University of California

Los Angeles

# Communication Paradigms for Mobile Ad Hoc Networks

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

**Justin Scott Collins**

2014

Abstract of the Dissertation

# Communication Paradigms for Mobile Ad Hoc Networks

by

**Justin Scott Collins**

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Rajive Bagrodia, Chair

Consumer implementations of mobile ad hoc networks (MANETs) are rapidly becoming possible due to the proliferation and ubiquity of smartphones, tablets, and other portable computers. However, the combination of wireless communication, device mobility, and the self-organizing nature of MANETs presents a challenging environment for developing distributed applications. Networking middleware and libraries for MANET applications typically provide adaptations of traditional distributed computing paradigms designed for wired networks. In this work we quantitatively evaluate existing projects and their fundamental underlying communication paradigms using real applications in realistic MANET environments. We then propose and evaluate a new communication paradigm specifically designed for MANETs.

The dissertation of Justin Scott Collins is approved.

Mario Gerla

Stott Parker

Greg Pottie

Rajive Bagrodia, Committee Chair

University of California, Los Angeles

2014

To my mom, without whom I wouldn't be.

To my wife, without whom I couldn't be.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgments

I would like to thank my advisor Rajive Bagrodia for his patience, guidance, and belief that I would eventually complete this research.

Sincere thanks to my managers at AT&T Interactive and Twitter who were gracious enough to pay me, mentor me, and allow me the flexibility to complete my degree at the same time.

Many thanks to the many who inspired me to study computer science. Special thanks to my uncle Michael Rudd who gave me his old TRS-80 Model 100 and permanently changed my career path from hardware to software and has remained a source of encouragement and academic advice. Thank you to David Joslin who provided my first research experience and publication.

I am very grateful to have been a part of the Computer Science Graduate Student Committee (CSGSC). I would especially like to thank the founding members who were friendly enough to include me and made several years much more pleasant.

I owe a surprising amount to Tom Murphy VII, who helped me become president of my high school, motivated me to investigate programming languages, led me to pursue a PhD, and inspired many of my projects. I hope to meet him some day.

Finally, there is no way I would have made it this far without the advice and love from my mom, nor the incredible patience, understanding, and love of my wife. Thank you.

## Vita

|  |  |
|---|---|
| 2002–2005 | Helpdesk Specialist, Seattle University Law School, Seattle, Washington. |
| 2005–2006 | Systems Administrator, Klir Technologies, Seattle, Washington. |
| 2006 | B.S. (Computer Science), Seattle University. |
| 2007–2010 | Graduate Student Researcher, Computer Science Department, UCLA. |
| 2009 | M.S. (Computer Science), UCLA. |
| 2010-2012 | Security Engineer, AT&T Interactive, Glendale, California. |
| 2012–present | Security Engineer, Twitter, Inc., San Francisco, California. |
| 2013–present | Technology and Security Advisor, Guide Financial, San Francisco, California. |

## Publications and Presentations

J. Collins and D. Joslin. "Improving genetic algorithm performance with intelligent mappings from chromosomes to solutions." In Proc. of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06) (Seattle, Washington, USA, July 08 - 12, 2006).

D. Joslin and J. Collins. “Greedy transformation of evolutionary algorithm search spaces for scheduling problems.” IEEE Congress on Evolutionary Computation, 2007 (CEC 2007).

Justin Collins and Rajive Bagrodia. “Programming in Mobile Ad Hoc Networks.” In Proceedings of the Fourth International Wireless Internet Conference (WICON 2008).

Justin Collins and Rajive Bagrodia. “A Quantitative Comparison of Communication Paradigms for MANETs.” In Proc. of the 7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2010).

2013 Fukuoka Ruby Award: Outstanding Performance. Presented in Fukuoka, Japan.

Justin Collins and Rajive Bagrodia. “MELON: A Persistent Message-Based Communication Paradigm for MANETs.” In Proc. of the 10th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2013).

Justin Collins and Rajive Bagrodia. “MANET Application Development with MELON.” In Proc. of the 13th International Conference on Ad-Hoc Networks and Wireless (ADHOC-NOW 2014).

“Mobile Ad Hoc Networks are Coming, But We Aren’t Ready.” Midwest.IO 2014, Kansas City, Missouri.

# CHAPTER 1

# Introduction

## 1.1 Motivation

Mobile ad hoc networks (MANET) comprised of small, mobile, wireless devices present a new and challenging area for the development and deployment of applications. As consumer devices equipped with WiFi capabilities such as smartphones become more widespread, the possibility of impromptu networks also increases. Mobile applications are no longer limited to stand-alone or client-server programs, but can interact and form useful networks directly with each other. Such networks are ideal for situations in which there is no time to set up a fixed access point, or when there is no fixed infrastructure available. Many new applications, particularly in the consumer space, are being applied to MANET, including collaborative software such as shared whiteboards, impromptu networks for communication and entertainment, and peer-to-peer applications for file sharing. While typical examples of MANETs include military units or disaster recovery scenarios, MANETs are also useful inside buildings where cellular reception might be unavailable, where censorship blocks free speech, or even for instant networked gaming between nearby friends.

In mobile ad hoc networks (MANET), high nodal mobility causes frequent topology and route changes, making it difficult to maintain network connections between nodes. Many routes in the network span multiple wireless hops and may experience dramatic and unexpected fluctuations in quality. The combination of mobility and wireless communication creates highly dynamic network topologies in which frequent, possibly permanent disconnections are commonplace, rather than exceptional events. The dynamics of the network and the wireless channel requires changes to the networking stack and alternative solutions at the application level.

While there has been a large amount of work focused on the network stack for MANETs, especially routing, the application layer is not insulated from the challenges faced at the networking layers. Mobile applications face several challenges when compared with programs intended for standard desktops: mobile devices are generally constrained in many ways: the screen size, processor power, memory, and battery power are often limited. Development platforms for mobile devices typically provide basic libraries for application support such as menus and access to data stored on the device. Networking, however, is generally limited to sockets, TCP/IP, and HTTP. In particular, applications are expected to either be standalone, like a calculator, or to only be using the network in a client-server manner, such as accessing websites or email servers.

However, most laptop computers are already equipped with WiFi which can operate in ad-hoc mode. Smartphones with WiFi are becoming ubiquitous. In 2013, surveys showed 91% of adults in the United States had cell phones, and 56% of those are smartphones [Smi13]. Among teens, 78% had a cell phone, of which 37% are a smartphone [Ma13]. Add smartphones to the proliferation of tablets and laptops and the ability for consumers to form mobile ad hoc networks (MANETs) is quickly becoming possible. However, applications designed for these networks remain in short supply.

One way to encourage creation of MANET applications is to simplify communication between devices. Given the challenges of distributed communication in such volatile networks, MANET applications often implement an abstraction layer for network communication. The majority of these abstraction layers are based on traditional distributed computing paradigms which were not designed for unreliable, rapidly-changing wireless networks. We have examined these paradigms and the performance of their implementation for MANETs and found them to be unsuitable for general purpose communication needs of MANET applications. Therefore, we have designed a new communication paradigm specifically to meet the challenges of MANETs, rather than modify an existing paradigm which was not originally intended for the MANET environment.

## 1.2 Contributions

The first contributions of this dissertation are two comparative studies of communication paradigms used in MANET applications. The first study is a survey of existing libraries and languages used to support MANET applications. Included in the survey is a quantitative comparison using a subset of the surveyed projects to examine how well they perform in a realistic MANET environment. This was the first such quantitative comparison of these projects in identical scenarios.

Once we had surveyed existing projects, it became clear the majority of projects rely on three traditional distributed computing paradigms: publish/subscribe, remote procedure calls, and tuple spaces. In the previous study, we compared performance of projects, but the implementations were in different languages with different levels of completeness and rigor. This made it difficult to conclude anything regarding the underlying communication paradigms. To directly compare the paradigms themselves and determine their suitability as the basis for MANET applications, we implemented canonical versions of the paradigms with as much shared code as possible. In the second contribution of this dissertation, we investigated the impact of wireless and mobility at the application layer for the different paradigms via an emulated network stack, detailed wireless models, and real applications.

After studying the traditional communication paradigms and their adaptations to the MANET context, we designed a new paradigm, MELON, to specifically to meet the challenges of distributed communication in MANETs. MELON provides message persistence, reliable FIFO multicast, read-only messages, simple message streaming, private messages, and efficient bulk operations. To operate well in the distributed and unreliable MANET environment, the design of MELON avoids any global state or locking and performs all operations on-demand. Thus, the third contribution in this dissertation is the design of the MELON communication language.

Our fourth contribution is a prototype implementation of MELON. In order to evaluate the practicality of using MELON in MANET applications, we implemented a prototype of MELON as well as implementing better versions of the three traditional paradigms from above. Again, the four paradigms share as much code as possible in order to eliminate performance differences caused by different implementations. We then compared the performance of MELON with the three traditional paradigms. Besides the quantitative comparison, we also implemented several example applications using MELON to demonstrate its utility in a number of scenarios.

Finally, while empirically evaluating all these paradigms, we created an experiment coordination framework implemented using MELON itself, which is presented as a case study in Section 6.4. The framework is responsible for managing the network emulator, running the applications under review, collecting output from the applications, and collating the results. It also coordinates the emulator and applications to start and stop at the same time, as well as communicating experiment parameters to the applications. This experiment coordination framework is the final contribution of this dissertation.

## 1.3 Dissertation Organization

The remainder of this dissertation is laid out as follows:

Chapter 2 reviews the challenges presented by MANETs and the fundamental communication paradigms which have been applied to MANET applications. In Chapter 3 we compare existing projects providing communication libraries for MANETs as well as canonical implementations of the underlying communication paradigms. We present both qualitative and quantitative analysis of the projects and paradigms. Chapter 4 presents the design of MELON and its communication model, then covers the prototype implementation of MELON in Chapter 5.

Several case studes are presented in Chapter 6 using MELON in applications, including the experiment coordination framework used in our later evaluations. Chapter 7 includes performance comparisons between MELON and traditional communication paradigms and discusses the suitability of MELON as a basis for MANET applications. Finally, our results and future work are presented in Chapter 8. Full MELON application code for a news server/reader and a chat room example may be found in Appendix A.

# CHAPTER 2

# Communication Paradigms for MANETs

This dissertation is mainly an investigation into patterns of communication used in applications operating in a MANET environment. These communication patterns largely fall into categories of communication paradigms - approaches to abstracting the interactions between distributed nodes. In this chapter, we first examine the characteristics of MANETs and the desired features of communication paradigms in Section 2.1, then provide an overview of various paradigms in Section 2.3.

## 2.1 Mobile Ad Hoc Network Characteristics

Mobile ad hoc networks (MANETs) are composed of mobile nodes which may join or leave the network at any point. These nodes are typically small and battery-powered, such as smartphones, smart watches, PDAs, tablets, netbooks, or laptops. MANETs are not limited to such small nodes, however. They can include unmanned aerial vehicles (UAVs), passenger vehicles, buses, commercial trucks, and any other kind of mobile device. Typically, though, MANETs are expected to consist mostly of devices with limited battery, processing power, memory, and storage.

Besides being mobile, the other main characteristic of MANETs is their communication via an infrastructureless wireless network. In other words, nodes wirelessly communicate directly with each other or by using other nodes to relay messages. The networks are entirely self-organizing and, in the most general case, rely on no access points, cellular towers, or any other type of fixed wireless receiver.

Mobility and wireless are at the root of all of the challenges MANET applications face. The network topology can change rapidly with nodes joining and leaving at any time without warning, leading to lost information and broken network routes. Wireless communication can be disrupted in many ways, including competing broadcasts, physical obstacles, and nodal mobility.

## 2.2 Requirements

Based on the characteristics of MANETs, we have identified three key challenges when creating applications to operate in MANETs.

### 2.2.1 Disconnection Handling

Due to the unreliability of wireless communication and the ability of nodes to physically move, applications in MANETs can experience frequent disconnections. Unlike traditional distributed computing, which typically views disconnections as exceptional events and errors, applications in MANETs must be able to handle disconnections as part of their normal operation.

In a MANET, nodes are highly mobile and disconnections occur frequently, either due to channel condition variation or the mobility of destinations and intermediate nodes. Disconnections may be prolonged, brief, or intermittent and applications must handle all three. Traditional networking treats disconnections as failures, but a programming environment for MANETs needs to handle disconnections as a natural element of the environment.

Besides simply hiding disconnections, the decentralized nature of MANETs causes difficulties for any operations which require global state, a global view of the network, or long-term coordination among nodes. Communication paradigms and their implementations must be able to operate in a distributed manner.

### 2.2.2 Addressing and Discovery

The lack of infrastructure in a MANET requires a decentralized method for finding and addressing resources. Traditional approaches such as DNS cannot be maintained in a MANET, so alternative means of discovering and addressing resources must be provided. The spontaneous nature of MANETs also dictates that discovery be dynamic, as the network topology cannot be known ahead of time and may change rapidly.

Unlike a wired network with a fixed infrastructure, MANETs cannot depend on centralized look up services like DNS to find peers. Since devices are constantly joining and leaving the network and it is not possible to maintain IP addresses or URLs to locate resources, applications must be able to locate them dynamically.

Addressing resources may be accomplished at different levels of abstraction. Addresses relying on MAC or IP addresses would be too fragile in a MANET. Instead, it is preferable to be able to refer to a resource by either a name or a value describing its contents or function. This allows the resource to be addresses independent of its actual location or even implementation. For example, an application may wish to print a document, so it would send print job to a printer resource. The actual printer which is used may vary over time and location, but the application only requires that it be able to print the document.

### 2.2.3 Flexible Communication

Basic communication between devices in a wired network is generally accomplished in a one-to-one unicast manner. In a MANET, multicast communication is also common due to the broadcast nature of wireless networking and bandwidth limitations. Collaborative applications, networked games, and streaming media also benefit from group communication. Providing flexible communication is crucial to developing applications for MANETs.

Besides unicast and multicast communication, it is also desirable to support *private* unicast communication between nodes, in the sense that nodes cannot eavesdrop on or disrupt each other's communication within the communication paradigm. Many mobile applications provide private one-on-one communication, such as instant messages, SMS, direct messages on social networks, email, and other confidential communication.

A general purpose communication paradigm should provide easy access to all kinds of communication patterns.

## 2.3 Communication Paradigms

Work in this area has mostly focused on adapting existing distributed computing paradigms to the mobile ad hoc environment, as we will discuss in this section.

It is useful to describe communication paradigms in terms of temporal and referential coupling [TS02]. When a paradigm is temporally coupled, it requires the sender and the receiver to both be present for a message to be sent. If it is temporally decoupled, a message can be stored at sending time and then delivered when the receiver becomes available. Referential coupling indicates whether or not the sender and receiver need to be directly aware of each other. A referentially coupled system explicitly addresses receivers, while a referentially decoupled system does not need to know with which nodes it is communicating (e.g. IP addresses), it operates at a higher level of abstraction.

Both temporal and referential decoupling are preferred in a MANET. If a paradigm is temporally decoupled, it is more likely to be able to handle disconnections and changing network topologies. Referential decoupling is useful, because it avoids the need for a centralized naming system and allows remote resources to be addressed using application-level semantics, rather than having to drop down to the network layer. This also allows resources to logically move between physical hosts without changing the address used by the application.

A third type of coupling, synchronization, is mentioned in [Eug03]. If a paradigm implements synchronization decoupling, then the message sender is not blocked when sending a message, and a receiver does not block when waiting for a message. In other words, sending and receiving of messages occurs outside the main thread of the application. Typically this means sending a message returns immediately and received messages are handled asynchronously in a callback method.

### 2.3.1 Publish/Subscribe

The *publish/subscribe* paradigm divides processes into publishers and subscribers. In topic-based publish/subscribe, publishers broadcast messages tagged with one or more topics. Subscribers receive the messages by subscribing to one or more topics and specifying a callback to receive the publications asynchronously and separately from the main process thread. Publish/subscribe does not guarantee any ordering of publications nor does it specify how to deliver messages if the subscribers is not available at the time of publication. Publish/subscribe is temporally, referentially, and synchronization decoupled. Messages may be sent and received at any time, and it is possible to handle multiple incoming publications concurrently.

Publish/subscribe is completely oriented towards group communication. The only method of communication is publishing a message to a topic. There may be zero or more subscribers to that topic, all of which will receive the message, provided the subscribers are available. One-to-one communication can only be achieved by coordination at the application layer. For example, two nodes may agree to communicate via a topic which they assume no one else will use.

Topic-based publish/subscribe is the simplest variation for subscriptions. Subscriptions may be based on hierarchical topics or tags, content of the messages, type structures, and so on.

Typically, there is a system of fixed nodes which serve as *brokers* which manage subscriber lists and delivery of messages. However, the brokering system is completely transparent to the application layer, which is only able to subscribe to topics and publish messages. This maintains the referential decoupling that publish/subscribe systems provide. Published messages may or may not be persistent, depending on the implementation. In distributed publish/subscribe such as MANETs, it is generally not expected that publishers would persist and deliver messages at a later time via brokers [Eug03], although some implementations exist [CP06]. Managing an overlay network of brokers adds considerable complexity.

STEAM (Scalable Timed Events and Mobility) [MC02] is an example of an event-based middleware which uses publish/subscribe for communication. REDS (REconfigurable Dispatching System) is a framework for building publish/subscribe systems in highly dynamic networks.

### 2.3.2 Remote Procedure Call

Remote procedure call (RPC) is a form of message passing in which remote procedures or method invocations are syntactically similar or identical to local function calls, but the code is actually executed on a remote machine. When the procedure is called, arguments are copied to the remote machine, which executes the requested method and sends the resulting value back to the local machine. RPC is temporally coupled, since remote methods must be available at the time of the call. However, RPC is referentially decoupled, since the application does not know which node it is communicating with, only that it supports a given method.

A host can "export" an object to be accessed remotely. Remote hosts discover these remote objects by name or type and then invoke methods defined on the object. RPC is spatially coupled, since the remote object must be available in order to invoke the method. Arguments may be passed to the remote method and the return value of the method is returned to the local process. Since RPC implies code execution, failures during the remote calls can be dangerous [TS02].

Group RPC invokes the same method with the same arguments on all matching remote objects. In a MANET, group RPC must be performed asynchronously to be practical: the call may return multiple values but the client cannot rely on all remote hosts returning a value successfully. A timeout could be used instead, but a short timeout would cause unnecessarily lost messages, while a long timeout could cause long delays in the execution of the application.

RPC inherently supports one-to-one two-way communication. Arguments to a method can be considered sending a message, and the return value of the method can be thought of as sending a reply message. RPC can also support group communication by invoking a given method on multiple remote hosts.

Many-to-Many Invocation (M2MI) [KB02] is an RPC implementation for MANETs which avoids costly ad hoc routing and discovery by broadcasting messages. Messages are addressed by object type, so if a device hosts an object of the addressed type, it will pass the message to that object.

The advantage of M2MI is simplicity. As messages are simply broadcast without expectation of reply, there is no need to worry about return values or blocking while waiting for confirmation. At the language level, there is no difference on the sender's side between a message which is actually received and one which is not received by anyone. Though this provides simplicity, it also means more work for the programmer. As there is no guarantee of message delivery, any functionality beyond simple unidirectional message passing must be implemented on top of M2MI.

AmbientTalk [CMB07] is a complete object oriented language inspired in part by M2MI's message passing. AmbientTalk implements a higher level abstraction of resource discovery and disconnection handling which is absent from M2MI, but retains the idea of object handles and remote method invocations. All remote events are handled asynchronously by AmbientTalk through the registration of callbacks. A block of code may be registered to be invoked when discovering a certain resource type. AmbientTalk also adds the ability to receive values from method invocations on remote objects through the use of futures. By default, messages sent to remote objects are buffered until they can be sent. The programmer can also choose to break the connection and recall buffered messages.

### 2.3.3 Tuple Spaces

*Tuple spaces*, introduced in the Linda [GC92] coordination language, operate on a distributed shared memory space of typed, ordered tuples. Tuples are sent using the **out** operation then retrieved by matching templates with **rd**, which copies the tuple, or **in**, which atomically removes the tuple from the tuple space. If multiple tuples match, one is chosen nondeterministically. Tuple spaces have strict semantics for **rd** and **in**: if a matching tuple exists, it *must* be returned. **rd** and **in** are blocking operations, but typically non-blocking versions are provided called *inp()* and *rdp()* which return immediately even if there are no matching tuples.

