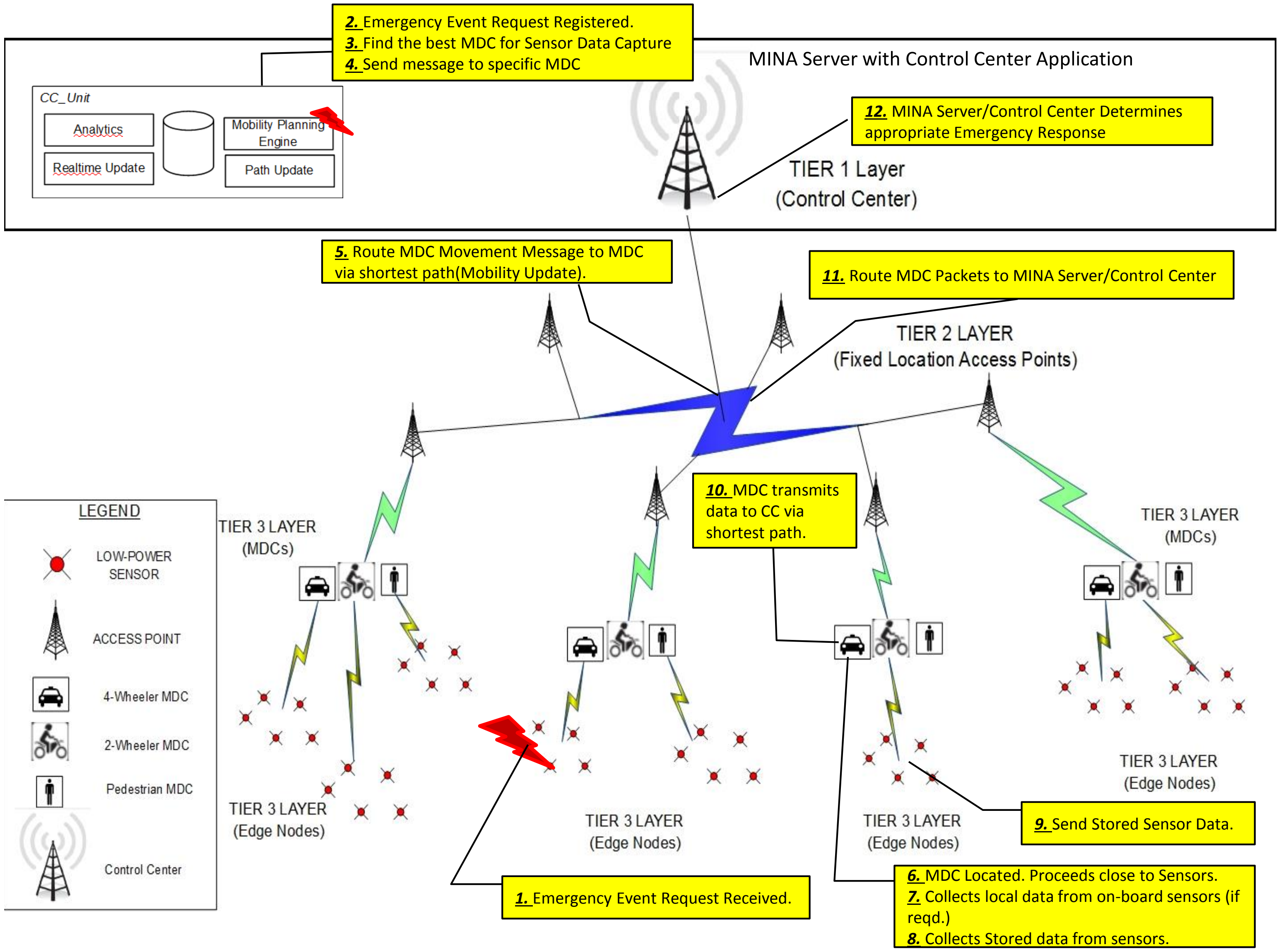