Tuple spaces are both temporally and referentially decoupled. Tuples may be read any time after they have been written to the tuple space, provided they have not been removed in the meantime. Where a tuple may physically reside is completely unknown to the application; all communication is performed by requesting a tuple matching a particular template. Tuple spaces easily support group communication, since a single tuple may be read by any number of nodes. One-to-one communication can be achieved by agreement on a particular tuple field, although this does not guarantee unintended receivers will not read or even remove the tuple.

An issue particular to tuple spaces is the “multiple read problem”: nondestructively retrieving all matching tuples requires repeated **rd** operations, which may return any matching tuple [RW96]. One solution is to use a mutex tuple to gain exclusive access to the tuple space, remove all matching tuples using **in**, replace the tuples, and then release the mutex. However, this approach prevents concurrent access and is dangerous in MANETs where the node with the mutex may disappear. Another solution uses a counter in each tuple. The application reads tuple 1, then tuple 2, etc., in order. Not only does this introduce performance issues (if a particular tuple is unavailable, for example, the application must wait on it), but if multiple processes are producing tuples they must coordinate to generate consistent counters, essentially requiring another global mutex.

Yet another option, proposed in [ER01], relies on being able to use transactions over the tuple spaces. Instead of using **rd**, the application uses **in** inside a transaction to retrieve all desired tuples, then cancels the transaction to replace all of the tuples back into the tuple space. This causes the tuples to be unavailable during the transaction and requires support for transactions, which does not seem common. A final option, proposed in [RW96], is to simply introduce a new **copy-collect** operation which copies all matching tuples. We will find in Section 6.5 that even this is not sufficient for typical needs of an application.

Several projects have adapted tuple spaces to MANETs, including L$^2$imbo [Wad99], LIME [Mur06], TOTAM [Ga13], and EgoSpaces [JR06]. LIME (Linda in mobile environment) is a well-established implementation of tuple spaces [GC92] for mobile environments. Each device or agent has its own tuple space, which can merge with remote tuple spaces when they come into range of each other. Tuples can be read and written from specific locations, but can also be read or written to the “federated” tuple space which includes the local tuple space and any tuple spaces which are currently merged with it. However, the tuple will reside in a particular tuple space, so when that device or agent moves away, the tuples in that tuple space will move with it and be out of reach. LIME requires tuple spaces joining or leaving the federation do so explicitly, which is clearly at odds with the dynamicity of MANETs where disconnections may occur unexpectedly.

Besides the typical tuple space operations, LIME also provides *reactions* which can be triggered when a matching tuple is added to the tuple space. Reactions allow asynchronous and push-based communication when tuple spaces are typically synchronous and pull-based. The implementation of reactions registers hooks on each remote host. When a new tuple is stored, the hooks are evaluated first to see if any match the tuple. For those that do, the registering host is notified and a copy of the tuple is transferred.

Note that *all* reactions must complete before any other actions may occur. This includes a tuple space leaving the tuple space federation. As discussed in [Ca01b], this requirement can lead to livelocks were a node is not "allowed" to leave the federation due to mutually recursive reactions. LIME II [Aa09], Limone [Fa04], and CoreLIME [Ca01a] are proposed to meet shortcomings in LIME.

MESH*Mdl* [HHM07] is another tuple space implementation, but varies slightly from the LIME model. In MESH*Mdl*, there is a single tuple space shared between all applications on a device. All communication between applications is performed via this shared tuple space. Remote tuple spaces are not shared like in LIME, but are accessible for reads and writes only: it is not possible to remove tuples from a remote tuple space. MESH*Mdl* supports mobile agents and recommends using them if actions need to be performed on a remote tuple space. MESH*Mdl* also adds the idea of being able to automatically write, read, or block tuples from other tuple spaces.

Tuples on the Air (TOTA) [MZ04] also implements a tuple space for MANETs, but differs from LIME and MESH*Mdl*. Rather than storing tuples on a particular device, tuples in TOTA are propagated through the network according to rules specified per tuple. As the tuples move through the network, they can acquire context information about the network, such as how many hops they have traveled from the source.

### 2.3.4 Mobile Agents

Software “agents” are software units which operate autonomously, reacting to their environment or outside instructions to perform tasks. “Mobile agents” are agents which may move from one system to another, including from one physical host to another [Lan98].

Using mobile agents in MANETs allows programs to move code to the data, rather than communicating the data to a running program. Code migrates to where data or resources are located, executes at that location, and then can return to its original location to report the results. This allows mobile devices, which usually have little processing power, to offload computationally intensive work to machines with more processing and power resources.

We include mobile agents here because our original investigation into projects providing communication libraries for MANETs included SpatialViews [NKS05a], a language extension to Java ME which allows programs to iterate over groups of devices. The code inside the loop is executed on the initial device and then migrates to the next, eventually making its way back to the initial node. This allows for complex operations to be written easily, as the language has built-in support for such things actions as reduction operations. The iteration itself is generally done according to some physical layout, although it is possible to iterate over all objects or to use logical locations instead. More information about SpatialViews along with performance data is presented in Section 3.1.

However, while copying application code from host to host works for specific work types, it is not an efficient method for general-purpose communication. Figures 3.1 and 3.2 show the high overhead incurred when using mobile agents. A ten year retrospective publication from the authors of an influential paper on mobile code [CPV97] notes the high complexity of using mobile agents and the narrow set of scenarios in which they are advantageous [CPV07]. Therefore, after evaluating them in Section 3.1 we drop mobile agents from consideration.

# CHAPTER 3

# Analysis of Existing Paradigms

## 3.1 Project-Based Comparison

The current projects for developing software in MANETs fall into three broad categories: runtimes, languages, and middleware, which offer increasing levels of abstraction for the developer. They can also be combined: a middleware solution can be written in a language which uses one of the basic runtimes for mobile devices. In many cases projects also provide additional resources for software development such as debuggers and emulators for testing code. Table3.1 summarizes the representative projects discussed in this section with respect to the requirements in Section 2.2. Broader overviews focusing only on middleware for MANETs can be found in [HAM06] and [SA14].

### 3.1.1 Project Overview

#### 3.1.1.1 Runtimes

Runtimes in this context are virtual machines for languages which are specifically intended for use on small, resource-constrained mobile devices. Runtimes are useful because they provide good portability for applications and thereby simplify some of the application development process.

Two common runtimes for mobile devices are Java ME [Micb] and the .NET Compact Framework [Mica]. A third runtime, BREW [Qua], is a proprietary

Table 3.1: Projects Summary

| **Project** | **Category** | **Disconnection Handling** | **Addressing and Discovery** | **Communication** |
|---|---|---|---|---|
| LIME | Middleware | Tuple removal | Merged tuple spaces | Tuple space |
| MESHMdl | Middleware | Tuple removal | Tuple exchange | Tuple space |
| TOTA | Middleware | Connectionless | Tuple propagation | Tuple space |
| STEAM | Middleware | Connectionless | Event content | Publish/subscribe |
| SyD | Middleware | Object proxies | Object type | Message passing |
| M2MI | Language | Connectionless | Object type | Message passing |
| AmbientTalk | Language | Flexible references | Object type | Message passing |
| SpatialViews | Language | Connectionless | Object type | Code migration |
| .NET CF | Runtime | None | URL | Sockets |
| Java ME | Runtime | None | URL | Sockets |

product from Qualcomm. These runtimes focus on using few resources and providing libraries for application development, especially user interfaces. They do not provide much networking support beyond basic sockets and HTTP support. While it is possible to use these runtimes as foundations for better abstractions, they provide little on their own to support MANET applications and will not be considered in the following comparisons.

#### 3.1.1.2 Languages

A language in this dissertation refers to any language, language extension, or library which provides new language constructs for programming in a MANET. Languages often include their own runtime or are built on top of existing runtimes. Libraries and language extensions are likely to be easier for developers to use if they are already familiar with the base language.

M2MI [KB02], AmbientTalk [CMB07], and SpatialViews [NKS05b] are three language-based projects intended for MANETs. Many-to-Many Invocation (M2MI) avoids costly ad hoc routing and discovery by broadcasting messages. Messages are addressed by object type, so if a device hosts an object of the addressed type, it will pass the message to that object.

The advantage of M2MI is simplicity. As messages are simply broadcast without expectation of reply, there is no need to worry about return values or blocking while waiting for confirmation. At the language level, there is no difference on the sender's side between a message which is actually received and one which is not received by anyone. Though this provides simplicity, it also means more work for the programmer. As there is no guarantee of message delivery, any functionality beyond simple unidirectional message passing must be implemented on top of M2MI.

AmbientTalk is a complete object oriented language inspired in part by M2MI's message passing. AmbientTalk implements a higher level abstraction of resource discovery and disconnection handling which is absent from M2MI, but retains the idea of object handles and remote method invocations. All remote events are handled asynchronously by AmbientTalk through the registration of callbacks. A block of code may be registered to be invoked when discovering a certain resource type. AmbientTalk also adds the ability to receive values from method invocations on remote objects through the use of futures. By default, messages sent to remote objects are buffered until they can be sent. The programmer can also choose to break the connection and recall buffered messages.

SpatialViews takes a completely different approach than M2MI and AmbientTalk. SpatialViews is a language extension to Java ME which allows programs to iterate over groups of devices. The code inside the loop is executed on the initial device and then migrates to the next, eventually making its way back to the initial node. This allows for complex operations to be written easily, as the language has built-in support for such things actions as reduction operations. The iteration itself is generally done according to some physical layout, although it is possible to iterate over all objects or to use logical locations instead.

#### 3.1.1.3 Middleware

Middleware is software which manages interaction and communication between applications, as well as providing various services which may be used by applications. Middleware may also include supporting libraries which can be used by applications.

LIME (Linda in mobile environment) [Mur06] is a well-established implementation of tuple spaces [GC92] for mobile environments. Each device or agent has its own tuple space, which can merge with remote tuple spaces when they come into range of each other. Tuples can be read and written from specific locations, but can also be read or written to the "federated" tuple space which includes the local tuple space and any tuple spaces which are currently merged with it. However, the tuple will reside in a particular tuple space, so when that device or agent moves away, the tuples in that tuple space will move with it and be out of reach. LIME does not currently have an implementation intended for mobile devices smaller than laptops, though there are variations of LIME intended for sensors.

MESH*Mdl* [HHM07] is another tuple space implementation, but varies slightly from the LIME model. In MeshMdl, there is a single tuple space shared between all applications on a device. All communication between applications is done through this shared tuple space. Remote tuple spaces are not shared like in LIME, but are accessible for reads and writes only: it is not possible to remove tuples from a remote tuple space. MESH*Mdl* supports mobile agents and recommends using them if actions need to be performed on a remote tuple space. MESH*Mdl* also adds the idea of being able to automatically write, read, or block tuples from other tuple spaces.

Tuples on the Air (TOTA) [MZ04] also implements a tuple space for MANETs, but differs from LIME and MESH*Mdl*. Rather than storing tuples on a particular device, tuples in TOTA are propagated through the network according to rules specified per tuple. As the tuples move through the network, they can acquire context information about the network, such as how many hops they have traveled from the source.

Haggle [SSH07] is a middleware for mobile networks which allows tagged data to persist in a publicly searchable storage space. The Haggle middleware focuses heavily on abstracting the network layers by implementing its own routing and naming schemes, which allows a single name to map to several addresses such as MAC addresses, phone numbers, or email. Haggle proposes folding in many routing protocols, MAC protocols, and application protocols into the middleware and dynamically selecting and combining them.

SyD (System on Mobile Devices) [Sus04] is a complete middleware solution for MANETs. The middleware centers around the idea of object registries which allows service registration and lookup. Methods can then be invoked on these remote objects. Disconnection is handled by allowing objects to also provide proxy objects. If an object is unavailable, the method invocation will be handled by the proxy object, which can then perform an action specific to that service. For example, the proxy may buffer the request and send it later, or send back a cached or default response.

STEAM (Scalable Timed Events and Mobility) [MC02] is an event-driven middleware which uses a publish/subscribe [Eug03] mechanism for propagating events. STEAM uses the concept of proximity groups for communication, limiting events to the local geographic area. Events are propagated by subscribers only when the subject and proximity match. Events are further filtered on the subscriber side by content, which determines if an event is delivered to the local application.

### 3.1.2 Suitability for MANETs

#### 3.1.2.1 Disconnection Handling

The main challenge in MANETs is handling disconnections, which may be intermittent, prolonged, or permanent. For example, at a busy conference there may be many mobile devices in contact with each other, but distance and physical obstacles may cause intermittent disconnections. Routes may also break and reform due to mobility or channel variations, possibly causing prolonged disconnections, but connections are eventually regained. When the attendees all leave, it becomes unlikely their devices will ever be in contact with each other again, making the disconnection permanent. A programming environment for MANETs must be able to handle all three kinds of disconnections.

One solution, used in LIME, MESH*Mdl*, and TOTA, is to use tuple spaces for communication. Tuple spaces exhibit both spatial and temporal decoupling, meaning that messages being sent do not need to be addressed to a particular recipient nor does the recipient need to be present when the message is sent. Tuple spaces generally operate by reading, writing, and taking tuples to and from a shared location. Rather than sending a message directly to a recipient, a tuple is written to the tuple space and can be read or taken from the tuple space by other clients. This allows the tuple space to withstand disconnections.

For example, a tuple may be written out to the tuple space and then retrieved by a different client an arbitrary amount of time later. The client which retrieves the tuple may not even be in existence when the tuple was written. However, there is still a problem if the writer of the tuple disconnects before the tuple is read by a receiver. For LIME and MESH*Mdl*, where the tuple space is associated with a particular devices, the tuple space is only available when the sender and the receiver are able to communicate directly with each other. In TOTA, tuples are disseminated throughout the network and can survive even if the original sender disconnects.

A different approach, used by M2MI and STEAM, is to forgo connections completely. Messages in M2MI are sent with no expectation of reply. In the general case, messages are sent to a particular object type, to be processed by any device hosting an object of that type. Messages are broadcast with no buffering whether or not there is a receiver available. This provides even more decoupling than tuple spaces, but tuple spaces have the advantage of having some feedback about a tuple's status. The sender can check if a tuple has been removed from the tuple space or not. If it has, the sender can have some assurance the tuple was received by someone, otherwise it will be available until removed by the original sender or another client.

STEAM avoids connections by filtering events on the subscriber's side. This eliminates the need for publishers to keep track of subscribers and completely decouples the two. However, publish/subscribe in a MANET environment does not provide any message reliability. Any message reliability or disconnection feedback would need to be implemented on top of the publish/subscribe framework.

Code migration, the approach used by SpatialViews, does not maintain connections, but can be affected by disconnections if the device currently executing the mobile code fails or leaves the network before completion. Most of the devices in the network will not be involved in executing code at any particular moment, in which case their failure or disconnection from the network would not have an effect. When it does have an effect, however, it may cause the entire iteration to fail. This can be mitigated by using a form of parallel iteration over the devices. Since the iteration order in SpatialViews is nondeterministic, it does not provide message reliability.

A third approach, implemented in AmbientTalk, relies on event handling and futures. Event handlers can be registered for various events, such as discovery of a service or disconnection of a remote object. Figure 3.1 illustrates the use of two of these callbacks to discover a printer service. Once a printer is discovered, a document is sent to be printed and a status message is returned. If a remote object is discovered and later moves out of range, AmbientTalk can call the disconnection code. By default, messages sent to a disconnected remote object will be buffered until it is possible to send them. This provides a solution for intermittent and even prolonged disconnections. If a remote object is disconnected for too long, the programmer can recall all buffered messages and close the connection.

Listing 3.1: Printer Discovery in AmbientTalk

```
def print(doc) {
 when: Printer discovered: { |printer|
   when: (printer<-print(doc)) becomes: {|res|
     system.println("Status: " + res);
   };
 };
};
```

AmbientTalk also offers AmbientReferences [CDM06], which are related to the M2MI model of object handles. AmbientReferences have a specified flexibility which determines how disconnections are handled. Sturdy is the default model of using buffered messages which will be delivered upon reconnection. Elastic references wait a specified amount of time before severing the connection and rebinding to another object of the same type. Fragile references will break immediately upon disconnection and rebind to another object as quickly as possible.

The approach used by SyD is to offer the ability for the application designer to specify how to handle disconnections. In SyD, it is possible to provide proxy objects which will be called when the actual remote object is unavailable. These objects can then handle the invocation in an object-specific manner, such as buffering or returning a default value.

#### 3.1.2.2 Addressing and Discovery

Unlike a wired network with a fixed infrastructure, MANETs cannot depend on centralized look up services like DNS to find peers. Since devices are constantly joining and leaving the network and it is not possible to maintain IP addresses or URLs to locate resources, applications must be able to locate them dynamically.

The tuple space implementations of LIME, MESH*Mdl*, and TOTA automatically discover neighboring tuple spaces. LIME will merge tuple spaces with the same name, while MESH*Mdl* does not merge tuple spaces, but uses special tuples to provide a method of addressing a remote tuple space. Tuple spaces can be used for service discovery by writing out tuples which describe available services, or by writing out tuples intended for a specific service, which will read the tuples when it is available.

Addressing is not necessary in general in tuple spaces, as it can be assumed a given service and a client will have pre-agreed upon tuple template to use for communication. For example, a ubiquitous application used by a museum might have "information points" with information specific to a location. The information point does not need to even be aware of clients, nor do clients need to know any identifying information about the information point, as they will simply read available tuples from the information point's tuple space.

Object types for discovery and addressing is used by M2MI, AmbientTalk, and SpatialViews. This is based on the assumption that objects with the same name will implement the same services. M2MI and AmbientTalk use this with object handles which refer to a specific object type. When methods are invoked on a given handle, the remote object will correspond to the type of the object handle. M2MI, however, does not provide a method for discovery beyond manually sending out messages periodically and waiting for replies. AmbientTalk offers event handlers to be automatically called when objects of a specific type are discovered. These can be called exactly once or each time one is discovered.

SpatialViews uses object types along with spacial properties to define a “view” of the network. Once a view is created containing a given object type, SpatialViews provides a method of iterating over the available nodes within that view. The code within the iterator is executed locally on the remote devices. After the code is run, the device locates another nearby node hosting an object of the correct type and the code migrates there. Within the iteration, the code can synchronously invoke methods on the local service through an object handle. In Figure 3.2, a simple SpatialView is created to broadcast a message in a chat application. The code within the `visiteach` block (line 4) is executed locally. Therefore, unlike the AmbientTalk printer example, `c.receive(...)` is a local method call, not a remote call.

Listing 3.2: Simple Messaging in SpatialViews

```
spatialview v = ChatService;

visiteach c : v {
  c.receive(sender, message);
}
```

SyD also uses objects to invoke remote services, but it requires that these objects register themselves with neighboring devices, as well as locally.

The publish/subscribe model used by STEAM relies on subscribers knowing ahead of time what subscriptions are interesting to them. The publishers do not need to explicitly know who is subscribed, as messages are simply broadcast. However, it is possible to periodically send out messages describing available subscriptions.

#### 3.1.2.3 Flexible Communication

Basic communication between devices in a network is generally accomplished in a one-to-one unicast manner. However, in a MANET, group communication is also common, due to the broadcast nature of wireless networking and the limitations of bandwidth. Collaborative applications, networked games, and streaming media also benefit from group communication. Having both one-to-one and group communication available in the programming environment is necessary, though it may be possible to implement one with the other.

Tuple spaces lend themselves naturally to group communication. Tuples are written to shared storage space, which is globally accessible. Since tuples can be read without being removed from the tuple space, tuples are inherently one-to-many. One-to-one communication is not as directly supported by tuple space. However, tuples can be sent to a specific recipient by setting one of the fields in the tuple to an agreed-upon address. The specified recipient can look for tuples addressed to itself and take them from the tuple space.LIME, MESH*Mdl*, and TOTA support this type of communication.

M2MI and AmbientTalk support object references which can refer to all objects of a type, a selected subset of those objects, or a particular object. These handles directly correspond to broadcast, multicast, and unicast. Since AmbientTalk expects return values from messages, it is possible to receive multiple replies when sending a multicast or broadcast message, resulting in event handlers running multiple times or the return value being set more than once.

Communication in SpatialViews is done through code and variable migration. This makes it very simple to perform complex group operations such as reductions over several devices, but it makes one-to-one communication difficult. Figure 3.3 shows how it is necessary to set a variable to ensure a document is only printed by a single printer. There is also no method to provide reliable message delivery, other than iterating until a prearranged flag is set.

Listing 3.3: Printer Discovery in SpatialViews

```
Container result = new Container();
spatialview v = Printer;
visiteach p : v {
    if(result.isEmpty()) {
        String result = p.print(document);
            if(result == "success")
                result.addElement(p.getName());
    }
}
```

Similarly, publish/subscribe naturally supports group communication, but attempting to send a message to a particular recipient is not directly supported by the middleware. Publish/subscribe is intended to be used in situations with a single sender and multiple receivers and does not adapt well to sending a message to a single receiver. To do so would require the sender and receiver using a predefined addressing, similar to setting an agreed-upon tuple value in tuple spaces. Successful message delivery in the publish/subscribe is less likely than in a tuple space, since messages are not persistent in the way that tuples are.

Listing 3.4: LIME: Print Job Reaction

```
LimeTupleSpace lts =
  new LimeTupleSpace();
lts.setShared(true);
ITuple printjob =
  new Tuple().addFormal(PrintJob.class);
UbiquitousReaction ur =
  new UbiquitousReaction(printjob,
    this, Reaction.ONCEPERTUPLE);
lts.addWeakReaction(new Reaction[] {ur});
```

### 3.1.3 Applications

To better understand the effect of using different programming approaches when developing applications, two separate applications were written using AmbientTalk, LIME, and SpatialViews. These projects were selected because they represent very different approaches and had publicly available implementations. For each application, we discuss issues with disconnections, discovery, and communication.

#### 3.1.3.1 Printer Discovery

The printer discovery application illustrates how the different projects can be used to approach the problem of resource discovery in a changing network. The client needs to locate a device offering a printer service. Next it sends the print job to a printer it has found, then waits for a reply. The printer processes the job and sends back a success or failure message.

Disconnection can occur at different points in this process. The printer may go out of range after the client has discovered it, but before the job is sent, or it may go out of range after the job is sent, but before the result is returned. In AmbientTalk, the default way of handling both cases is to wait until the printer can be contacted again and then resume the connection. The print job or result message will be buffered until the connection can be made again and then the message will be delivered. This works well in the case where there is only transient disconnection, but if a client has permanently left the area of the printer the application may wait forever unless the programmer explicitly uses a timeout.

In SpatialViews, the only way to communicate between nodes is to visit them in the course of an iteration over all nodes which offer a given service. With the SpatialViews implementation of printer discovery, disconnection after discovery and before sending back the success message are essentially the same. The iteration will never complete and the originating node will eventually timeout.

One of the strengths of tuple spaces is temporal decoupling. The sender and receiver do not both need to be present at the same time for a message to be sent. The LIME version of printer discovery does not face the disconnection issues above, partially because a print job remains in the tuple space until a result tuple is received. A printer which reads the print job and then goes out of range does not affect the operation. If the client is not in range when the result tuple is sent, but reconnects later, the result tuple will still be available for it to read. Even if the client permanently leaves an area, a different printer can pick up the print job instead. The downside of this approach is that multiple printers may process the same job, wasting resources.

In this example, addressing and discovery are needed to find a printer service and also to send the result message. AmbientTalk and SpatialViews handle addressing with interface types. The printer offers a service with a typed interface and the client is able to find nearby services of a given type. Discovery is also built into both AmbientTalk and SpatialViews. In AmbientTalk, a callback function is set up to be called when the printer service is discovered, as shown previously in Figure 3.1. SpatialViews uses the idea of an iterator which loops over objects of a certain type nearby. In this respect, AmbientTalk is more reactive, while SpatialViews is proactive. The disadvantage to the SpatialViews approach is the service must be available at the time of the iteration, otherwise it will complete without a result. It will then be up to the programmer to retry the iteration until it is successful.

Discovery in LIME merely requires the registration of reactions to tuple templates. Addressing the printer is not necessary, but the result message contains a print job identifier so the client knows which job it represents.

Printer discovery does not really involve group communication, but there is one-to-one communication in the sending of the print job and the result message. In AmbientTalk, an object handle to the remote printer service is created when the the printer is found. The print job is then sent by invoking a method on the handle and waiting for a return value. On the printer side, it only needs to return a value, it does not require any knowledge of the client.

For SpatialViews, care must be taken to make sure the print job only goes to a single printer. Since the only method of communication is to visit every printer available, it is necessary to set a flag in a shared variable indicating the print job was already successfully printed and subsequent printers do not need to address it, as seen in Figure 3.3. Again, the printer does not need to know anything about the client.

In LIME, all communication is also inherently group communication, so the result tuple needs to be explicitly addressed to the client using a some sort of identification. The client will be waiting for a tuple with that specific ID.

#### 3.1.3.2 Chat

The second example is a chat application. Clients can send out public or private messages. Public messages are delivered to all other chat clients nearby, while private messages are directed to a specific recipient. As in most chat applications, there is no history and clients do not expect to receive messages sent earlier or when disconnected. Disconnection can occur at any time while clients are exchanging messages.

Disconnection has less effect in this application than with printer discovery, as the clients do not depend on the delivery of messages to continue operating. However, the AmbientTalk implementation does buffer both public and private messages and delivers them when the client reconnects, as this is built into the language. LIME also handles disconnection well in this case, since there is no need to guarantee message delivery.

SpatialViews suffers from the same issue in the printer discovery example: if the code migrates to a section of the network which then becomes disconnected from the rest of the network, or the current node goes down, the iteration just stops. Since each message is a separate iteration, it will not affect the overall operation of the application.

Addressing is handled similarly to the printer discovery example, except the user needs to know the names of other users when sending private messages. The AmbientTalk version notifies the user when other clients come into range and adds their name and object handle to a list. Figure 3.5 shows the implementation of the methods for sending out messages. Public messages are sent out to an AmbientReference, defined in line 1, which only needs to know the interface name and implicitly tracks individual clients.

Listing 3.5: Chat in SpatialViews

```
def all := ambient: Chatter
    withCardinality: omni
    withElasticity: fragile;

def sendAll(message) {
  all<-send(message, name);
};

def send(buddy, message) {
  def b := buddy_list.get(buddy);
  b<-send_private(message, name);
};
```

SpatialViews, like in the printer discovery application, iterates over all nodes with the chat interface, delivering the message to each as it visits. This is shown in Figure 3.5. For private messages, it still must iterate in the same manner, but the recipient is encoded in the message. Unfortunately, this means private messages still require visiting every node, possibly without even reaching the recipient in the case of disconnection. Likewise, LIME must rely on encoding the recipient in the message tuple and assuming no one but the intended client will read the message.

Group communication is natural for public messages and one-to-one communication for private messages. All three projects handle group communication well. AmbientTalk has omni-handles which refer to all interfaces of a given type and will broadcast the message to all nearby clients. The only communication in SpatialViews and LIME are essentially group communication, so for one-to-one communication, SpatialViews and LIME require the programmer to implement an addressing scheme on top of the group communication. The client side of the application needs to pick out private messages intended for it and ignore the rest.

### 3.1.4 Experimental Results

This section compares the performance of AmbientTalk, LIME, and SpatialViews in a regular wired LAN and in a MANET context using EXata.

The wired LAN environment provides a nearly ideal network in which the cost of communication is very low, there is little contention for the communication channel, all nodes are connected directly to each other, and collisions are minimal. By minimizing these factors, it is possible to focus the experiment results on the overhead of the programming environments. In these experiments, all nodes were directed connected to a 100Mb/s switch, providing essentially an independent, one hop channel for each pair of nodes

The results using EXata more accurately reflect the MANET environment. For these experiments, all communication was performed through EXata, which emulated an 802.11b ad hoc wireless network with an available bandwidth of 11Mb/s.

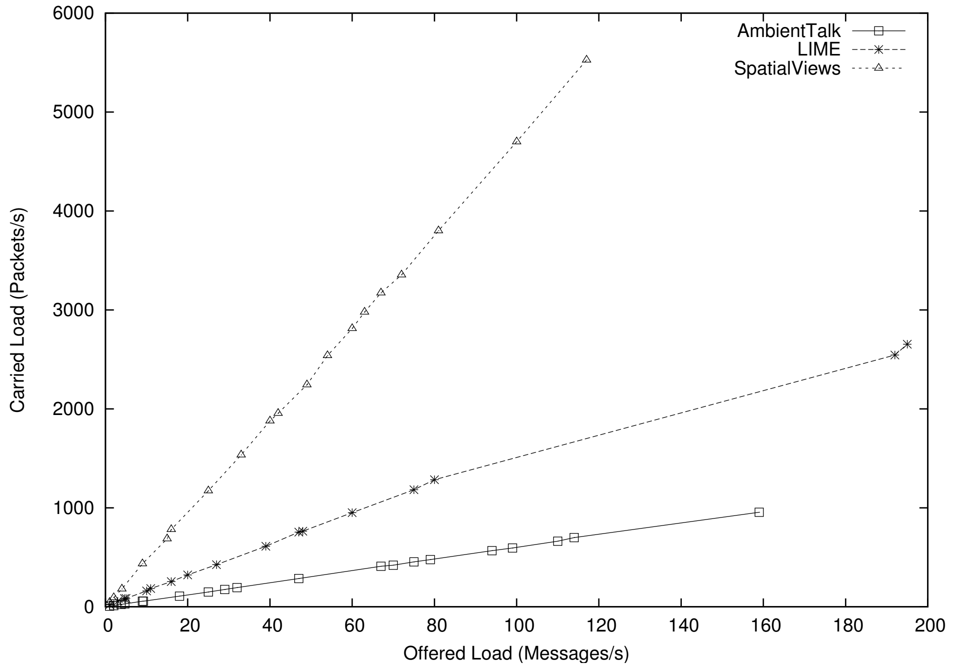


Figure 3.1: Communication Overhead with Wired Links

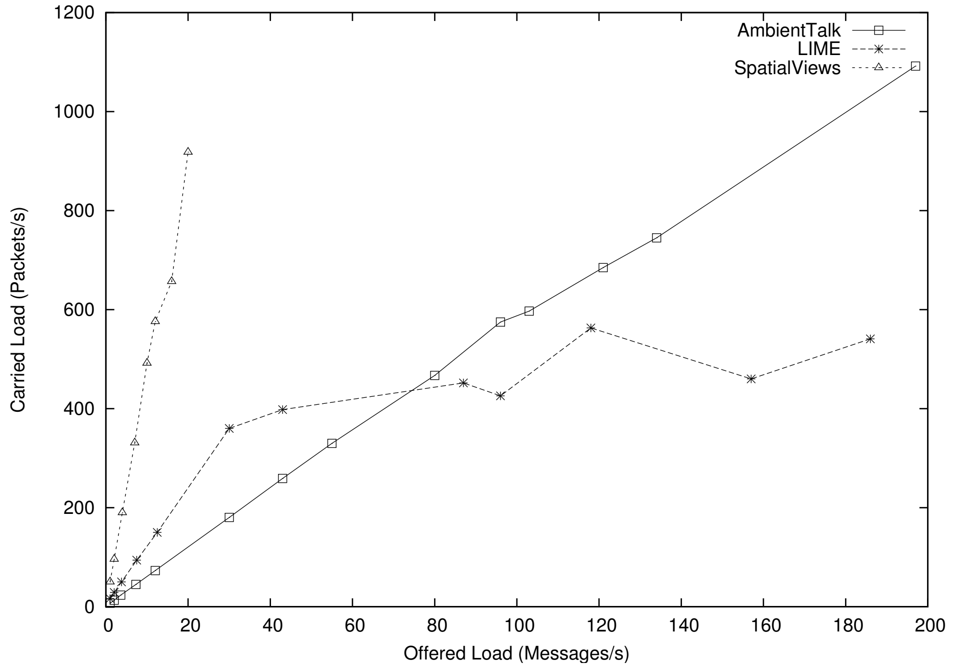


Figure 3.2: Communication Overhead with Simulated Wireless Links

#### 3.1.4.1 Communication Overhead

AmbientTalk, LIME, and SpatialViews use very different messaging systems. This experiment demonstrates the overhead for each using a client-server setup as the simplest base case. Messages are sent out from the sever to the client at an increasing rate. The number of IP packets generated by doing so include control and discovery packets. Each node is within wireless range of the others so all communication is performed over single hop routes. Figure 3.2 and Figure 3.1 show the results from wired LAN and QualNet, respectively.

AmbientTalk has the lowest overhead, as it is simply performing a method call on a remote object and there is no return value. LIME requires some communication to alert merged tuple spaces of the messages' presence and then more communication to actually transfer the tuple. SpatialViews shows the highest amount of overhead, which is expected since it is migrating code and data to communicate a simple message.

Although the results were similar in the wired LAN and EXata, the performance of SpatialViews was considerably slower, peaking at 20 msgs/s, while in the wired LAN it was possible to reach 117 msgs/s. This is due to contention for the wireless channel. It is also worth considering that LIME and AmbientTalk use asynchronous messages while SpatialViews uses a blocking synchronous message send. This allows LIME and AmbientTalk to take advantage of system level buffers, while SpatialViews cannot.

#### 3.1.4.2 Group Communication

In this experiment we consider the common situation where one node needs to request information from the rest of the network and then collect the results, with increasing numbers of receivers. The application sends out a message then measures the time elapsed for responses. For SpatialViews, this involves visiting

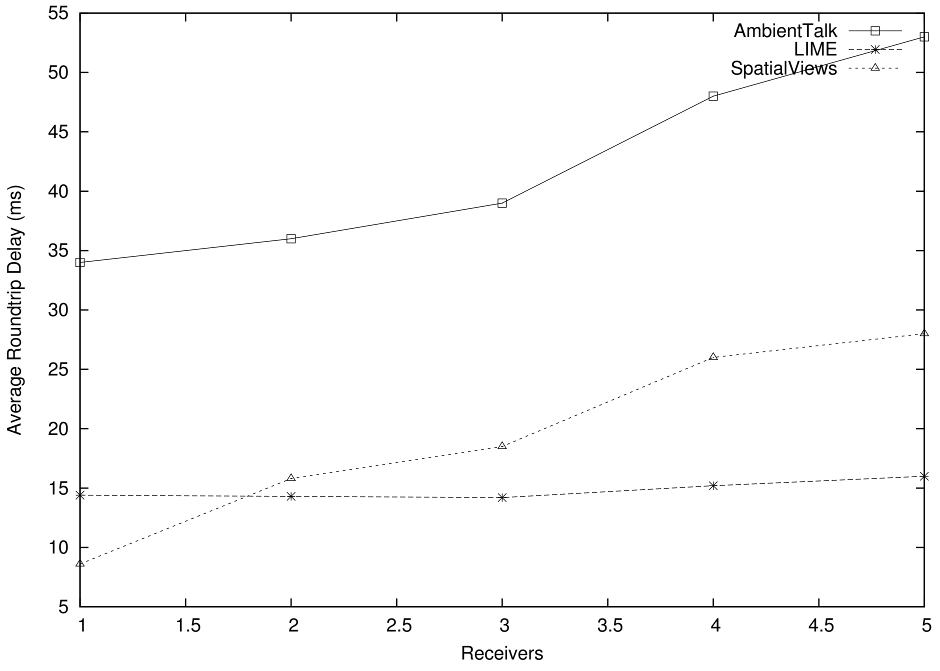


Figure 3.3: Group Communication with Wired Links

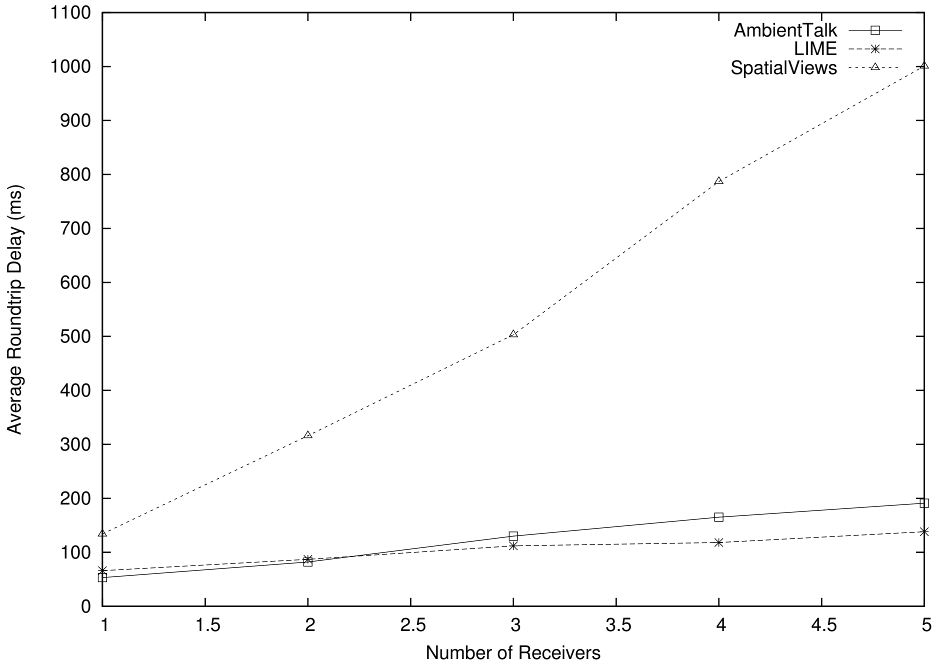


Figure 3.4: Group Communication with Simulated Wireless Links

each node and that node then visiting the sending node. The AmbientTalk version uses an omnihandle, as in the chat application, to broadcast the handle of the sender and then the receivers use the handle to send a return message. For LIME, each message is sent as a tuple, to which the receivers send a response tuple. Again, the network is set up so that no node is farther than one hop from any other node. Figures 3.4 and 3.3 show the results.

In the wired LAN, LIME shows the least variation as the number of receivers increases. This is because the sender writes out a single tuple and each receiver can respond independently and in parallel. SpatialViews slows down considerably as the number of receivers increases, since SpatialViews visits each receiver in turn and waits on a response before continuing. The delay for AmbientTalk is the highest but does not increase quite as quickly as SpatialViews. Although AmbientTalk uses a single send at the application level, messages to individual receivers are sent serially, causing the delay for the last receiver to be higher than the first.

When run using EXata, the effect of using the wireless channel is seen again. The delay with AmbientTalk and LIME increases, but not as dramatically as SpatialViews, which reaches a delay of about 1 second with 5 receivers, while a single receiver averages 134 ms. As in the previous experiment, the traffic generated by SpatialViews quickly creates conflicts in the wireless channel, causing retransmission and delay at the MAC layer.