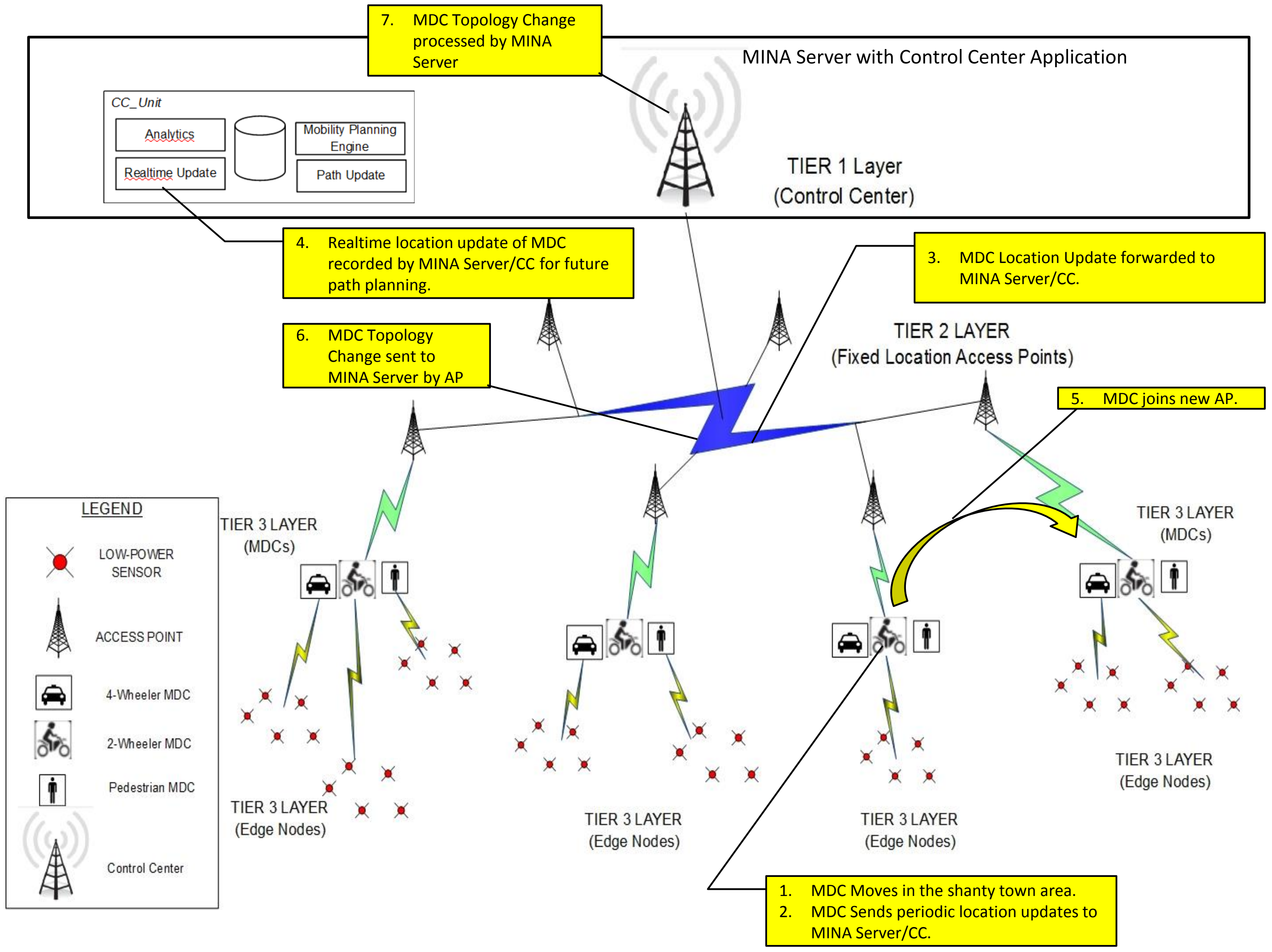