#### 3.1.4.3 Mobility and Disconnection

In order to isolate and examine disconnection recovery, a simpler experiment in a wired LAN was performed, still using the same client-server application. In this case, a 5 second disconnection was caused by turning the network interface off and then turning it back on. Each project reacted similarly, as shown in Figure 3.5

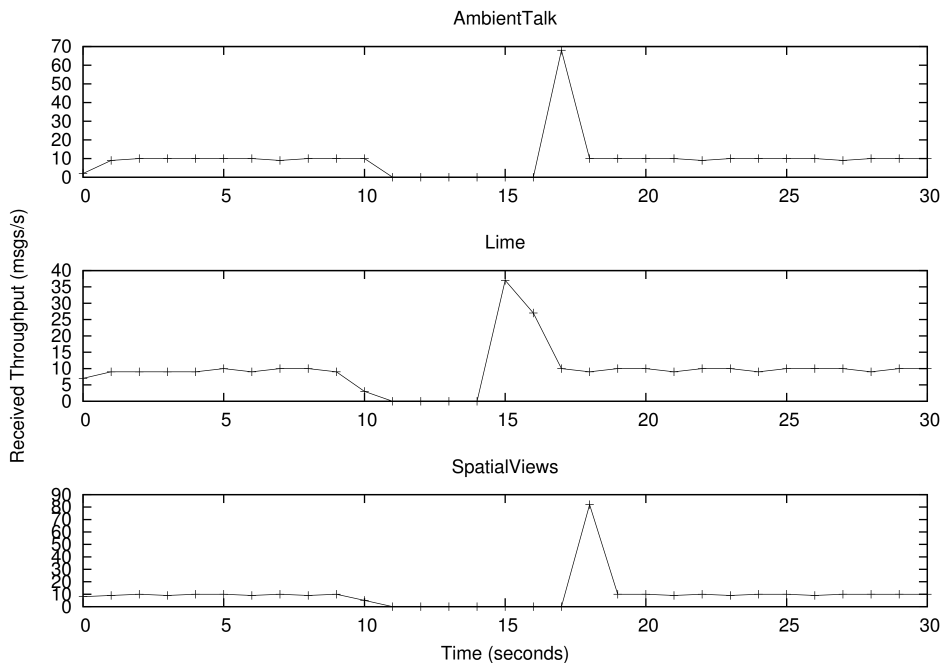


Figure 3.5: Disconnection Recovery

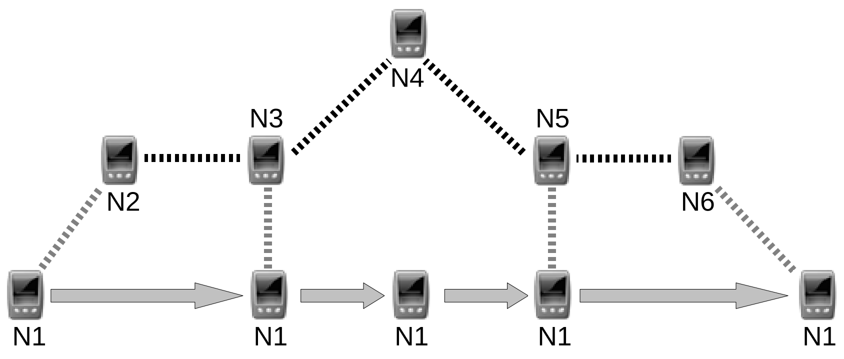


Figure 3.6: Simulated Mobility Scenario

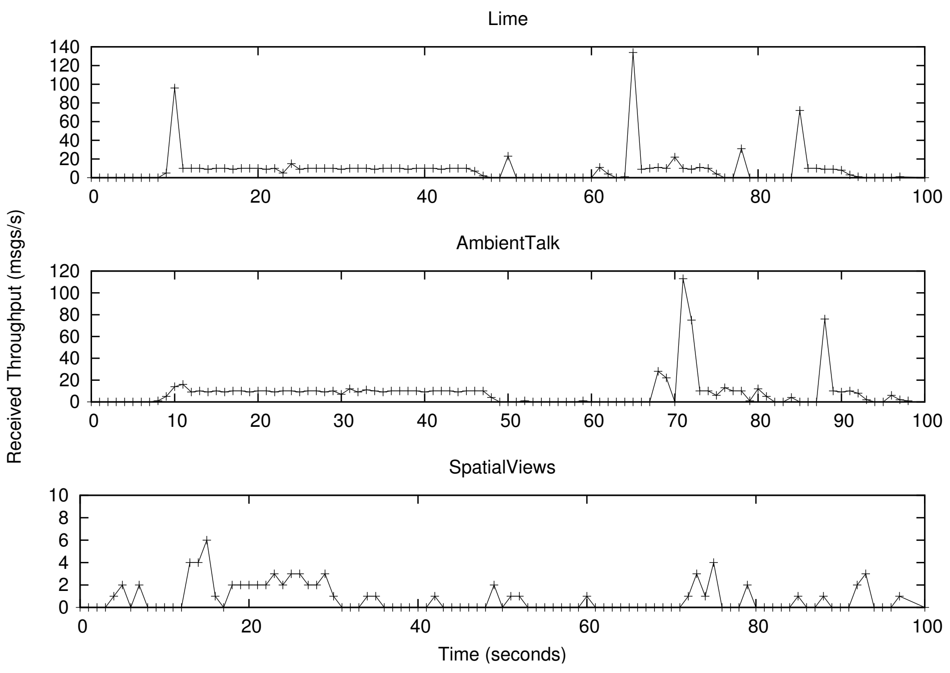


Figure 3.7: Client-Server Throughput with Mobility

though LIME showed the fastest recovery time. Interestingly, SpatialViews exhibited delivery of buffered messages. As SpatialViews does not buffer messages itself, this buffering was the result of the operating system attempting to locate the remote node.

Using EXata, it was possible to evaluate the projects in a mobile environment which provided disconnections, routing changes, and multi-hop communication. For this experiment, the network layout and mobility pattern shown in Figure 3.6 was used. Node 1 user the same client-server application as in the first experiment and is attempting to send messages to Node 4. The distance between the two nodes forces a multi-hop route through the intermediate nodes. Node 1 moves from left to right at a constant rate during a time period of 100 seconds. This experiment again measures message delivery rate. Results are shown in Figure 3.7.

The large spikes for the AmbientTalk and LIME results indicate the delivery of buffered messages. For AmbientTalk, the sender did not begin until the receiver was discovered, while LIME began sending messages immediately. The flat part of the graphs indicates when Node 1 was in between Node 3 and Node 5 and was outside the range of both. SpatialViews did not perform well in this experiment because it lacks the sophisticated disconnection handling and message buffering of the other two experiments. Also, the code migration was difficult, more time consuming, and more susceptible to disconnections. As in the previous experiments, this demonstrates the difference between experiments using a wired LAN compared to a simulated wireless network.

## 3.2 Paradigm-Based Comparison

### 3.2.1 Suitability for MANETs

#### 3.2.1.1 Disconnection Handling

In the unreliable MANET environment, disconnections frequently occur during the exchange of messages. Since disconnections can be prolonged, the networking layers will assume the connection is entirely lost and cease retrying. By having a message persist in some way, a paradigm can overcome these disconnections and deliver the message at a later time.

Publish/subscribe allows message publication without any consideration for the state of the subscribers. Publish/subscribe itself does not specify how "missed" publications should be handled. A publish/subscribe system can utilize "brokers" which manage subscriptions and facilitate delivery of publications. Brokers can then serve as message buffers and provide more reliable message delivery in the face of disconnections. In distributed publish/subscribe systems, however, the brokers must be self-organizing, and in MANET this is complicated by how quickly the network can change. It is common for distributed publish/subscribe systems not to provide message persistence. If a subscriber is not available at the time of publication, the message will not be received.

Since RPC requires a connection for communication, disconnections typically cause RPC to block a process entirely until a remote node hosting an appropriate method is available. Messages themselves only exist briefly during the RPC transaction. Messages cannot be sent if a connection cannot be made to a remote host.

In tuple spaces and MELON, message persistence is inherent in the paradigms. In both paradigms, exchange of messages is achieved by storing the messages in a shared storage space. Any amount of time may elapse between the storage of a message and its retrieval. This allows reliable communication even in the face of prolonged disconnections and is the reason we have chosen it for MELON.

#### 3.2.1.2 Addressing and Discovery

All of the paradigms discussed here provide indirect addressing of resources separated from the physical machines. Publish/subscribe uses topics or content to deliver message to subscribers, RPC uses class and method names, and tuple spaces and MELON retrieve messages by matching content to templates. While all three traditional paradigms originally relied on centralized services (brokers from publish/subscribe, service directories for RPC, and a centralized database for tuple spaces), simple distributed versions may be implemented by having each node act as part of a distributed service.

#### 3.2.1.3 Flexible Communication

A general purpose communication paradigm for MANET applications should have the flexibility to support both unicast and multicast communication.

Publications in publish/subscribe are inherently multicast, since any number of nodes can subscribe. Unicast communication is much less comfortable in publish/subscribe, as it involves negotiating which topics should be used to identify which nodes. Publish/subscribe also does not provide any mechanism for ensuring or even acknowledging message delivery to any given subscriber, especially since publishers and subscribers are intended to be unaware of each other.

Since RPC mimics local method calls, it is natural that RPC is best suited for unicast communication, in which the message is the argument to the method and the return value is the response from the remote host. Assuming multicast RPC functions in the same manner, then a multicast RPC invocation would expect multiple return values, one from each remote host. In a MANET, it is likely not every remote host would reliably return a response, further complicating the semantics. A typical RPC invocation would block waiting for a response, but it is not practical to wait for all responses to a multicast RPC invocation when some responses may never be received. The use of futures or asynchronous callbacks can improve the situation, but causes semantics to differ even more from unicast RPC.

Tuple spaces are naturally multicast, since any number of nodes may read a given tuple. Unicast communication can be achieved by using a field in the tuple as the recipient's address. The recipient then performs in operations on tuples with their address in order to receive the tuples.

Networked applications also commonly require private, unicast communication. For example: SMS services, direct messaging in social networks, or communication of sensitive data. For our purposes, private communication is the exchange of messages between two parties which cannot be disrupted or eavesdropped upon by a third party from within the context of the paradigm itself. In other words, concerns such as encrypting data or sniffing network traffic would be outside the paradigm context.

RPC is unicast by default and there is no method in the paradigm for eavesdropping or disrupting RPC between two nodes. However, RPC has a different complication: remote hosts are generally identified by their exposed methods and there is no mechanism for attaching identity to the hosts. RPC will connect to any remote method with the expected API. So while private communication is the default in RPC, there is an addressing issue which makes it complicated to communicate with a specific recipient.

Communication in publish/subscribe is public and multicast by nature. Any subscriber can subscribe to any set of publications, making it simple to eavesdrop on communications. Bidirectional communication is also difficult in publish/-subscribe, since there is no information attached to a publication indicating the identity of the publisher. This is by design, but it complicates situations in which two hosts need to dialog.

In tuple spaces, tuples are public and available to any recipient. Not only can any node read any communications without detection, any node can also disrupt communications by removing tuples intended for a specific recipient.

#### 3.2.1.4 Multiple Read Problem

The multiple read problem [RW96] is specific to tuple spaces: in a situation where the tuple space contains many tuples of interest, how do multiple readers read *all* relevant tuples? In tuple spaces, the non-destructive **rd** operation returns a copy of a matching tuple, but it may return the same tuple any number of times since the tuple is chosen nondeterministically between all matching tuples. In many tuple space implementations, this occurs because the tuples are stored sequentially and so the first matching tuple is always the same [Da12].

One solution is to use a single tuple as a mutex, lock the tuple space, remove all matching tuples with **in**, then replace them in the tuple space. However, this ruins any concurrency the tuple space could have had with multiple readers.

Another solution is to provide a bulk **rd** operation to return all matching tuples. However, once a “snapshot” of the tuple space has been taken with a bulk **rd**, new matching tuples may be introduced. A second bulk **rd** would return both the old (already seen) tuples and the new tuples. A similar suggestion from [ER01] is to remove all matching tuples inside a transaction, then abort the transaction in order to actually leave the tuple space unmodified. Unfortunately, this again leaves the problem of separating new tuples from previously-read tuples.

### 3.2.2 Paradigm Implementation

#### 3.2.2.1 Publish/Subscribe

As stated in Section 2.3.1, it is not reasonable to expect any nodes to be reliable enough to serve as message brokers. Therefore, our publish/subscribe implementation assumes each node can serve as its own broker, which is not uncommon in MANET publish/subscribe systems [Den09, Cer08]. Subscription requests are broadcast to all available nodes, which maintain lists of subscribers corresponding to a particular topic. For simplicity, topics are specified as simple strings in a flat address space. When an application publishes a message, it sends a copy of the message to all subscribers to the specified message topic. As is common in distributed publish/subscribe systems [Eug03], published messages are not persistent.

#### 3.2.2.2 RPC

Our RPC implementation uses a simple reflection-based mechanism for invoking methods on remote objects. An application may enable remote availability for any Java object. Remote nodes can then search for an object by its class. When found, the application is given a handle to that object which the application can use to call a generic *invoke()* method with the desired method name and parameters. The RPC library handles communication with the remote object and returns the resulting value from the method. This implementation avoids requiring any method stubs or compile-time knowledge of remote objects or method names.

The library provides both synchronous and asynchronous remote invocations. Synchronous invocations will block until a remote object of the expected type is found and a return value is received. Asynchronous calls register a callback to handle the return value when it arrives.

Our implementation also supports group communication. Group invocations attempt to invoke a given method on all known remote objects of the specified class. This must be done asynchronously, since multiple return values must be handled and it is not possible to know how many hosts will respond. The registered callback will be invoked each time a return value is received.

#### 3.2.2.3 Tuple Space

Our tuple space implementation is largely modeled on LIME [Mur06] and uses the same local tuple space library called LighTS [Bal07]. This library provides storage of local tuples and matching of templates against the local tuple space. This allowed us to implement the communication features separately.

While the tuples are logically located in a shared tuple space, they are actually stored locally. For example, an *out()* operation does not actually involve any communication (unless there are existing requests for the output tuple). Operations on the tuple space, however, operate across the entire shared tuple space. When a *rd()* or *in()* is requested, a search is first performed on the local tuple space. If the request can not be completed locally, a request is sent to all known remote tuple spaces. The remote nodes then return a message indicating how many matching tuples they contain. The requesting node then chooses from the nodes with existing matches and requests the matching tuple itself.

Requests which do not match any tuples are handled differently depending on whether the request is blocking or non-blocking. If a blocking request cannot be fulfilled, the request is stored and a reply will be sent if any future tuples match the request. A non-blocking request, on the other hand, will immediately return a message indicating zero matches.

In our implementation, blocking requests will block the requesting application until the request can be filled. If no matching tuples exist at the time of the original request, the request will be periodically repeated until it is met. Non-blocking requests require a callback to be registered, which will be called when a matching tuple is received.

We also provide a *reaction* mechanism [Mur06]. An application may register a tuple template and a callback. The callback will be invoked when a matching tuple is added to the tuple space. This is equivalent to either periodically using a non-blocking request or making a blocking request in a separate thread, but is provided as a convenience.

An application may also perform group requests. These are always asynchronous, due to the possibility of multiple matching tuples, but can still be considered blocking or non-blocking. A non-blocking group request will not be saved on remote nodes to be served later, while a blocking request will be.

### 3.2.3 Experimental Results

In the following sections, we present measurements of message delay and message delivery reliability for unicast and group communication, as well as for a non-trivial whiteboard application. We also examine the message overhead and the influence of routing algorithms. These experiments demonstrate the impact of the wireless network and mobility at the application level.

We compared application-level metrics using unicast and group communication in three network scenarios which are used throughout the experiments: a single hop, static network; a multi-hop, static network; and a fully mobile network. Each node in the emulated network is equipped with an 802.11b wireless interface. The two-ray model is used for path loss. Based on preliminary results, we used DSR [JM96] as the routing protocol for the static scenarios and AODV [PR99] for the mobile scenario.

The mobile scenario uses random waypoint mobility with a pause time of $30s$ and maximum speed of 1 meter/second, representing pedestrians carrying handheld devices. The nodes move within a $1500m$ x $1500m$ indoor space where transmission range is limited to 50m. To avoid network segmentation, the scenario ensures there are always possible routes between any two nodes by having four fixed nodes. However, the remainder of the nodes are highly mobile and routes between nodes change frequently. .The mobility pattern in each experiment is identical.

The emulation environment is provided by EXata [Net08], a network emulator which allows actual applications to run on an emulated wireless network in real time. EXata provides a high fidelity emulation of the entire network stack and detailed simulation of the wireless channel. This provides the realistic environment required for accurate assessment of the paradigms while also facilitating repeatability and fairness [TMB01, VB04]. Rather than comparing the utility and performance of the paradigms theoretically, the emulation approach allows them to be evaluated within actual applications. This more closely reflects their eventual purpose: the development of applications which will execute on a MANET.

The first application used for these experiments is a simple client-server application which can send messages between hosts. This provides a baseline for the performance results and allows us to easily test performance with varying message sizes and frequency. The second application is a shared whiteboard. Collaborative applications are often cited as use cases for MANETs and the shared whiteboard is a common example [Lie09, Bad08, Leg05, SL04]. This provides a non-trivial,

Table 3.2: Message Sequence Overview

| Paradigm | Sender | Receiver | Size (bytes) | Overhead (bytes) |
|---|---|---|---|---|
| Publish/Subscribe | | | | |
| | | Subscribe | 175 | |
| | Publish | | 1182 | |
| | | *Total* | 1357 | 357 |
| RPC | | | | |
| | Search | | 146 | |
| | | Search Reply | 187 | |
| | Invoke | | 1238 | |
| | | Return Value | 152 | |
| | | *Total* | 1571 | 571 |
| Tuple Space | | | | |
| | | Search | 608 | |
| | Search Reply | | 133 | |
| | | Tuple Request | 588 | |
| | Tuple Reply | | 1586 | |
| | | *Total* | 2915 | 1915 |

realistic test case for each of the three communication paradigms.

Each application has three functionally equivalent implementations, one for each of the communication paradigms.

#### 3.2.3.1 Unicast Communication

##### Message Overhead

**Application Overhead** The first step of our experimental evaluation of these three paradigms is discovering the basic cost of communication. Table 3.2

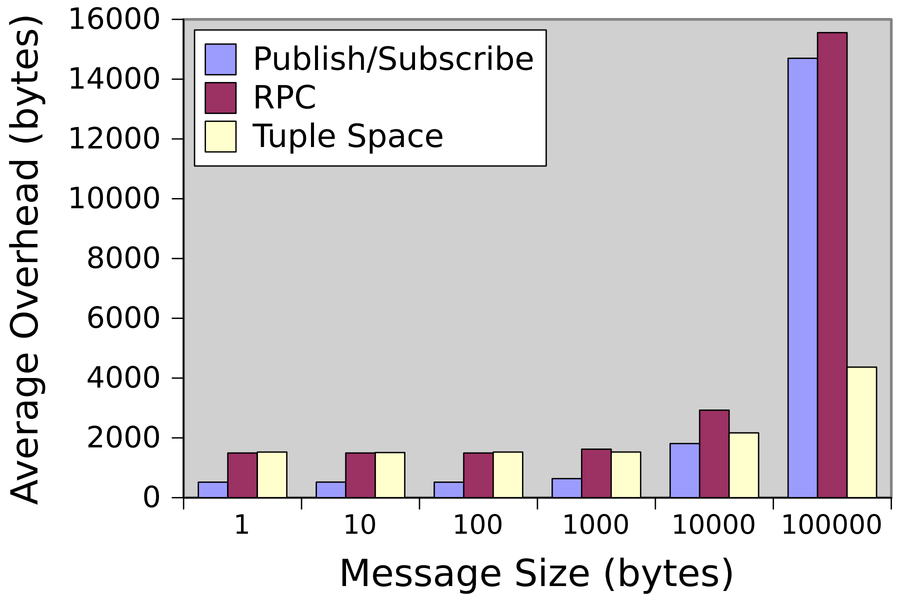


Figure 3.8: Bytes per Message

provides an overview of the sequence of messages involved when using each of the communication paradigms in the simple case of a single sender and a single receiver sending a 1KB payload. The total size includes the 1KB payload. Publish/subscribe requires only two messages to be sent: one to subscribe to a topic and one to publish. Since publish/subscribe only needs to add a string indicating a topic, there is very little overhead added to the original message.

RPC first sends out a query to find the desired remote object. Once found, it sends a second message to invoke the method and transfer any arguments. The final message in the sequence is the return value from the method, which is dependent on the size of the return value.

Tuple spaces require the same number of messages as RPC, but the overhead is 2.3 times higher. Except for the search reply messages, all messages include a tuple object, making them larger than the simple messages exchanged in RPC.

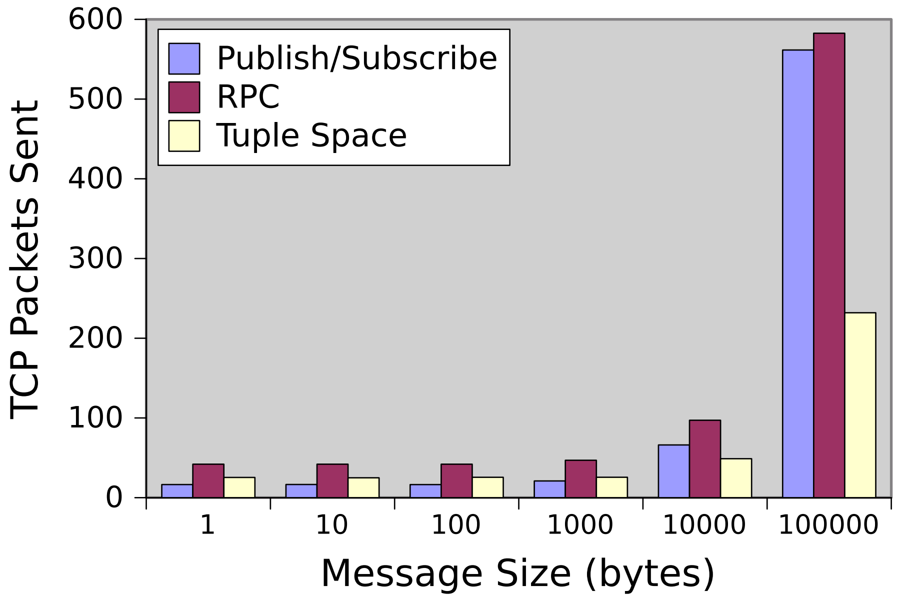


Figure 3.9: TCP Packets per Message

**Network Overhead** While Table 3.2 indicated the overhead added at the application layer, Figure 3.8 shows the average amount of TCP traffic which is sent over the network for a single message, calculated as *bytes sent - message size*. These results use the single hop static scenario and are averaged from 50 messages.