2. Emergency Event Request Registered.
3. Find the best MDC for Sensor Data Capture
4. Send message to specific MDC
MINA Server with Control Center Application
CC_Unit
Analytics
Mobility Planning Engine
Realtime Update
Path Update
12. MINA Server/Control Center Determines appropriate Emergency Response
TIER 1 Layer (Control Center)
5. Route MDC Movement Message to MDC via shortest path(Mobility Update).
11. Route MDC Packets to MINA Server/Control Center
TIER 2 LAYER (Fixed Location Access Points)
10. MDC transmits data to CC via shortest path.
TIER 3 LAYER (MDCs)
TIER 3 LAYER (MDCs)
TIER 3 LAYER (Edge Nodes)
TIER 3 LAYER (Edge Nodes)
TIER 3 LAYER (Edge Nodes)
TIER 3 LAYER (Edge Nodes)
9. Send Stored Sensor Data.
1. Emergency Event Request Received.
6. MDC Located. Proceeds close to Sensors.
7. Collects local data from on-board sensors (if reqd.)
8. Collects Stored data from sensors.
LEGEND
LOW-POWER SENSOR
ACCESS POINT
4-Wheeler MDC
2-Wheeler MDC
Pedestrian MDC
Control Center

Figure 4.6: Use Case 2 - Location Update of MDC


7. MDC Topology Change processed by MINA Server
MINA Server with Control Center Application
CC_Unit
Analytics
Mobility Planning Engine
Realtime Update
Path Update
TIER 1 Layer (Control Center)
4. Realtime location update of MDC recorded by MINA Server/CC for future path planning.
3. MDC Location Update forwarded to MINA Server/CC.
6. MDC Topology Change sent to MINA Server by AP
TIER 2 LAYER (Fixed Location Access Points)
5. MDC joins new AP.
TIER 3 LAYER (MDCs)
TIER 3 LAYER (MDCs)
TIER 3 LAYER (Edge Nodes)
TIER 3 LAYER (Edge Nodes)
TIER 3 LAYER (Edge Nodes)
TIER 3 LAYER (Edge Nodes)
LEGEND
LOW-POWER SENSOR
ACCESS POINT
4-Wheeler MDC
2-Wheeler MDC
Pedestrian MDC
Control Center
1. MDC Moves in the shanty town area.
2. MDC Sends periodic location updates to MINA Server/CC.

4.6.1 Use Case 1 - Handling Shanty Town Emergency/Disaster Response

Fig. 4.5 describes the scenario of emergency data gathering using MINA and the mobility planning components.

When an emergency event has occurred in some corner of the shanty town, we assume that the Control Center gets some notification of this event. Shanty towns typically have a cellular phone coverage that reaches practically all the area. However, the data rates may not be high/reliable enough to support any data service based applications and hence not considered to be a part of our solution.

The CC captures the event location and time of the event. The CC_Unit running on the MINA server determines the MDC that has the best chance of capturing the data. This is based on the fact that the system keeps track of the current or last known location of each MDC and also keeps track of the next location where an MDC is headed.

The CC_Unit creates an appropriate message destined for the specific MDC with a navigable path for the MDC to take next and capture the event data. In fact, the CC_Unit may have to adjust the trajectory of more than one MDC so as to capture the data from all pending events. MINA determines the best network path to route the instructions to the MDCs whose path has been modified based on the current positions of the MDCs.

MINA in the meantime has been constantly *Observing* the network state. It is able to determine the most optimal network route to the MDCs. It uses its internal query capability to forward the message appropriately. When the message is received, the MDC updates its trajectory to be followed and keeps moving unless it is interrupted to accommodate another event.

When the MDC reaches the event location, it establishes a connection to sensor node. The overlay framework of MINA captures this connection and informs the MINA server and the CC_Unit of its successful capture of data at the event location. Optionally, the MDC, using its on-board unit called MDC_Unit (Fig. 4.4) can capture additional data that may give more context to the event data collected by the sensor. All the event data collected by the sensor is routed to the CC_Unit by the MINA framework. In order to do this efficiently, MINA once again provides the optimal route for the packets to be sent to the CC.

MINA continuously monitors the state of the network. It *Analyzes* the network for any faults or congested routes and updates the routing tables via the *Adapt* functionality of all the Tier 2 nodes appropriately so that the connectivity to the MINA server can be ensured.

When the event data finally reaches the CC_Unit, it is analyzed and an appropriate response is dispatched to the event location.

The Fig. 4.5 illustrates the sequence of activities performed by MINA and the Mobility Update piece in the CC_Unit in detail for a specific MDC.

4.6.2 Use Case 2 - Location Update of MDC

We now describe a routine scenario where MINA helps provide periodic location updates to the CC_Unit. Fig. 4.6 describes the details.

In steady state, the MDCs are either en-route to an event location to capture data or may be stationary. Maintaining an accurate picture of the MDC positions across the shanty town is a crucial part of the Mobility Planning module. This can be achieved only with an infrastructure like MINA that can guarantee a periodic update.

GPS locators on the MDC_Unit allow MDCs to accurately determine their current location in the shanty town. This information may be very small packets of data that can be periodically transmitted to the central CC_Unit. In order to transmit the information efficiently and promptly to the MINA server, the MDC forwards the location information to the AP or Router. MINA maintains the routing tables of the Tier 2 to ensure that any packet destined to the Tier 1 server comes in via its most optimal route.

Thus, apart from maintaining the topological information, MINA can be very easily extended to capture these Location Update messages. The MINA framework is easily extendable to additional messages. We propose to use this framework to capture the additional data from MDCs at periodic intervals. MINA routes these messages appropriately to the CC_Unit. The Real-time Update component of the CC_Unit consumes these messages and updates its repository accordingly.

Another feature of MINA that continues to play a significant part is that as the mobile nodes move from one AP to another in the network, its movements are detected as node joining or leaving an pair of Access Points or Routers in the Tier 2 of the MINA network. Such messages are also routed to the MINA server and MINA appropriately adjusts the routing tables to reflect this change.

4.6.3 Benefits of Mobility Planning to MINA

In the previous sections, we have shown how the mobility planning plays an important part in the combined solution. One can argue that by adding this capability to MINA, we have changed MINA from being a reactive system to being a proactive system. Without the Mobility component, MINA was just observing the network state and keeping track of the mobile node movement within the network topology. However, the mobility option gives the opportunity to place a mobile node at a specific location on the map. By itself MINA never had a functionality that directs the mobile nodes to go anywhere.Adding the Mobility component allows the mobile nodes to be directed to perform tasks.

Chapter 5

CONCLUSION

In this thesis, we have described in detail the changes made to MINA's Database Access Layer. We have also described an extension to MINA that introduces a Mobility Planning capability into MINA. In order to show the true value of the solution, we have presented the revised changes to MINA in combination with Mobility Planning and applied it to a real-world scenario. We conducted a series of stress-tests on the MINA DB Access layer and have been able to give some ballpark estimate on the volume of topology changes our enhancements will support.

MINA's central, novel feature is the use of a reflective Observe-Analyze-Adapt approach to manage the underlying multi-network environment. The effective implementation of the OAA cycle is critical to the management of dynamic multi-networks. It uses a separate shallow Overlay Hierarchy of nodes to keep track of the real network topology. Efficient on-the-fly analysis of data gathered by the MINA server is crucial for decisions concerning optimization of the underlying network performance thereby delivering the desired end-user QoS requirements. We have shown that an in-memory graph representation of the underlying network topology provides a robust architecture that handles large network topologies. By modeling the network as a graph we have the ability now to apply efficient graph traversal techniques to aid the Analysis phase.