The results are fairly constant until the packet size is exceeded. There is some increase at 10KB, and a dramatic increase at 100KB. Figure 3.9 shows the same data in terms of TCP packets and indicates the cause of the sharp increase in traffic at 100KB is the result of packet fragmentation.

Despite having large message sizes, tuple spaces have much lower overhead in terms of TCP traffic. This difference arises from a side issue related to TCP send window sizes. For tuple spaces, where the receiver initiates the connection, the TCP send window size grows to accommodate larger packet sizes. With RPC and publish/subscribe, the send window size remains constant, causing the large messages to be split into many more packets. For RPC and publish/subscribe, the sender initiates the TCP session, while in tuple spaces it is the receiver (of the tuple) which initiates the TCP session.

**Message Reliability** How reliably a communication paradigm handles message delivery has a direct impact on the application layer. The more reliable the communication paradigm, the less responsible the application is for handling lost messages. We measured reliability in terms of message delivery. In the single hop scenario, all paradigms achieved 100% delivery and figures 3.12 and 3.12 indicate nearly perfect message delivery for all the paradigms in the unicast scenario. Publish/subscribe performed the worst and still only lost 4 messages. However, this is not unexpected, since publish/subscribe sends out publications immediately, whether or not any subscribers are present.

**Message Delay** Message delay is another important application-level metric, as it determines how quickly information is transferred and the freshness of the application's information. Figures 3.15(a), 3.15(c), and 3.15(e) show delay in terms of round trip times for each paradigm in a single hop scenario. The majority of the messages in each paradigm are under the $200ms$ mark, with just a few wayward messages taking longer. Even for tuple spaces, 80% of the messages take less than $400ms$ to complete their round trip. However, some messages take much longer, up to $8s$. For tuple spaces, this is partially due to the complexity and overhead of the messages required to perform the round trip message delivery.

However, the time delay for tuple spaces in the single hop scenario is also related to the pull (rather than push) nature of the paradigm. A tuple is timestamped when it is output, but the tuple is not actually sent to the receiver until the receiver requests it. The same situation happens on the return trip, when the tuple must be pulled back to the original sender. Any delays in this process cause the round trip time to increase.

On the other hand, publish/subscribe messages are sent out almost immediately after being timestamped. Nearly all the delay is caused by the network itself. RPC has more potential for delays since it must find the remote method before invoking it. However, the return message can reuse the existing TCP connection, which appears to provide an advantage over tuple spaces.

The multi-hop scenario introduces more message latency, as seen in Figures 3.15(b), 3.15(d), and 3.15(f). Again, most messages complete the round trip very quickly ($<300ms$), but the maximum times for publish/subscribe and RPC increased from $383ms$ and $110ms$ to $1537ms$ and $902ms$, respectively. Not only does it take longer due to the packets needing to traverse multiple hops, there is also delay introduced by the time to find routes. In the single hop scenario, routes are set up at the beginning of the scenario and there is virtually zero routing activity after that. On the other hand, the multi-hop scenario performs routing updates throughout the run time of the scenario.

The mobile scenario introduces even greater delays. Routes are changing frequently and may be several hops long. While the publish/subscribe and RPC results are clustered around $100ms$ and remain under $500ms$, the tuple space values are considerably higher with a median at $256ms$ and a high of nearly $20s$. This is again due to the pull nature of tuple spaces and the overhead seen in Section 3.2.3.1.

#### 3.2.3.2 Group Communication

Group or multicast communication is a useful but more complex part of MANETs, where information and resources are often disseminated in a peer-to-peer manner. Group communication differs significantly from unicast communication. Given the mobile characteristics and decentralized nature of MANETs, a group's membership may be in constant flux, so it is unlikely a sender has perfect knowledge of the members of the group. The time difference between replies from members of the group may vary greatly, and the initiating node cannot know how many replies to expect.

We have investigated how well each paradigm handled group communication by again evaluating message delay and message delivery reliability, but with multiple receivers.

**Round Trip Application** The method for achieving group communication is slightly different for each paradigm. In publish/subscribe, there are two topics: one for outgoing messages and one for incoming messages. The sending node publishes a message containing a timestamp to the outgoing topic. When an outgoing message is received, the receiving node republishes the message to the incoming topic. The round trip time is then calculated when the sending node receives the incoming copy of the original message.

In RPC, a group method call is asynchronously invoked with a timestamp as the parameter. A group invocation will attempt to invoke all available copies of the remote method and register a callback to handle the return values. In this case, that callback will receive the original timestamp and calculate the round trip time from it.

Since tuple spaces only support pull operations, the situation is inverted. Receivers request an outgoing tuple, which will contain a timestamp. The sender outputs a proper tuple with a timestamp, which is then sent to the requesting receivers. The sender then requests a reply tuple and registers a callback to handle the tuple when it arrives. Upon receiving the tuple with a timestamp, the receiver will output a reply tuple containing the same timestamp. This tuple will be sent back to the original sender, which can then compute the round trip time. This is illustrated in Figure 3.14.

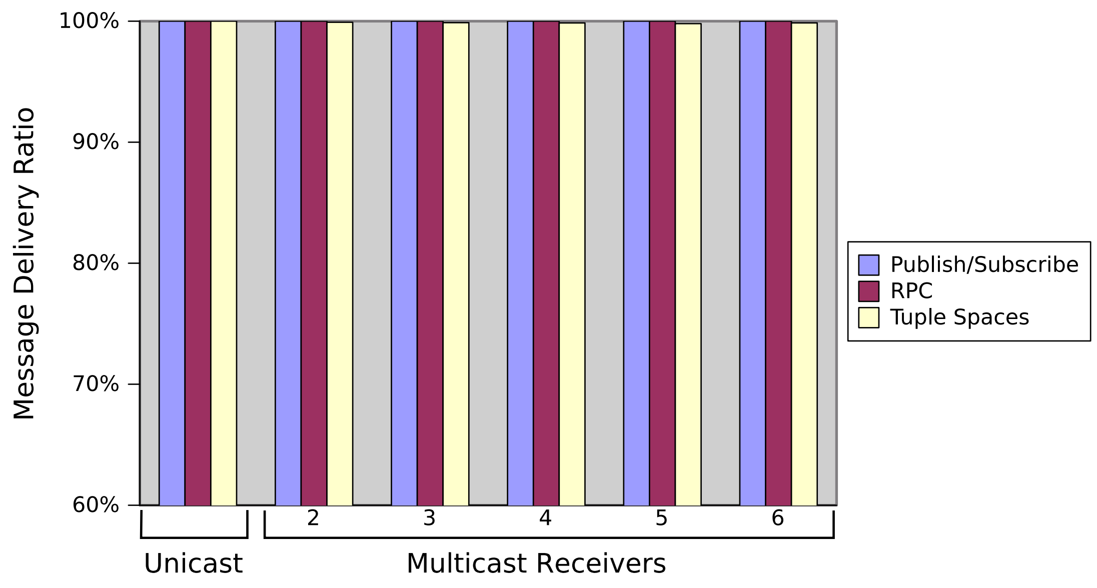


Figure 3.10: Message Delivery - Single Hop

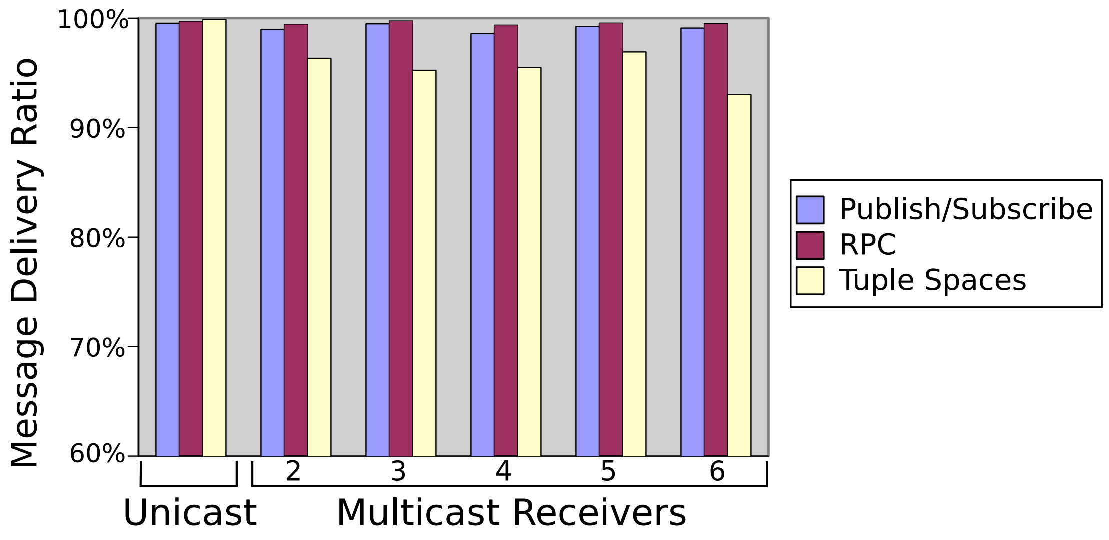


Figure 3.11: Message Delivery - Multihop

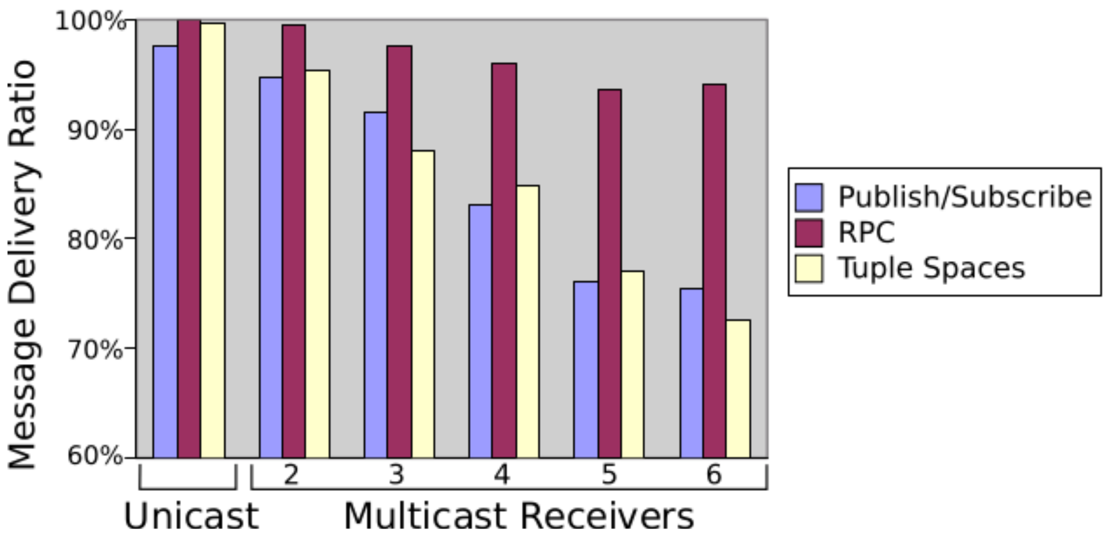


Figure 3.12: Message Delivery - Mobile

**Message Reliability** With this application, message reliability refers to messages which make the circuit from the sender to the receiver and back to the sender. This is useful, for example, in situations where a sink node aggregates information from other nodes.

Figures 3.11 and 3.12 show the percentage of messages successfully completing the round trip. The single hop scenario is not shown, as all paradigms achieved $>$ 99% reliability in that scenario. In the multi-hop scenario, there are more losses even without mobility, but there is no significant trend as the number of receivers increases.

RPC has a slight advantage with this metric, as it will wait until at least one receiver is available. Publish/subscribe and tuple space will send out messages whether or not any receivers are available at the time. However, none of the communication paradigms will retry a message which is lost in transit. A message lost anywhere in the circuit causes the entire attempt to be reported as a failure. For example, if RPC is able to connect to a remote method and invoke it, but never receives a return message, it will not attempt to invoke the method again. Since these results require a message to complete the round trip circuit, there are

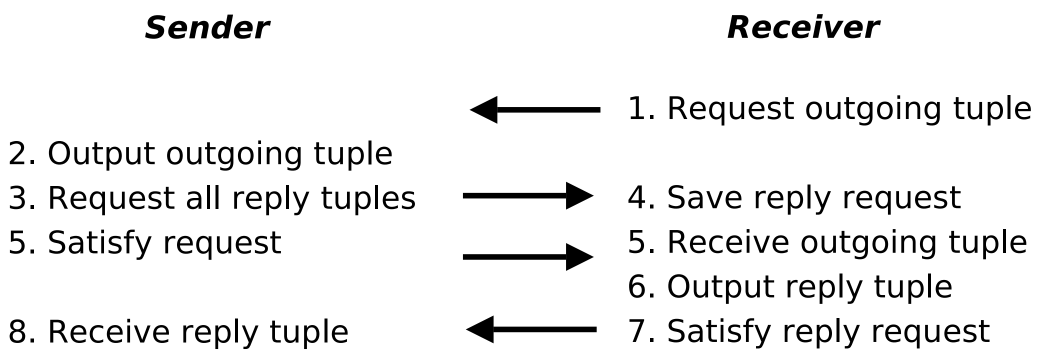


Figure 3.13: Tuple Space Ideal Scenario

multiple opportunities for message delivery to fail.

This contributes to tuple spaces showing the lowest delivery ratio (93%) in the multi-hop scenario and a low delivery ratio (72.6%) in the mobile scenario. While tuple spaces can easily handle the delivery of the outgoing tuple, it is more difficult to guarantee the return of the reply tuple. If a node is not available to receive the request broadcast for a reply tuple, then the reply will never be sent even if the original outgoing tuple is received.

Figure 3.14 illustrates why this is the case. The interaction in the ideal scenario assumes the sender and receiver are present for the entire interaction. In the second scenario, however, the receiver moves away from the sender after sending the initial request for an outgoing tuple. In step 3, the sender sends the reply request, but it cannot be delivered. When the receiver returns, it repeats its request for an outgoing tuple, because it is a blocking request which has not yet been satisfied. The sender satisfies the request, but the reply is never sent since the receiver never receives the reply request.

The solution to this situation would be to repeat the request for the reply tuples. However, the reply request is a group request. Retries are problematic for group requests, because it is unclear when a group request has been completely

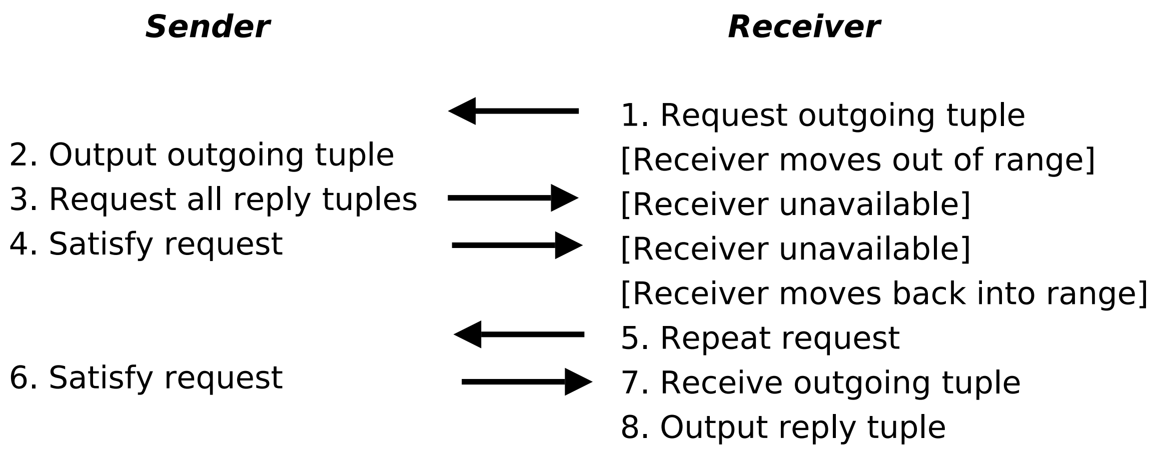


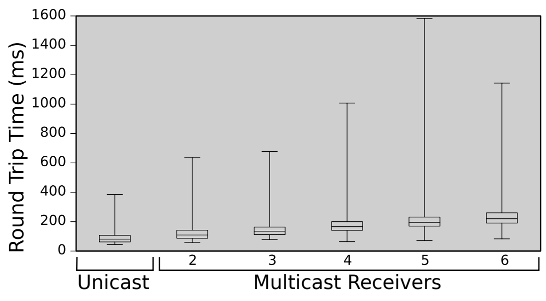
Figure 3.14: Tuple Space Failure Scenario

satisfied. While a normal *rd() or in()* operation is satisfied by a single tuple, there is no upper bound on how many tuples may be available to satisfy a group request, so the requester cannot know when to cease retrying. The LIME [Mur06] project defines a group request as non-blocking and only operating on the current state of the tuple space. This solution would fail in *both* exchanges shown in Figure 3.14, as the reply request may be received before the original request is satisfied. Therefore, we compromised by using blocking group requests, but without retries.

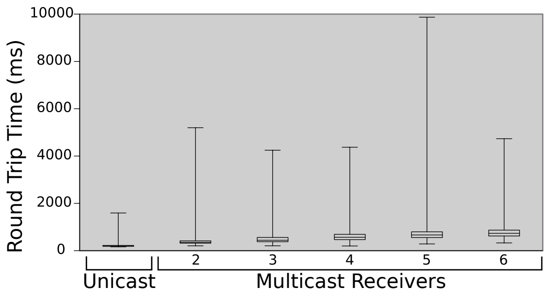
**Message Delay** We again consider round trip time for each of the paradigms, but this time with an increasing number of receivers. Figures 3.15(a) - 3.15(i) show the results for each paradigm and scenario.

For the single hop and multi-hop scenarios, where there is no mobility, the majority of the round trip times are fairly fast. The bottom 75% of the messages have very similar results, while the top 25% varies much more. This indicates that an application can expect most messages to be delivered quickly or not at all, but about a quarter of the messages may arrive up to minutes later.

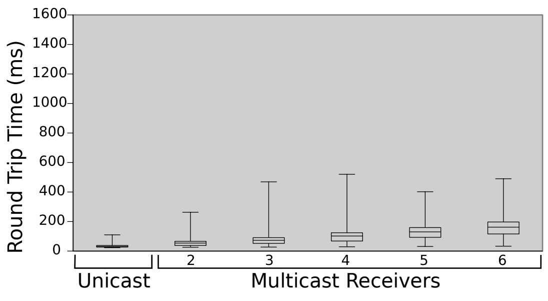
The median delay does increase as receivers are added, especially in the mobile scenario. In the static scenario, the median delay publish/subscribe increased



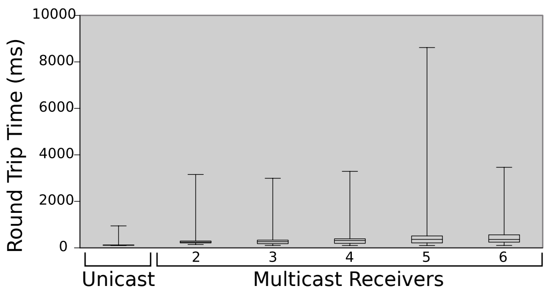
(a) Publish/Subscribe - Single Hop



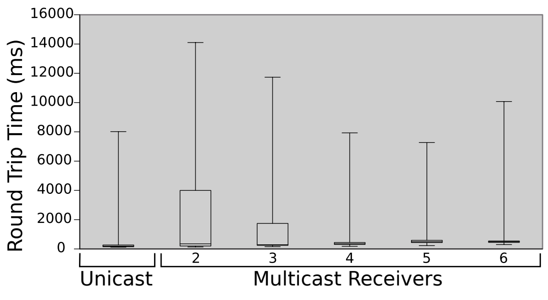
(b) Publish/Subscribe - Multi-hop



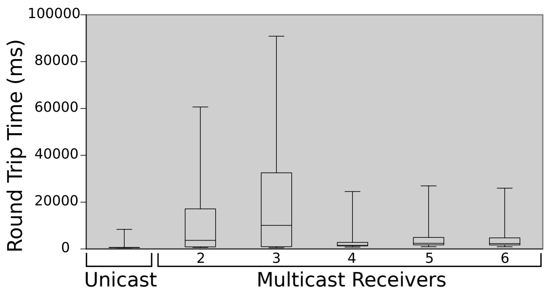
(c) RPC - Single Hop



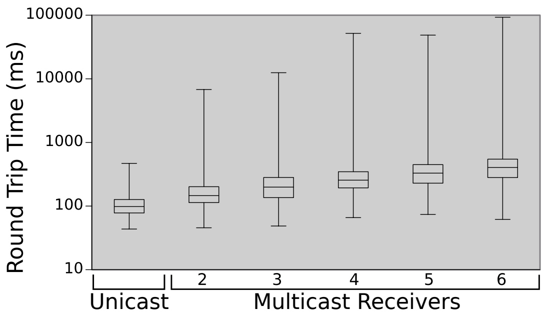
(d) RPC - Multi-hop



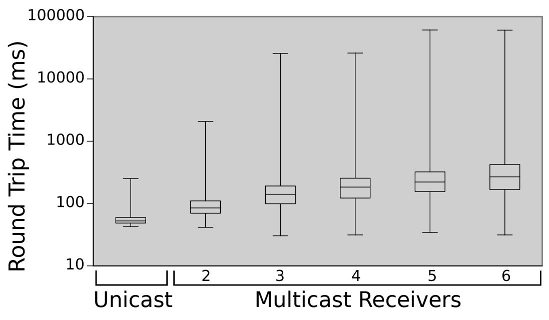
(e) Tuple Space - Single Hop



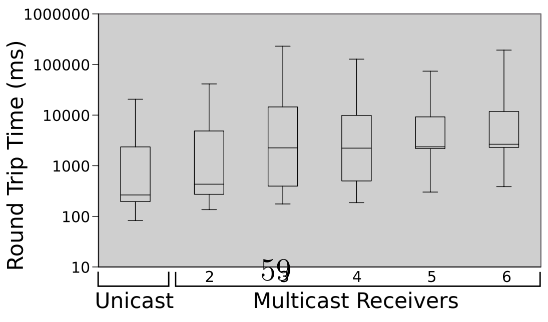
(f) Tuple Space - Multi-hop



(g) Publish/Subscribe - Mobile



(h) RPC - Mobile



(i) Tuple Space - Mobile

Figure 3.15: Round Trip Times

121$ms$ from two receivers to six receivers. RPC increased 147$ms$, and tuple spaces increased 140$ms$. For the mobile scenario, the median times for publish/subscribe increased 255$ms$, RPC increased 237$ms$, and tuple spaces increased by 2035$ms$. The maximum delay values varied much less predictably. For tuple spaces, the static scenarios have unusually long delays with two and three receivers. In the static scenarios, the first three receivers are located in close proximity. One node would dominate the channel for several seconds before relinquishing it. Once again, this shows how influential the wireless channel is on the performance and behavior of applications in MANETs.