MINA has the capability to discover the underlying topology of the network using the Observe process from the first node. In reality some portions of the network does not change frequently, like the Tier 1 and Tier 2 of the network. A significant benefit of this robust DB layer is that we now have a faster initialization process to instantiate the runtime image of the network.

We have successfully put this new DB layer to stresses that can be expected of a typical multi-network. Our tests have shown that for a fixed load of topology changes, this layer provides a constant response irrespective of the size of the node network. A predictable response allows us to size a system right and set an expectation of the volume of changes a system can handle in a given instant. Based on the time taken to absorb changes, we can the set the frequency of updates such that there is no backlog.

The implementation of MINA was designed to treat all artifacts of the network topology as Java artifacts with little emphasis on where the data persists. An Object-Relational Mapping (ORM) layer was used to bridge the Java object with the database implementation. With this work, we have realized that when the design a system incorporates the persistence of the data in a specific technology, then it is always better to investigate ways that allow us to use the benefits of the underlying technology.

We had some past work on the design of a Shanty Town Emergency Response system. We had proposed Mobile Data Collectors (MDCs) navigating the complex road network of a shanty town to capture emergency related information and relay it to a Control Center(CC) for appropriate redress. Given some parameters on the location of the MDCs and the coordinates of the new and pending emergency events that need to be serviced, we provided some efficient algorithms that show a very high rate of data capture. However, we did not work on the task of relaying the packets of data gathered from the emergency location to the CC. In a shanty town assuming a guaranteed connection from the MDC to the server at the CC is unrealistic. We therefore propose here a multi-network deployment of fixed nodes like Access Points and Routers that these MDCs could connect to. By deploying redundant hardware, we could minimize the *dead zones* (areas with no WiFi coverage) in the shanty town and provide a path to the server at the CC.

We have shown this topology can be extended with a simple multi-network topology. We can consider instrumenting the shanty town liberally with 100s or more low-cost short-range sensors operating on point-to-point or other short range networks. They capture the most accurate information of the emergency event as and when it occurs. However, the sensor nodes do not have the ability to send this information to the CC. We therefore dispatch a MDC via a message through MINA. MINA allows us to track the position of these MDCs near real-time.

We thus show that MINA and Mobility Updates go hand in hand and improve the perceived value of the combined solution. By adding the mobility component to the MINA server, we now attach a mission to the MDCs rather than be opportunistic in the data capture. We believe that this is a very valid usecase for MINA.

In the discussion on the topology for a shanty town emergency response, we indicated that the entire shanty town can be instrumented with low-cost, low-power sensors. However, we can minimize the complexity of the system by eliminating the entire Tier 3 - Edge sensors altogether. In our proposed design of the MDC_Unit (Fig. 4.4), we have suggested an on-board sensor(s) installed that are capable of capturing many different parameters. Our solution using MINA combined with the node mobility therefore makes it a more complete solution. When we add a DTN capability to the MDCs, our solution will then make this implementation a more robust network applicable for real-world shanty town scenarios.

Scope for Improvement - Potential Next Steps

In the short time-span that we had to complete this dissertation, we realize we may have left a lot of room for improvement. We take this opportunity to enumerate them and perhaps tackle it in future work.

- The current implementation of MINA is based on a centralized server. Perhaps we should consider decentralizing the server and find ways to manage the network appropriately.
- The Database Access layer has been implemented with plain JDBC and SQL to prove the need for optimization at the DB design/access level. It does perform well. However, there are other persistence layer implementations in Java that address the shortcomings of Hibernate that we have not considered.
- We could explore the use of Graph databases like neo4j or other technologies that target the persistence of networks more natively. We may be able to derive much more scalability for managing multi-networks this way.
- We would have liked to test the modifications we made to MINA with a public dataset or generate one ourselves that mimics the Shanty Town Emergency Response topology. We will need to simulate the Delay Tolerant Network(DTN) techniques for MDCs to use before they can transmit to the CC. We would like to see the impact of these delays in the effectiveness of the emergency response.
- We would like to test the performance of the mobility planning algorithms with the location updates coming through MINA. It is important to measure the overheads generated by MINA in a true network simulator.

Bibliography

[1] OpenStreetMap.

[2] The JUNG (Java Universal Network/Graph) Framework.

[3] Mumbai's Shadow City. National Geographic Magazine - NGM.com, May 2007.

[4] BBC. Urbanization in LEDCs. BBC - GCSE Bitesize, May 2007.

[5] T. Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3), 2006.

[6] P. Bellavista, A. Corradi, and C. Giannelli. The real ad-hoc multi-hop peer-to-peer (ramp) middleware: an easy-to-use support for spontaneous networking. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 463–470. IEEE, 2010.

[7] R. Biswas and E. Ort. The java persistence api-a simpler programming model for entity persistence. *Sun, May*, 2006.

[8] R. Dondo and J. Cerdá. An milp framework for dynamic vehicle routing problems with time windows. *Latin American applied research*, 2006.

[9] A. Hilal, J. N. Chattha, V. Srivastava, M. S. Thompson, A. B. MacKenzie, L. A. DaSilva, and P. Saraswati. CRAWDAD data set vt/maniac (v. 2008-11-01). Downloaded from http://crawdad.org/vt/maniac/, Nov. 2008.

[10] R. L. Iannario. Design and implementation of effective monitoring solutions for heterogeneous wireless networks. Master's thesis, Universita Di Bologna, 2011.

[11] G. Jain, S. Babu, et al. On disaster information gathering in a complex shanty town terrain. In *GHTC-SAS*. IEEE, 2014.

[12] S. Kapadia, B. Krishnamachari, and L. Zhang. Data delivery in delay tolerant networks: A survey, 2011.

[13] G. Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Jl. of Operational Research*, 1992.

[14] C. Mascolo and M. Musolesi. Scar: Context-aware adaptive routing in delay tolerant mobile sensor networks. In *International Conference on Wireless Communications and Mobile Computing*.

[15] OCHA. Humanitarian issues: Tackling environmental emergencies in slums. OCHA, Feb 2012.

[16] E. J. O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356. ACM, 2008.

[17] J. Potvin, T. Kervahut, et al. The vehicle routing problem with time windows part i: tabu search. *INFORMS Journal on Computing*, 8(2), 1996.

[18] Z. Qin, L. Iannario, C. Giannelli, P. Bellavista, G. Denker, and N. Venkatasubramanian. Mina: A reflective middleware for managing dynamic multinetwork environments. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–4. IEEE, 2014.

[19] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 1987.

[20] R. Spliet, A. F. Gabor, and R. Dekker. The vehicle rescheduling problem. *Computers & Op. Res.*, 43, 2014.

[21] UDF-RCL-08-270. Empowerment of shanty towns settlers through democratic spaces. UN Democracy Fund, July 2012.

[22] Y. Wang, M. Colledanchise, et al. A distributed convergent solution to the ambulance positioning problem on a streetmap graph. *World Congress*, 19(1), 2014.

[23] X. Yi. Controlled mobility for event data collection within wireless sensor networks. Master's thesis, UC Irvine, 2012.

[24] M. Zhao and Y. Yang. Optimization-based distributed algorithms for mobile data gathering in wireless sensor networks. *Mobile Computing, IEEE Transactions on*, 11, 2012.