The median and maximum tuple space results are much longer than the other two paradigms. The median delay for tuple spaces ranges from twice as much as publish/subscribe in the single hop scenario up to 6 times as high in the mobile scenario. For publish/subscribe and RPC, the majority of delays can only be caused by the network, since they do not attempt to retransmit messages. Tuple spaces, on the other hand, can have very large delays due to the paradigm itself. Note that steps 1 and 2 in Figure 3.14 can be reversed: the outgoing tuple can be timestamped before it is even requested by the receiver.

If a receiver is "behind" it may spend time receiving older tuples before the newest tuple is requested. This causes the round trip times to increase while only improving one-way message delivery. While it does improve one-way message delivery, it does not improve the round trip message delivery ratio, due to the reasons discussed in Section 3.2.3.2.

#### 3.2.3.3 Shared Whiteboard Application

When testing the whiteboard application, we considered the metrics which a user might care about at the application level: how reliably and quickly users receive updates. This models a classroom or presentation setting where the instructor or

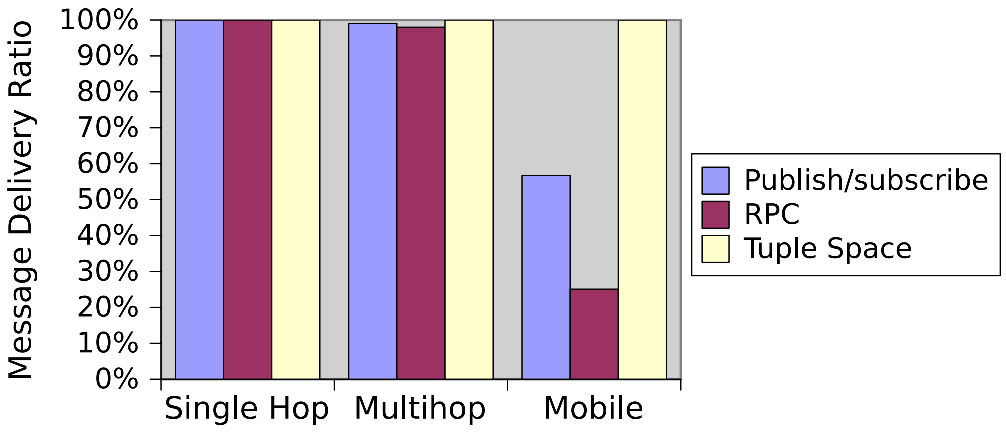


Figure 3.16: Whiteboard Message Delivery (DSR)

presenter is the only one writing on the shared whiteboard, but the contents of the whiteboard are shared out to the class or audience. As in the group communication experiments, in each scenario Node 1 is the sender, with the other nodes marked with triangles as the receivers. In the results below, a single user is updating the whiteboard and the updates are propagated to 6 receivers. We used traffic traces from Coccinella[1]to ensure our implementation accurately represented a typical whiteboard application. For these experiments, 250 whiteboard update messages of varying sizes were sent out over a 10 minute period at varying intervals.

Furthermore, we tested the whiteboard application under the two different routing protocols we have been using, AODV and DSR. This is not meant to be an exhaustive comparison of the routing protocols themselves, but is intended to show how the choice in routing protocols might affect the performance of the communication models in a nontrivial application.

**Message Reliability** Unlike the previous results, these represent one-way communication from the whiteboard user to the receivers. Message reliability determines how accurately the receivers' views reflect the state of the shared white-

[1]http://thecoccinella.org/

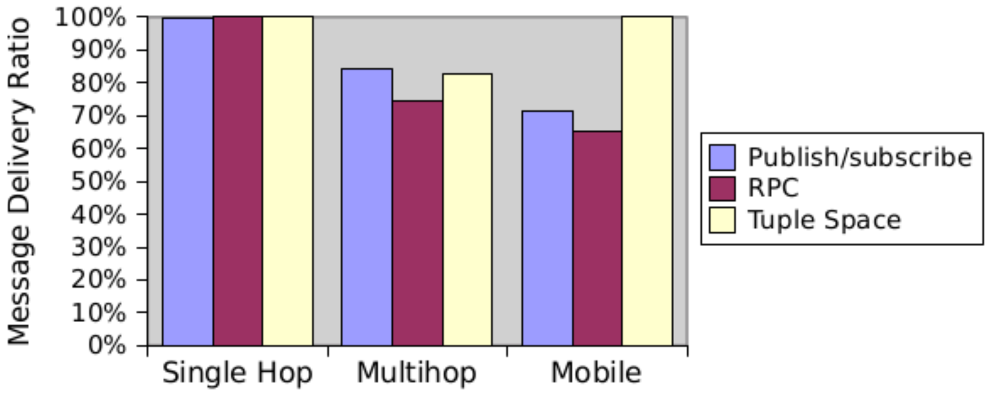


Figure 3.17: Whiteboard Message Delivery (AODV)

board.

Figures 3.16 and 3.17 show the percentage of whiteboard messages delivered for DSR and AODV, respectively. As before, the results are nearly 100% for all paradigms and both protocols in the single hop network. AODV performs poorly on the multi-hop scenario, while DSR achieves nearly 100% delivery for all paradigms. On the other hand, DSR performs much worse in the mobile scenario, with the delivery ratio for RPC only reaching 25%.

The reliability of tuple spaces is considerably better in these experiments than in the round trip scenario, with 100% delivery in all but the AODV multi-hop scenario. The difference between these results and Section 3.2.3.2 is the lack of a return message. Each receiver is responsible for requesting the whiteboard updates, so the blocking request will be retried until the tuples are received. The only exception is the multi-hop scenario with AODV, in which all three paradigms perform much worse. Since all three paradigms are affected equally, these results must be directly due to the behavior of AODV in this scenario. Investigation of this phenomenon is outside the scope of this dissertation.

While the choice of routing protocol can have a significant effect on the paradigm performance, it appears to affect each paradigm in the same manner. In other words, neither routing protocol improves the performance of one paradigm while

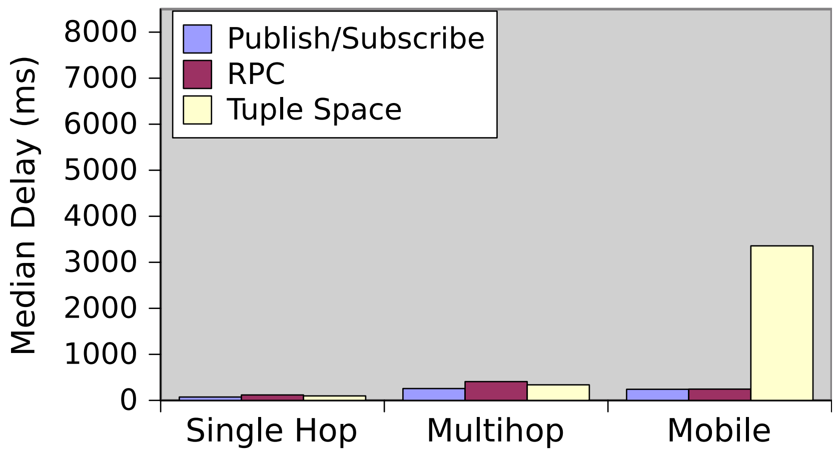


Figure 3.18: Whiteboard Message Delay (DSR)

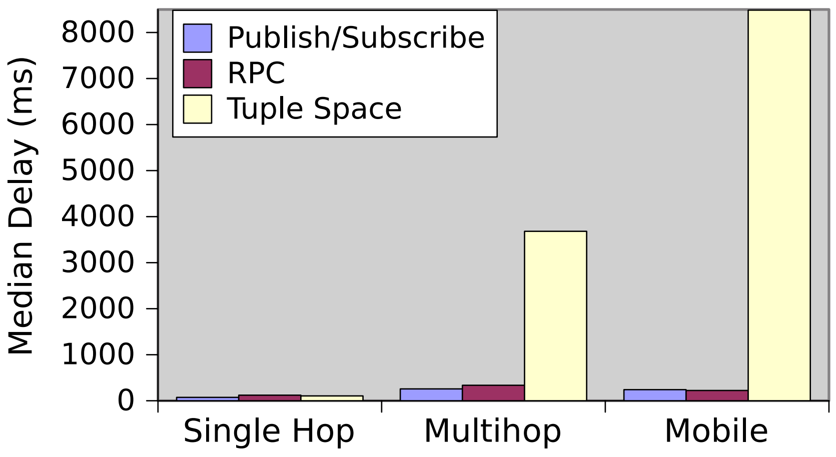


Figure 3.19: Whiteboard Message Delay (ADOV)

causing a different paradigm to perform worse.

**Message Delay** Message delay is measured as the time from when a whiteboard update is sent by the application until it is delivered to the receiver's whiteboard. Update delays are very noticeable in a shared whiteboard application, so the delay time should be minimized.

Figure 3.18 shows the results when using DSR and Figure 3.19 shows the AODV results. Not unexpectedly, tuple spaces have the highest median latencies of 8,486$ms$ with AODV and 3,357$ms$ with DSR. For publish/subscribe and RPC, the median delay remained under 400$ms$. As noted previously, there is an obvious direct relationship between reliability and the message delay time. If a message cannot be delivered immediately, the communication paradigm can either drop the message or retry later. Dropping the message decreases reliability, but the median message delay will be low. However, attempting to deliver the message later increases both reliability and message delay.

With DSR, tuple spaces report a nearly 100% delivery ratio in every scenario, yet the delay times are <400$ms$ in the static scenarios. In contrast, AODV causes long delays for tuple spaces in both the multi-hop and mobile scenarios. Since tuple spaces will repeatedly attempt to deliver messages, retries are expected to contribute to the majority of the delays. This is supported by the long delay times experienced by tuple spaces with AODV in the multi-hop scenario. However, in the mobile scenario tuple spaces achieve 100% delivery with AODV and DSR, but the median delay with DSR is less than half as with AODV.

From the mobile reliability results, we can infer that DSR does not maintain viable routes, because the results of publish/subscribe and RPC are poor. However, the delay results suggest DSR is faster than AODV at finding new routes as they become available.

#### 3.2.3.4 Routing Overhead

Above, we investigated application and transport layer overhead for each paradigm. In this section, we look at the routing overhead incurred for each paradigm when used in the whiteboard application. Ideally, the amount of routing overhead should be as small as possible in order to conserve bandwidth.

Figures 3.20, 3.21, and 3.22 show the number of routing packets generated in each scenario. For the single hop and multihop scenarios, DSR has much lower overhead. Once mobility is introduced, however, the routing packets from DSR

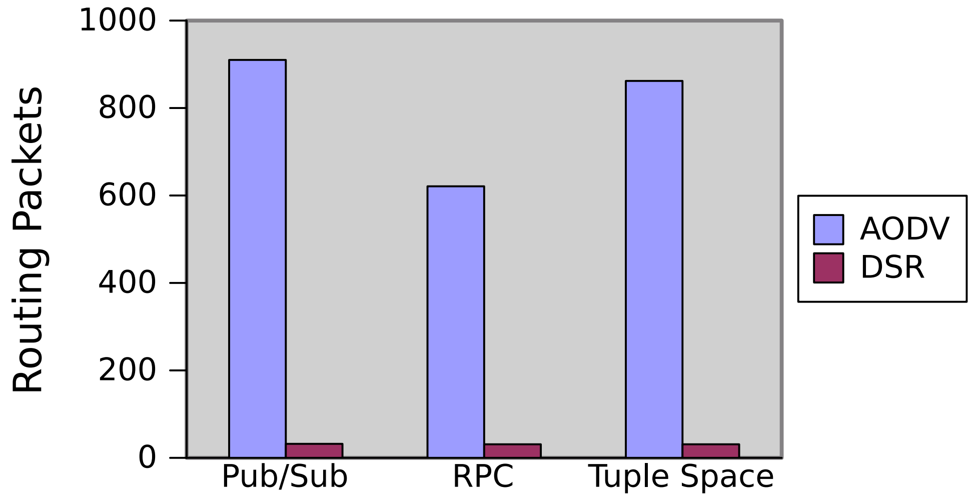


Figure 3.20: Whiteboard Routing Overhead - Single Hop

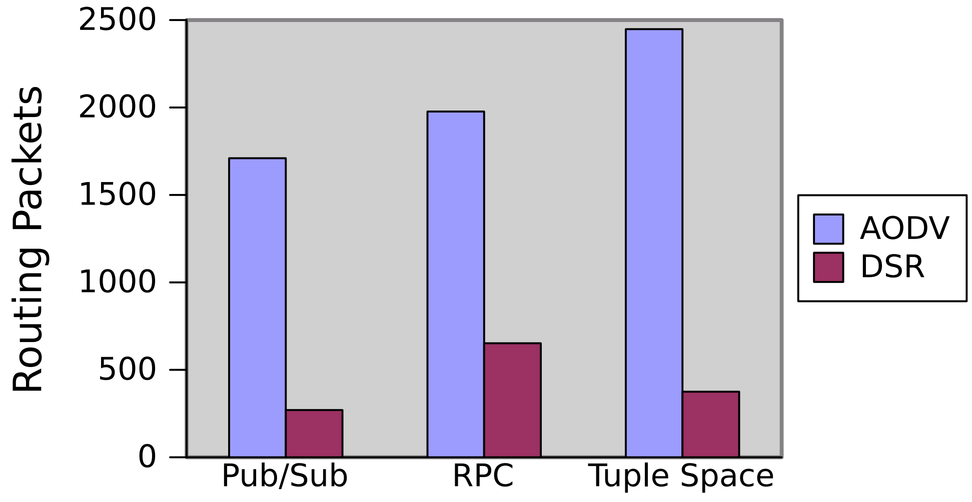


Figure 3.21: Whiteboard Routing Overhead - Multihop

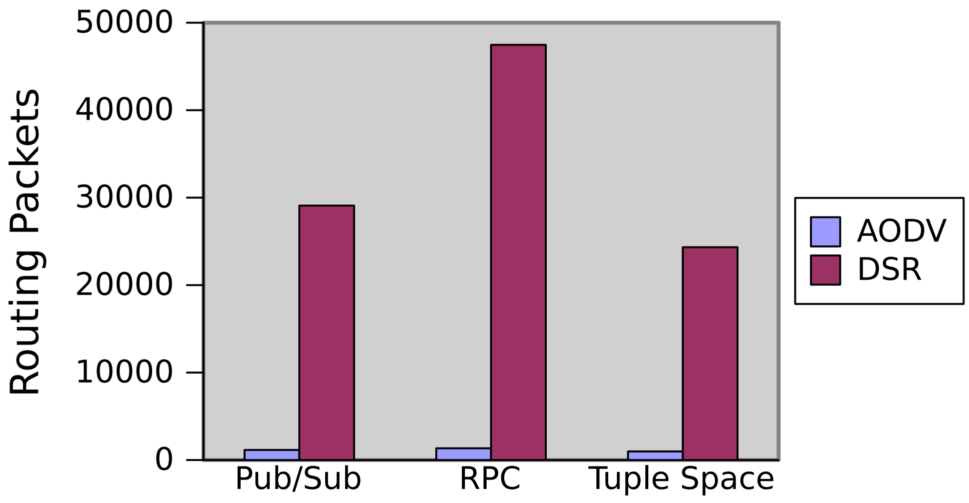


Figure 3.22: Whiteboard Routing Overhead - Mobile

dwarf the number used by AODV. This appears to correspond to the low message delivery in Figure 3.16, although the message delay remains low for the messages which are actually delivered.

From these results, no clear correlation can be drawn between the paradigms and the amount of routing overhead. It varies according to both the network scenario and which routing protocol is used.

# CHAPTER 4

# MELON Coordination Model

MELON[1] is a practical approach for distributed communication in MANET applications that provides persistent messages, reliable FIFO-ordered multicast, efficient bulk retrieval, and simple message streaming. MELON is intended to be a general purpose MANET communication paradigm, as opposed to focusing on a specific use case such as context-aware applications, pervasive computing, or sensor networks.

## 4.1 Design Overview

MELON is a pull-based communication model in which applications asynchronously exchange messages through a shared message store. The messages are either stored as read-only or take-only. Read-only messages can only be copied from the message store, not removed, while take-only messages may only be retrieved by removal from the message store. Messages are retrieved by matching them against templates. MELON also provides bulk operations to read or take all matching messages at once.

### 4.1.1 Disconnection Handling

The main difficulty of communication in MANETs is the frequent disconnections caused by the unreliable wireless channel and the very dynamic network topology

[1] Message Exchange Language Over the Network

where nodes may join, leave, and move at any time. As noted in Section 3.2.1.1, message persistence is a useful approach to hiding disconnections from the application. Instead of requiring a reliable connection to the receiver, messages may be delivered whenever the receiver is available. Disconnections that occur between the sending and receiving of a message are hidden from the application.

In order to provide message persistence, the design of MELON is centered around a distributed shared message store. By communicating through a shared message store, the concept of a connection between hosts is eliminated and thus disconnections are no longer an issue at the application layer. A host suddenly leaving the network does not disrupt an application and applications do not need to handle a communication operation returning an error or failing due to intermittent network connectivity or physical wireless interference. The application is effectively insulated from these issues by the nature of the paradigm and the semantics of the operations.

Besides hiding intermittent network issues, message persistence also provides temporal decoupling between hosts, since messages can still be delivered even after prolonged disconnections. MELON is a pull-based paradigm in which the receivers request messages from the message store and senders merely deposit the messages.

Due to the dynamic network topology of MANETs, maintaining any type of logical or overlay network structure becomes challenging, so MELON does not rely on a particular network structure. Discovery of available messages is performed dynamically for each operation. While this does increase the amount of communication required for each operation, it removes the need for global state and allows the network to change at any time.

### 4.1.2 Addressing and Discovery

Since MANETs are self-organized and infrastructureless, it is not feasible to rely on centralized sources of information such as resource directories. Additionally, it is advantageous to avoid tying data to physical locations, but rather to address resources by their content or other labels instead of IP or MAC addresses. This allows resources to migrate and for multiple hosts to service a request instead of being associated with just one.

MELON provides spatial decoupling (where the sender and receiver need not be aware of each other) by matching messages based on content, rather than by a host address or location. The messages themselves may physically reside on any host in the network. The sender of a message is not aware of the receivers' identities nor even how many receivers might read a message. This frees applications from tracking remote addresses or contacting a directory service to find remote resources.

### 4.1.3 Flexible Communication

The shared wireless communication medium in MANETs is well-suited to group or multicast communications. MELON supports multicast communication by allowing any number of receivers to read the same message. MELON also provides bulk receives, which allow applications to efficiently receive multiple messages from multiple hosts in a single operation.

Applications often require point-to-point or unicast communication as well. While unicast communication can be accomplished through by storing regular messages in MELON, this communication can easily be disrupted by a process removing a message intended for a different receiver. Additionally, it is possible to eavesdrop on messages unnoticed by reading a message and not removing it. For applications such as instant messaging, it is important to have private unicast communication. In MELON, messages may be directed to a specific receiver when stored to ensure the messages are only taken by the intended recipient.

### 4.1.4 MELON Features

MELON also includes features uncommon to shared message stores to further simplify application development in MANETs. First, messages are returned in first-in first-out order per host. When a host receives a message request, it returns the oldest matching message in its local storage. In applications where a single host generates the majority of the messages, this eliminates the need to order messages on the receiver side.

Secondly, MELON provides operations to only read messages which were not previously read by the same process. This enables an application to read all matching messages currently in the message store, then read only newly-added messages in subsequent operations. It also prevents an application from reading the same message twice. This approach avoids the "multiple read" problem demonstrated by tuple spaces and discussed in Section 3.2.1.4.

Lastly, MELON differentiates between messages which are meant to persist and be read by many receivers versus messages intended to be removed from the message store. For example, messages in a news feed would have many readers, but the messages themselves should not be removed. On the other hand, a job queue expects each job to be removed by exactly one worker. MELON provides operations to support both of these scenarios.

## 4.2 MELON Operations Overview

Messages can be copied to the shared message store via a **store** or **write** operation. A **store** operation allows the message to later be removed from the storage space. Messages saved with a **write** operation cannot be explicitly removed from the

Table 4.1: Operations Summary

| | Add single message | Retrieve single message | Retrieve many messages |
|---|---|---|---|
| Nondestructive retrieval | **write** | **read** | **read_all** |
| Destructive retrieval | **store** | **take** | **take_all** |

storage space, only copied.

Messages added via **store** may be retrieved by a **take** operation using a message template which specifies the content of the message to be returned. A **take** operation will remove a message with matching content from the message store and return it to the requesting process. **take** operations are atomic: a message may only ever be returned by a single **take** operation.

A **read** operation will also return a message matching a given template, but does not remove the original message from the shared storage. Any number of processes may read the same message. However, repeated applications of a **read** operation in the same process will never return the same message. Only messages stored with **write** can be returned by a **read** operation.

The basic **take** and **read** operations return a single message per invocation. To facilitate the exchange of multiple messages, MELON includes the bulk operations **take_all** and **read_all**. The bulk versions operate the same as the basic operations, except all available matching messages will be returned instead of a single message. For **read_all**, only messages which were not previously returned by a **read** or **read_all** in the same process will be returned.

By default **take**, **take_all**, **read**, and **read_all** will block the process until a matching message is available. MELON also provides non-blocking versions of these operations. The non-blocking operations will return a null value if no matching messages can be found.

When a message is saved with a **store** operation, it may optionally be directed to a specific receiver. In a directed message, the identity of a receiver is included in the message as the addressee. Only the addressee may access a directed message through a **take**.

Due to the limited resources of most devices in a mobile network, storage space in MELON is explicitly bounded. Any message may be garbage collected prior to being removed by a **take** if capacity is reached.

### 4.2.1 Operation Details

Processes in MELON communicate by storing messages to a distributed shared message store and retrieving the messages based on templates. FOr simplicity, we assume messages consist of an ordered list of typed values and optionally an addressee. However, nothing in the paradigm itself limits how messages might be constructed (e.g., they could be an unordered tuple with named values instead).

A message template is similar to a message, except it may contain both values and types. For example, a message containing `[1, "hello"]` could be matched by a template containing `[1, String]` or `[Integer, "hello"]` or `[Integer, String]`. A type will also match any subtypes.

Each operation is implemented as a separate function call. **store** and **write** operations have null return values and return as soon as the saved message is available in the message store. **take** and **read** operations block by default until a matching message is returned, but may be set to non-blocking on a per-call basis.

The **store** operation takes a message as an argument and optionally an address. When called, **store** saves a copy of the message in the message store. Messages saved with **store** may only be retrieved with a **take** or **take_all** operation. If an

| **Operation** | **Return Type** |
|---|---|
| **store**(*message*, *[address]*) | *null* |
| **write**(*message*) | *null* |
| **take**(*template*, *[block = true]*) | *message* or *null* |
| **read**(*template*, *[block = true]*) | *message* or *null* |
| **take_all**(*template*, *[block = true]*) | *array* |
| **read_all**(*template*, *[block = true]*) | *array* |

Table 4.2: MELON Operations

address is provided, then only the host with a matching identity can remove the message. Since storage space is bounded, messages may be automatically garbage collected from the storage space prior to explicit removal by a **take** or **take_all** operation.

The **write** operation also stores a single message in the message store, but the message may only be copied from the storage space with a **read** operation, never explicitly removed. Messages written with the **write** operation may be automatically garbage collected.

A **take** operation requires a message template as the first argument and an optional boolean for the second argument.

The message template is matched against available messages in the message store which were added with a **store** operation. If a matching message is found, it will be removed from the message store and returned.

The block argument, which defaults to true if no argument is given, controls behavior of the operation if no matching message is available. If *block* is true, the operation will wait until a matching message is available, then return it. If *block* is false, the operation will return a null value.

Once a message has been returned by a **take** operation, it is removed from the

Table 4.3: Read from multiple processes

| **Process A** | **Process B** | **Process C** |
|---|---|---|
| `write([1, "hello"])` | `m = read([Integer, String])` | `m = read([Integer, String])` |

message store and may not be returned by a subsequent operation in any process.

The **read** operation accepts the same arguments as **take**. A **read** operation will only return messages stored with a **write** operation which have not already been read by the current process.

If a message matching the given message template is available, it will be copied and returned, but not removed from the message store. Once a message has been returned to a process, the message is considered to have been read by that process and will not be returned by any subsequent read or read_all operations in the same process.

When a matching unread message is not available, behavior of **read** depends on the *block* argument. If the argument is true or unspecified, the operation will block until a matching message is available, then return that message. If the argument is false, the operation will return a null value.

A message may be **read** by any number of processes, but each process may only read each messages at most once.

Table 4.3 illustrates one process writing a single message containing the integer `1` and the string `"hello"`. Processes B and C each perform a **read** operation with the template `[Integer, String]` which matches the message stored by process A. Since **read** does not modify the storage space, the value of *m* for both process B and C will be a copy of the message `[1, "hello"]` from Process A.

The **take_all** operation performs a bulk **take** on the given message template. The return value of **take_all** is an array of matching messages. As with **take**, messages returned by a **take_all** are removed from the shared storage and may not be returned by any subsequent operation in any process. A **take_all** operation will not return a directed message unless the addressee matches the current process. Only messages stored by a **store** operation will be returned by **take_all**.

When there are no matching messages and the value of *block* is *true* or unspecified, the operation will block until at least one matching message is available and then return an array of available messages. If *block* is *false*, **take_all** will return an empty array.

**read_all** performs a bulk read on the given message template and returns an array of matched messages. **read_all** only returns messages which have not been previously returned in the same process by a read or **read_all**. A **read_all** operation will only return messages written by a **write** operation.

When there are no matching messages and the value of *block* is true or unspecified, the operation will block until at least one matching message is available and return an array of available messages. If *block* is false **read_all** will return an empty array.

## 4.3 Message Store Model

MELON's operates through a semi-persistent shared message store. The message store must match message templates to actual messages, provide concurrent access for multiple clients, manage returning messages in per-host FIFO order, implement basic access control as required by MELON operations, and of course be reasonably fast.

MELON's message store is only *semi*-persistent because it does not attempt to reliably retain all stored messages, and because the MELON paradigm explicitly acknowledges space limitations on mobile devices and will perform garbage collection of messages if necessary to store newer messages. Given the turbulent nature of MANETs, it is not practical for the message store to reliably retain all messages. Nodes will unexpectedly disconnect from the network, taking with them any hosted messages. If some global data structure were used for the message store, nodes moving from one network to another would need to explicitly disconnect from one shared message store, then explicitly join the next. This is not a practical constraint with the fluid and unpredictable nature of MANETs.

From the perspective of an application, the message store is a single entity and it is not necessary or possible for an application to determine where a message physically resides. But in reality the message store is distributed across hosts, with each host being responsible for the messages stored by applications running locally on that host. MELON's message store is designed so each host may operate independently and without coordination between hosts. The only necessary communication originates from an application requesting a message to each remote host. Some coordination may be added for more advanced features (in particular, message replication in Section 5.8), but is not necessary to support MELON's operations.

## 4.4 Message Ordering

The order in which messages are received in a communication paradigm can have a large effect on an application. In tuple spaces, for example, messages may be retrieved in any order and when multiple tuples match a template the choice of which tuple to return is explicitly nondeterministic. This puts a burden on the application to maintain desired ordering. In other paradigms, ordering is essentially undefined.

It would be ideal to have messages delivered in the order in which they are sent, but managing this is very difficult in a MANET where the network and hosts are unreliable and communication is expensive. Providing total-ordering or causal-ordering is still difficult for the same reasons: a message is not guaranteed to ever be delivered. It may be lost in transit and the sender may leave or fail before retransmission. For pull-based paradigms such as tuple spaces and MELON, ordering is difficult to define since different messages may be requested by different receivers.

However, it is still very useful to have some ordering for message delivery. Video streaming from a single host needs to be displayed in order on the receivers. Order may be enforced by the receiver via a buffer, but if messages are too far out of order the buffering time would become unacceptable. In general, we noted there are cases in which messages from a single source should be ordered, but have no need to be ordered relative to messages from other sources.

To provide some useful ordering with minimal overhead, messages retrieved from MELON's message store are returned in per-process FIFO ordering as described in [TS02]. Messages sent by the same host are received in the order sent. This does not provide a global ordering, but still relieves applications of some responsibility. One example where this was convenient was logging output from experiments as described in Section 6.4. The coordinator reads messages output from all hosts and records them in file per host. The messages are written in order automatically for each host without the application needing any logic to manage the ordering. The message store is responsible for enforcing this ordering and a simple approach to implementing this functionality is described in 5.4.

# CHAPTER 5

# MELON Implementation

This chapter describes a prototype implementation of MELON we developed in order to validate our design and obtain empirical performance data.[1]

## 5.1 Architecture

The architecture of our MELON implementation illustrated in Figure 5.1 is divided into five parts. The MELON API is the only interface exposed to the application and provides the six operations described above. The MELON API interacts with the distributed message storage through the storage API, which provides the same interface for both local and remote storage. The storage server proves a network interface to a local storage space and accepts connections made through the remote storage stub.

## 5.2 MELON API

The MELON API as provided to the application is very simple. It only provides the six MELON operations, plus the ability to manually specify remote hosts. The API implementation does very little except interact with either local or remote storage. Both local and remote storage offer the same API, so they can be accessed uniformly.

[1]Current source code for the prototype is available at `https://github.com/presidentbeef/melon`

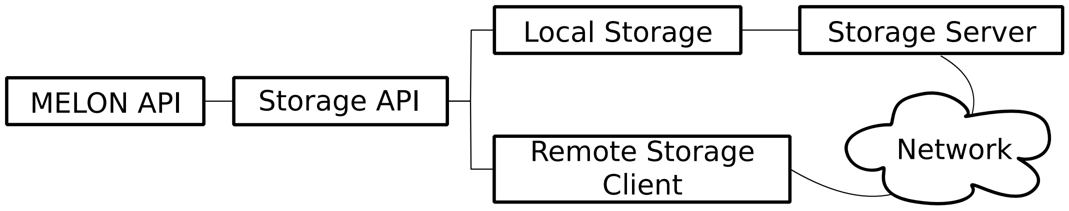


Figure 5.1: Paradigm Architecture

The prototype library tracks the local storage, remote servers, and read messages. **write** and **store** operate on the local storage only. **read** and **take** make no distinction between local and remote storage but simply iterate through the list of stores and invoke operations on them until the operation is satisfied. Each operation accesses the stores in a random order to spread the load and also provide a variety of sources.

## 5.3 Storage API

The storage API is an interface between the MELON API and either local or remote storage. In practice, however, the MELON API calls methods on the local storage directly. The API offers exactly the same six operations as MELON.

Remote storage is accessed through the remote storage client which implements the same API as local storage, except without **write/store** since those are only performed locally. Each remote host is represented with its own remote storage client. Internally, the remote storage clients manage connections with the remote storage servers.

## 5.4 Local Message Store

While applications view the message store as a single entity, it is actually the federation of local message stores hosted by each node running MELON. Each message in the network is by default stored on exactly one node, which is the node on which the application is running which performed the **store** or **write** operation. In other words, storing a message is a local operation. This allows MELON to provide atomicity for message removal with **take** and to ensure per-host FIFO ordering when returning matched messages.

Local storage is implemented simply as two dynamic arrays, one for **write/read** messages and the other for **store/take** messages. For atomic updates, the **write/read** array uses a readers/writer lock[2] to allow multiple **read** operations to access the array in parallel, but locks the array for **write** operations. The **store/take** array does not permit concurrent operations, since both **store** and **take** modify the store. The two arrays may be accessed and modified independently.

Implementation of **store/write** is simple: exclusive access is obtained for the appropriate array and the message is appended to the end.

For **take**, the lock is obtained for the **store/take** array and the message store starts at the oldest message and linearly scans until a matching message is found. When a matching message is found, the message is removed from the array and the elements shift appropriately to fill the gap.

When performing a **read**, a readers lock for the **write/read** array is obtained. The message store starts at the oldest message and linearly searches the array for a matching message. If a message is matched, it must also be checked to not exist in the provided read message set (see Section 5.7 below). If it is in the read message set, the search continues. Otherwise, the matching unread message is returned.

[2]Source code is available at `https://github.com/presidentbeef/rwlock`

In the architecture described here, the local message store is unaware of the location of the requesting or storing client, although it is assumed **store/write** operations are local. The storage server provides an API for remote applications to connect to and query the local storage.

## 5.5 Storage Server

Each local storage is accompanied by a storage server which allows remote hosts to connect and query the local storage. The storage server handles incoming connections, converts queries into calls to the local storage, and converts messages from local storage into responses back to the remote hosts. Each storage server can handle multiple concurrent requests.

## 5.6 Networking

Network communication is handled using ZeroMQ [Hin13], a high performance, high level networking library. For the prototype, the network communication was intentionally kept simple. For example, a **read** request queries remote hosts in a random order and stops when a matching result is returned. For bulk operations, the prototype implementation also queries remote hosts in random order, but continues fetching results until all hosts have either returned a response or a timeout is reached.

While it is possible to improve upon this approach using multicast, it would greatly complicate the implementation by requiring the client to handle multiple asynchronous responses, choose between them, request the actual matching message, and then handle failure scenarios if the matching message cannot be returned. Our approach was to trade off potential performance gains for simplicity.

## 5.7 Read Message Tracking

When a messages is stored, it is given a unique identifier $[P,\ M]$, where $P$ is a globally unique integer identifier for the storing process, and $M$ is an integer identifier for the stored message. Each process maintains an integer ID which is incremented for each store. Messages stored from the same process with sequential **store** or **write** operations will have consecutive $M$ values and share the same $P$ value.

In order to prevent **read** from returning a message more than once in the same process, each process maintains a sparse bit set for each process from which a message has been read. The identifier $[P,\ M]$ is condensed into a single unique integer $Q$ using the "elegant pairing function" [Szu06] shown in Equation 5.1. Since the values of $Q$ will be consecutive integers for all consecutive values of $M < P$, it is helpful to set $P$ to be higher than the number of expected messages. The value $Q$ is then stored in a sparse bit set with a hash table using integer keys and bit field values.

$$f(M,P) = \begin{cases} M^2 + M + P & : M \geq P \\ P^2 + M & : M < P \end{cases} \tag{5.1}$$

The index $i$ in the sparse bit set indicates the range stored in the bit set. If $w$ is the number of bits for each bit set, then each bit field can store up to $w$ values of $n$, where $w \times i \leq n < w \times (i+1)$. A message with ID $n$ will be stored in index $n/w$ by setting the bit at $n \bmod w$ in the bit field to `1`.

If the index value is of size $l$ bits and the bit field contains $w$ bits, then the cost for storing a single value is $l + w$. For storing a set of consecutive values of length $m$, the cost is $\lfloor \frac{m \times l}{w} \rfloor + m$ bits. In other words, the total cost is one bit per message, plus the cost of one index per $w$ messages.

Consecutive messages (from any starting value) are the best-case scenario for

Table 5.1: Sparse bit set example

| Index | Bit Field |
|---|---|
| 0 | 01100001 |
| 4 | 00010000 |
| 15 | 10100100 |

sparse bit sets. In the worst case, the message IDs differ by at least $w$, causing each message to incur a $l + w$ cost for storage and a total cost of $m \times (l + w)$ bits.

Determining if a message $[P, M]$ is in the set is accomplished by first computing $Q$. If there is no key at index $Q/w$, the message has not been read. Otherwise, retrieve the bit field $b$ at index $M/w$. If $b \wedge 2^{M \bmod w} \neq 0$ then the message has been read, otherwise the message is unread.

### 5.7.1 Sparse Bit Set Performance

For matching read-only messages, the read message data structure will need to be fast and small. In this section we are using a sparse bit set implementation developed for MELON[3] based on Ruby hashes and using 64-bit integers. Testing is performed using Ruby 2.1.2.

In Figure 5.2 we compare the speed of the sparse bit set implementation to the standard Ruby hash table implementation. The times reported for each operation are per 1 million records. The sparse bit set outperforms the hash table for each experiment except adding random numbers to the set. For the sparse bit set, adding an integer involves creating and storing two values: the index and the bit field. For the random values, this occurs more frequently. For sequential integers, the cost of creating the index and bit field is amortized over several stores (essentially the number of bits in the bit field).

[3]Source available at https://github.com/presidentbeef/dumb-numb-set

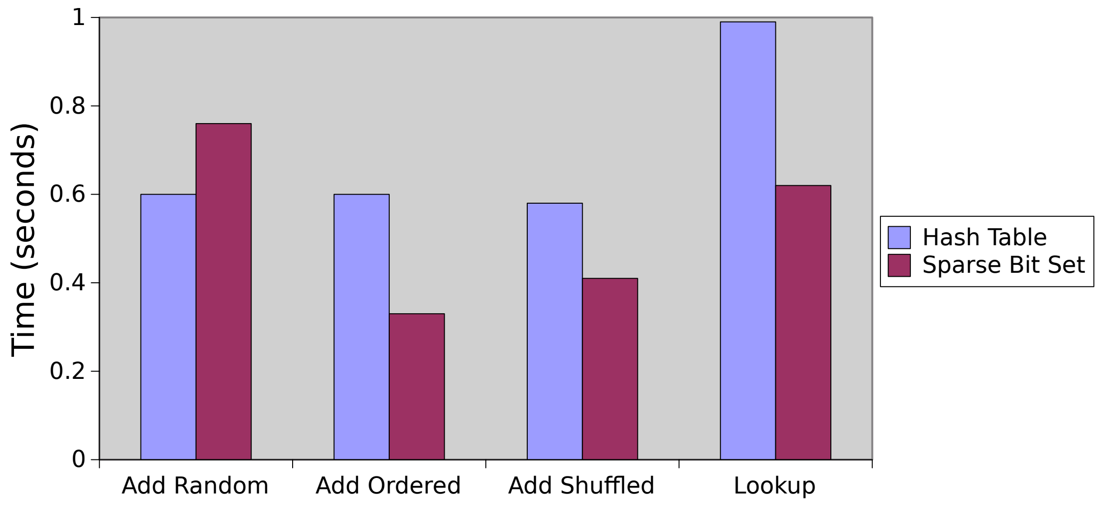


Figure 5.2: Sparse Bit Set Operation Speed

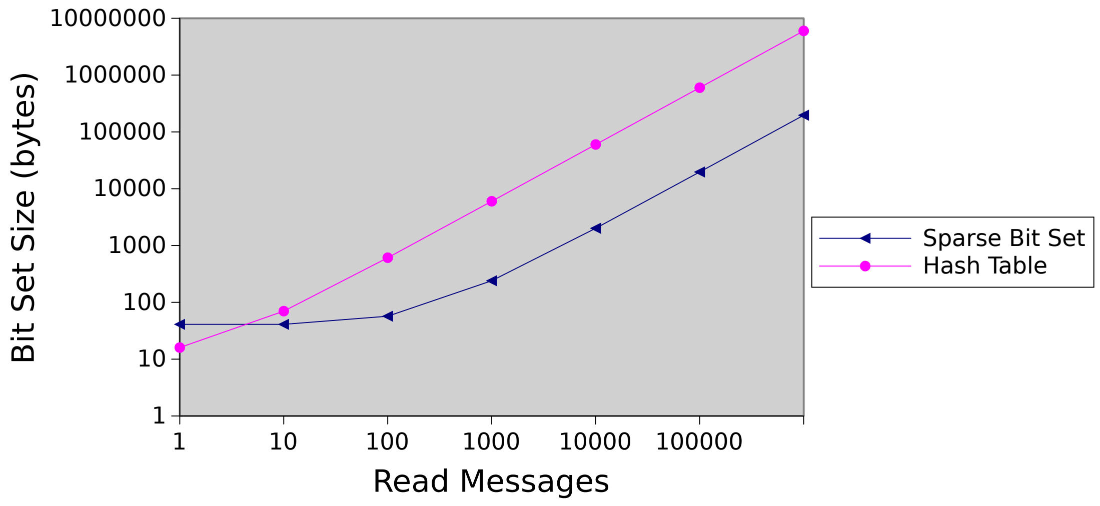


Figure 5.3: Sparse Bit Set Size

Figure 5.3 shows the increase in the size of the bit set as the number of read messages increases. This is the actual serialized size of the data structure as it is sent over the network. After about 1,000 messages, the size grows linearly and averages 1.6 bits per message. At 1 million records, the data structure is about 192 KB. However, this is about 96% smaller than the standard Ruby hash table.