Appendix A

Detailed Test Results

A.1 Test Results from Random-Generated Test Data

DETAILED RESULTS OF TESTS ON RANDOM-GENERATED DATA 3/16/2015

	A	B	C	D	E	F	G	H
1	Test Group	Test Description	InitialState Number of Nodes	InitialState - Number of Links	Setup Time of a Start Network	StartState (Nodes#) for Topology Updates	StartState (Links#) for Topology Updates	Time to Load Graph Network From MySQL
2	1	1000L/250N/100A/50R	250	-	10,374.00	250	987	270.29
3	1	2000L/500N/100A/50R	500	-	32,646.93	500	1,991	282.16
4	1	3000L/750N/100A/50R	750	-	38,642.34	750	2,985	307.55
5	1	4000L/1000N/100A/50R	1,000	-	47,481.10	1,000	3,979	320.47
6	1	6000L/1500N/100A/50R	1,500	-	58,985.45	1,500	5,986	354.10
7	1	8000L/2000N/100A/50R	2,000	-	75,226.10	2,000	7,987	439.86
8								
9	2	8000L/2000N/100A/50R	2,000	-	74,244.11	2,000	7,980	421.06
10	2	8000L/2000N/200A/100R	2,000	-	74,932.05	2,000	7,987	420.25
11	2	8000L/2000N/400A/200R	2,000	-	75,596.91	2,000	7,973	421.16
12	2	8000L/2000N/800A/400R	2,000	-	73,390.36	2,000	7,975	420.65
13	2	8000L/2000N/1000A/500R	2,000	-	71,490.02	2,000	7,982	421.85
14	2	8000L/2000N/2000A/1000R	2,000	-	78,586.48	2,000	7,985	471.29
15	2	8000L/2000N/3000A/1500R	2,000	-	68,962.77	2,000	7,981	424.30
16	2	8000L/2000N/4000A/2000R	2,000	-	66,573.34	2,000	7,987	420.40
17	2	8000L/2000N/5000A/2500R	2,000	-	71,133.29	2,000	7,985	421.59
18								
19	3	1000L/250N/1000A/500R	250		11,618.59	250	980	271.47
20	3	2000L/500N/1000A/500R	500		31,942.82	500	1,983	281.52
21	3	3000L/750N/1000A/500R	750		39,861.32	750	2,979	306.19
22	3	4000L/1000N/1000A/500R	1,000		42,152.27	1,000	3,988	328.04
23	3	6000L/1500N/1000A/500R	1,500		62,063.31	1,500	5,988	360.11
24	3	8000L/2000N/1000A/500R	2,000		67,018.16	2,000	7,982	423.75
25								
26	4	1000L/250N/2000A/1000R	250		10,261.86	250	985	270.64
27	4	2000L/500N/2000A/1000R	500		31,549.62	500	1,979	290.37
28	4	3000L/750N/2000A/1000R	750		38,002.88	750	2,981	308.18
29	4	4000L/1000N/2000A/1000R	1,000		45,802.04	1,000	3,983	323.01
30	4	6000L/1500N/2000A/1000R	1,500		61,029.07	1,500	5,981	354.79
31	4	8000L/2000N/2000A/1000R	2,000		68,467.76	2,000	7,978	420.23
32								
33								

DETAILED RESULTS OF TESTS ON RANDOM-GENERATED DATA 3/16/2015

	I	J	K	L	M	N	O	P	Q	R
1	EndState - Node Count	EndState - Link Count	LOAD: Link Add Requests in Each Update	Total Link Add Messages Ignored	LOAD: Link Remove Requests in Each Update	Total Link Remove Messages Ignored	Min Time for Topology Update	Max Time for Topology Update	Avg Time for Topology Update	Min All Pairs Shortest Path Setup Time
2	250	3,204	100	202	50	1,169	564.44	2,825.08	1,424.39	0.07
3	500	4,386	100	66	50	1,211	541.24	1,249.86	797.04	0.07
4	750	5,427	100	43	50	1,235	511.67	1,072.37	735.50	0.07
5	1,000	6,440	100	25	50	1,236	534.20	1,194.69	801.69	0.06
6	1,500	8,468	100	13	50	1,245	496.05	1,186.62	790.26	0.07
7	2,000	10,468	100	11	50	1,242	555.49	1,308.74	779.77	0.06
8										
9	2,000	10,459	100	11	50	1,240	540.11	1,012.97	781.61	0.06
10	2,000	12,940	200	29	100	2,482	1,019.33	2,207.47	1,495.57	0.07
11	2,000	17,878	400	64	200	4,969	2,019.92	3,762.31	2,963.53	0.07
12	2,000	27,740	800	159	400	9,924	5,191.31	6,788.93	5,960.00	0.07
13	2,000	32,618	1,000	252	500	12,388	6,828.32	8,476.87	7,695.69	0.07
14	2,000	56,745	2,000	829	1,000	24,589	12,605.56	16,772.67	14,660.36	0.07
15	2,000	80,477	3,000	1,687	1,500	36,683	19,561.77	23,679.16	21,250.46	0.07
16	2,000	103,639	4,000	2,900	2,000	48,552	25,375.68	33,001.72	29,571.07	0.07
17	2,000	126,436	5,000	4,298	2,500	60,249	31,750.47	43,165.91	37,958.73	0.07
18										
19	250	14,696	1,000	7,448	500	8,664	6,642.08	24,472.17	8,724.87	0.07
20	500	23,047	1,000	2,620	500	11,184	6,335.09	10,016.34	7,947.81	0.07
21	750	26,016	1,000	1,322	500	11,859	5,950.14	8,268.09	7,084.30	0.07
22	1,000	27,728	1,000	817	500	12,057	6,142.09	8,375.89	7,400.91	0.07
23	1,500	30,352	1,000	438	500	12,302	6,033.95	7,306.88	6,684.61	0.07
24	2,000	32,578	1,000	271	500	12,367	5,691.30	8,066.70	6,874.14	0.07
25										
26	250	18,647	2,000	21,488	1,000	14,150	12,993.59	30,049.83	17,080.09	0.08
27	500	38,443	2,000	8,886	1,000	20,350	13,545.45	20,573.28	17,688.38	0.07
28	750	46,132	2,000	4,512	1,000	22,663	12,918.12	18,086.77	15,833.27	0.07
29	1,000	49,801	2,000	2,716	1,000	23,534	13,237.79	18,144.95	15,844.53	0.07
30	1,500	53,950	2,000	1,332	1,000	24,301	12,393.62	15,418.51	13,804.66	0.07
31	2,000	56,738	2,000	788	1,000	24,548	13,211.13	16,222.76	14,676.68	0.07
32										
33										

DETAILED RESULTS OF TESTS ON RANDOM-GENERATED DATA 3/16/2015

	S	T	U	V
1	Max All Pairs Shortest Path Setup Time	Min Time For Random Query	Max Time For Random Query	Test Run Time
2	2.15	1.51	8.59	35,965.92
3	2.18	2.32	13.02	20,329.34
4	2.15	3.42	16.10	18,838.97
5	2.21	4.42	22.55	20,547.93
6	2.19	6.57	29.04	20,362.51
7	2.25	9.83	32.63	20,254.82
8				
9	2.27	8.87	32.98	20,277.89
10	2.21	9.12	31.77	38,149.42
11	2.26	8.82	30.78	74,899.45
12	2.14	9.56	49.36	149,943.08
13	2.18	10.16	33.04	193,373.77
14	2.13	11.63	58.35	367,864.84
15	2.24	12.36	118.15	532,945.31
16	2.15	13.00	111.73	741,228.15
17	2.19	13.06	230.15	951,433.76
18				
19	2.23	2.97	16.01	218,618.23
20	2.19	4.13	21.75	199,303.35
21	2.17	4.92	21.75	177,759.99
22	2.29	5.60	24.18	185,749.18
23	2.15	9.14	28.07	167,979.76
24	2.16	0.32	62.96	172,846.66
25				
26	2.12	3.54	23.03	427,591.83
27	2.14	4.88	66.37	443,096.43
28	2.23	6.34	40.53	396,743.64
29	2.18	7.56	43.61	397,102.06
30	2.18	8.69	51.08	346,245.79
31	2.17	9.96	67.13	368,239.48
32				
33				