Without pruning, the read message data structure will grow unbounded. There are some potential solutions to this. One solution would be to track read messages per host. When requesting a read-only message from a given host, only the read messages from that host are sent. Unfortunately, this ties applications to hosts and assumes messages will only be on the host which output them. This would prevent message replication as discussed in Section 5.8.

Since read messages may be garbage collected, hosts may provide feedback about which messages have been garbage collected. These messages may then also be garbage collected from the read data structure, essentially pruning it. This could be returned as optional data when a request for read-only messages is made, thereby avoiding any extra communication.

## 5.8 Message Replication

Distributing copies of messages to multiple hosts can increase message availability when a host is temporarily unavailable, under heavy load, or even in the face of network partitioning. Additionally, it can improve performance if messages can be fetched from a nearer host or from multiple hosts in parallel. While not a requirement of the MELON paradigm, message replication may be implemented as an additional feature of MELON without adding any new operations although it does add complexity.

Take-only messages are generally not eligible for replication since their removal must be atomic. Coordinating removals for all replicated copies is not only impractical, it is impossible if a node containing a replicated message leaves the network. Read-only messages, however, are expected to be read many times and cannot be explicitly removed, making them a candidate for replication.

FIFO order (per host) must be maintained no matter which host may actually return the messages. When each host manages its own stored messages, ordering is easily accomplished. When messages are distributed and multiple copies of the same message are available from different hosts the problem is more challenging. However, each host is still aware of the order in which the messages should be returned.

Instead of sending a request for matching messages, a process requests a list of message IDs which would have fulfilled the request, in order. Each host responds with a list of matching message IDs, but only for messages output by that host. The requesting process can then request messages by ID rather than message templates. Any host may return the actual messages, either the original message or replicas. Since the requesting process will be aware of the correct ordering, it can ensure the FIFO ordering is maintained when returning messages to the application.

This approach still requires the original host to be available when the first request is sent, and it is limited to fetching messages which have already been output at the time of the request. As such, it is best fitted to bulk retrieval (**bulk_read**) of messages.

One other type of message may be replicated. Take-only messages which are addressed to a specific receiver may also be safely replicated since they may only be removed by that receiver. The receiver can track which directed messages it has already taken and simply discard duplicates. Replicating messages which will only be needed once seems wasteful, though, except in the case where delivery of private messages is critical.

In any scenario, replicated messages would need to be kept in a separate storage from regularly output messages. Replicated messages would only be returned to requests by message ID or for directed take-only messages. A background process would be needed to replicate the messages out to different hosts. A mechanism would also be needed to determine when to replicate the messages, and when to garbage collect replicas. Overall, message replication requires a considerable amount of added complexity and overhead to MELON.

## 5.9 Garbage Collection

The MELON model explicitly acknowledges memory and storage are especially limited on mobile devices. For simplicity, in this section we consider this to be an absolute limit on the number of messages to be stored at any time. This is inexact in relation to the actually memory used since messages may vary in size.

Since MELON persists messages and read-only messages cannot be removed by applications, it would not be difficult for an application to exhaust available space on a device. This is an issue for any paradigm which persists its messages but most proposals do not address it. When there is no more space to store a message, a communication paradigm implementation may crash, raise an exception, simply drop the message, or perform garbage collection to remove existing messages and free space for new ones.

To offset its otherwise permanent storage of read-only messages, MELON implementations should have a mechanism to remove old messages. MELON's requirements for garbage collection straddle memory management garbage collection and cache eviction. In memory management, any references to a value in memory will require it to be kept, but MELON messages do not have direct references and we cannot know which messages may be needed in the future. This is similar to maintaining a cache, in which the same limitation of knowledge applies. Unlike a cache, there are generally no other copies of the message available to fall back upon. Once evicted from the MELON store, the message is lost.

First-in first-out (FIFO) and least-recently used (LRU) are basic strategies for choosing which messages to replace in storage. For MANET applications, there is often a few messages which are expected to be available for extended periods of time - for example, messages containing identity of nodes or other static data. This suggests simply discarding the oldest messages might not be the best approach. When evicting by LRU, it is necessary to track the last access time for each message. When deciding which messages to remove, the LRU policy drops the messages with the oldest access time. This is probably a good fit for MELON, but there are also more sophisticated variants on LRU which may be useful. Determining what replacement strategy is best for MELON is a topic for future work.

Besides determining which messages should be removed from storage, it is also necessary to have a policy for when to perform garbage collection. Unlike memory garbage collection, MELON garbage collection is not expected to have a significant impact on performance. MELON garbage collection does require locking the store to actually remove the messages, but the determination of which messages to remove may be performed without locking since it does not need to be exact. One approach is to only perform garbage collection when the message limit is actually reached, then to remove either a single message or a percentage of messages. Removing more messages reduces frequency of garbage collection, but increases the time spent removing messages each time.

Since MELON messages need to be stored in FIFO order, garbage collection does require some form of compacting to ensure new messages may be added at the end of the queue.

Fitting the best garbage collection strategy to MELON remains as future work.

# CHAPTER 6

# Case Studies

This chapter presents the implementation of several applications in MELON. The code examples are written using the Ruby implementation of MELON in order to demonstrate concrete usage of MELON operations.

## 6.1 News Server/Reader

In this section, we consider news servers which produce news reports, each with a category and a headline. News readers use a client which fetches all news headlines from a given category.

Listing 6.1: News Server

```
class NewsServer
  def initialize
    @melon = Melon.new
  end

  def report(category, headline)
    @melon.write([category, headline])
  end
end
```

A class implementing the news server is shown in Listing 6.1. To ensure all interested parties can read the news, the server uses **write** to disallow a reader from removing a news item and preventing other readers from reading it. When a news item is reported, the server simply writes a message containing the category and headline.

Multiple news servers may be producing news reports at the same time.

Listing 6.2: News Reader

```
class NewsReader
  def initialize
    @melon = Melon.new
  end

  def fetch(category)
    @melon.read_all([category, String])
  end
end
```

The news reader is just as simple, as shown in Figure 6.2. The `fetch` method will fetch all reports in the given category. Repeated calls to `fetch` will only return news reports which have not already been read. The method will block if no new reports are available.

## 6.2 Chat Application

Basic chat applications are a common example of networked communications. Participants broadcast messages tagged with their name, which are received by all participants.

Listing 6.3: Chat Application

```
class Chat
  def initialize name
    @name = name
    @melon = Melon.new
  end

  def chat(message)
    @melon.write([@name, message])
  end

  def read_messages
    @melon.read_all([String, String])
  end

  def start
    monitor
    loop do
      print "? "
      message = gets.strip
      chat message unless message.empty?
    end
  end

  def monitor
    Thread.new do
      loop { print_messages(read_messages) }
    end
  end

  def print_messages(messages)
    messages.each do |name, message|
      puts "\n<#{name}> #{message}" unless name == @name
    end
  end
end
```

The class in Listing 6.3 implements a simple command-line chat client. When the `start` method is called, the client starts a new thread which reads and shows any messages sent by other chatters. Note the `print_messages` method filters out messages sent by the current chatter (this assumes everyone has a unique name). This is because the `read_messages` method will pull in all unread messages, even those sent by the current client. A more sophisticated message matching system would be able to provide negative template values (i.e., "match any string which does not match this pattern"), but for our prototype this is not possible.

In the main thread, the client requests messages from the current chatter and sends them using the `chat` method. The `chat` method then simply wraps the message in an array with the chatter's name as the first value, then sends the message using **`write`**, since we want the message to be available for many chatters to read.

As usual, messages from a single chatter will always be received in the order the messages were sent, but no order is imposed across all chatters.

## 6.3 Job Queue

In a job queue, tasks are added to a shared queue. Workers remove tasks from the queue and execute them. Naturally, access to the queue should be atomic to maintain consistency and a job should only be executed by a single worker.

Listing 6.4: Job Producer

```
class Producer
  def initialize
    @melon = Melon.new
  end

  def add_job(job)
    @melon.store([job])
  end
end
```

The job producer in Listing 6.4 does very little, just saves the jobs in the MELON shared storage using **store**. Jobs are assumed to be a subclass of a `Job` class with an `execute` method to be called by the workers. Any number of producers may add jobs to the queue.

Listing 6.5: Worker

```
class Worker
  def initialize
    @melon = Melon.new
  end

  def fetch_job
    @melon.take([Job])[0]
  end

  def work
    loop do
      fetch_job.execute
    end
  end
end
```

Workers retrieve jobs using **take** and invoke the `execute` method on them. If no jobs are available, the worker will block. Since MELON's **take** operation is atomic, there is no danger of a job being run by more than a single worker. If there is only a single job producer, the jobs will be executed in first-in, first-out order as enforced by MELON semantics.

## 6.4 Experiment Coordinator

This case study examines an application developed and used to generate the experimental results in Chapter 7. To ease the process of repeatedly setting up

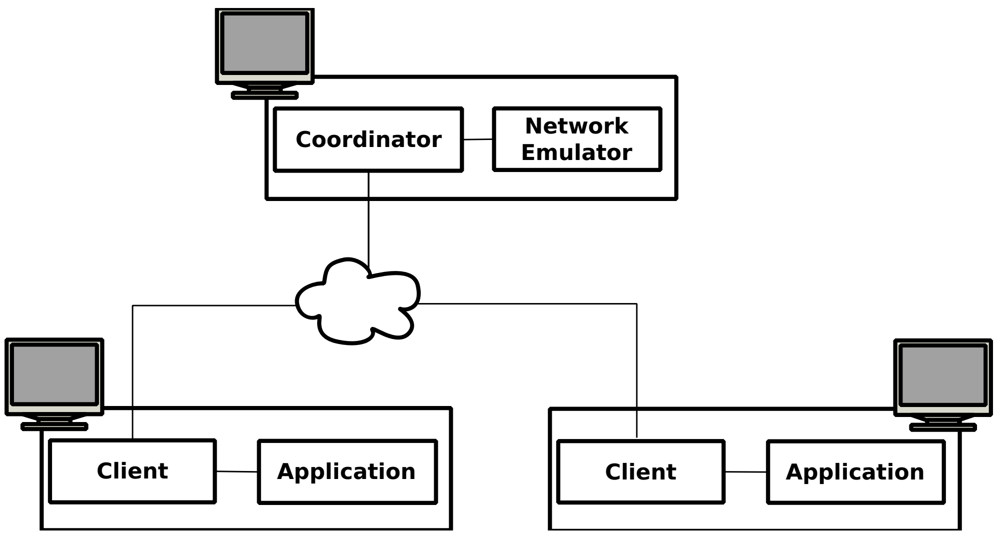


Figure 6.1: Coordinator Architecture

experiments, we developed an experiment coordination framework written with MELON. The framework handles running real applications on multiple hosts, executing the network emulator, and gathering results into a single location.

The architecture of the framework is illustrated in Figure 6.1. For simplicity, the coordinator resides on the same host as the network emulator. The coordinator sends out commands to clients which reside on each host. The clients are responsible for executing programs on their local host and sending resulting output back to the central coordinator.

When running the coordinator, the number of clients and their IP addresses is known. This simplifies operations but also allows an experiment to be run with a specific number of nodes. It is also possible to set two different commands to be run on different types of nodes. For example, one node can be a "source" and run one command, while the other nodes are "sinks" and run a separate command.

Table 6.1 lists an overview of the MELON commands used to communicate between the coordinator process and the client processes. The order of communications are not always strict. For example, it does not matter if the coordinator first sends the commands and then the clients wait, or if the clients wait first.

Table 6.1: Coordination Framework Messages

| Coordinator | Clients | Description |
|---|---|---|
| | **read**([type, String]) | Await command |
| **write**([“client”, client_command]) | | Send client command |
| **write**([“server”, server_commnad]) | | Send server command |
| **take**([“confirm”, ip]) | | Wait confirmations |
| | **store**([“confirm”, ip]) | Store confirmation |
| | **read**([“go”]) | Await go signal |
| **write**([“go”]) | | Send go signal |
| **take**([“done”, ip]) | | Wait for clients to finish |
| | **store**([“result”, ip, output]) | Store results |
| | **store**([“done”, ip]) | Store done signal |
| | **read**([“finished”]) | Await finished signal |
| **take_all**([“result”, String, String]) | | Gather results |
| **write**([“finished”]) | | Send finished signal |

In practice, these operations will be concurrent. Coordination between processes is achieved by blocking on messages containing signals (“confirm”, “go”, “done”, etc.)

When starting, the coordinator writes a message for each command type (called “server” and “client” for convenience). Then it performs a **take** operation for a confirmation message from each client IP address. Each client reads the relevant command, starts the command, waits for the application to initialize, then uses **store** to send a confirmation message with its IP address.

When the coordinator has taken a confirmation from each host, it starts the network emulator, then writes a “go” message. Upon reading the “go” message, each client signals the application to begin. The coordinator then blocks waiting on “done” messages from each client.

As the application runs, the client reads the output line by line and stores each line using **store**, labeled with the client’s IP address. When the application finishes, the client stores a “done” signal. When the coordinator has taken a “done” message from each client, it collects the results and then sends a “stop” message. The clients then stop the applications and the framework is ready to start the next experiment.

Unlike the other examples in this chapter, this application was developed to fill an actual need, as opposed to being strictly a demonstration of how to use MELON. We have used all of MELON’s operations except **read_all** in this application. Read-only operations were used to broadcast messages from the coordinator to the clients, while **store/take** operations were used to send messages back to the coordinator. The default blocking behavior for **read/take** were useful to coordinate actions.

In our usage of the experiment coordination framework, the output from the applications were stored and organized by the coordinator into separate files by client. Since MELON retrieves messages in order per host, no ordering information needs to be included in the output messages. The coordinator merely writes the messages out to the proper file (by client IP address, which is included in the message) in the order it received them.

## 6.5 Shared Whiteboard

This section presents the implementation of a shared whiteboard not only using MELON, but also publish/subscribe, RPC, and tuple spaces in order to compare their features and performance. The performance results are presented in Section 7.1.5.

A shared whiteboard is a digital document which may be edited and viewed by multiple users concurrently and is commonly proposed as an example of an application well-suited to MANETs [Ra13] [RK13]. Shared whiteboards are distributed, real-time, and interactive, which presents some interesting characteristics. Since many participants may be updating the whiteboard, ordering of changes is very important to maintain a consistent document. It is also important that changes be propagated quickly so that each user is working with the latest document.

Each version shares common code related to the actual whiteboard itself, which is implemented in the `Whiteboard` class. Changes to the shared whiteboard are encapsulated in a `Figure` object. Each version implements an `add_local_figure` method which is called when the user modifies the shared whiteboard. The MELON and tuple space versions also implement an `add_remote_figures` method which is used to retrieve updates from remote nodes.

### 6.5.1 Publish/Subscribe

The publish/subscribe whiteboard in Listing 6.6 sets up a subscription to the "whiteboard" topic and a callback to add remotely published figures to the whiteboard. This allows the whiteboard to receive updates at any time in a separate thread, which is precisely what would be desired. To output a new figure, the whiteboard simply publishes the figure to the "whiteboard" topic.

Listing 6.6: Publish/Subscribe Whiteboard

```
require "ps"
require "whiteboard"

class PSWhiteboard < Whiteboard
  def initialize
    @ps = PS.new

    @ps.subscribe("whiteboard") do |figure|
      add_figure(figure)
    end
  end
  def add_local_figure(figure)
    @ps.publish("whiteboard", figure)
  end
end
```

Listing 6.7: RPC Whiteboard

```
require "rpc"
require "whiteboard"

class RPCWhiteboard < Whiteboard
  def initialize
    @rpc = RPC.new
    @rpc.export(self)
  end

  def add_local_figure(figure)
    wbs = @rpc.find_all("RPCWhiteboard")
    wbs.add_figure(figure)
  end
end
```

### 6.5.2 RPC

A shared whiteboard implementation using RPC is shown in Listing 6.7. When the whiteboard is initialized, it exports itself as a remote object. Remotes hosts can then remotely invoke `add_figure`. Like publish/subscribe, this allows the whiteboard to accept remote figures asynchronously from the main process thread and is a natural feature of RPC. Distribution of remote figures is performed by first finding all remote instances of `RPCWhiteboard`, then invoking the `add_figure` method (defined on the parent class) directly, passing in the new figure as an argument. Since group RPC is asynchronous, it is possible that a call might complete before a prior call.

### 6.5.3 Tuple Spaces

Listing 6.9 shows the tuple space version, which is very similar to MELON. To send an update, it outputs a tuple containing just the new figure. Unlike MELON, a misbehaving or misconfigured client could remove the messages from the tuple space, disrupting the shared whiteboard communication. Retrieval of remote messages uses a **bulk_rd** operation to read all messages containing a figure. To continuously retrieve messages asynchronously, this method can be called inside a loop in a separate thread. Once a group of figures is retrieved, each individual figure is added to the local whiteboard.

As discussed in Section 2.3.3, **copy-collect** may be used to solve the "multiple read problem". We have implemented this as the **bulk_rd** operation. However, this does not solve what might be termed the "multiple multiple read problem": since our tuple space is not static, reading all matching tuples once is not sufficient. We need to be able to perform multiple **bulk_rd**s to add all figures the whiteboard. Without *a priori* knowledge of remote hosts in the system, the only option which allows concurrent access to the tuple space is to read *all* matching tuples. Naturally, this becomes considerably expensive as the number of tuples grows large.

Listing 6.8: MELON Whiteboard

```
require "melon"
require "whiteboard"

class MelonWhiteboard < Whiteboard
  def initialize
    @melon = Melon.new
  end

  def add_local_figure(figure)
    @melon.write([Figure])
  end

  def add_remote_figures
    figures = @melon.read_all([Figure])

    figures.each do |figure|
      add_figure(figure[0])
    end
  end
end
```

Listing 6.9: Tuple Space Whiteboard

```
require "tuplespace"
require "whiteboard"

class TSWhiteboard < Whiteboard
  def initialize
    @ts = Tuplespace.new
  end

  def add_local_figure(figure)
    @ts.out([Figure])
  end

  def add_remote_figures
    figures = @ts.bulk_rd([Figure])

    figures.each do |figure|
      add_figure(figure[0])
    end
  end
end
```

### 6.5.4 MELON

The MELON whiteboard in Listing 6.8 writes out each figure in a tuple containing just the new figure. It uses the **write** operation since every remote node needs to be able to read the figures. To retrieve remote figures, MELON uses **read_all** to nondestructively read all messages containing a `Figure`. Like tuple spaces, the `add_remote_figures` method should be called in a separate thread to provide asynchronous updates. Unlike tuple spaces, MELON's **read_all** operation only retrieves unread messages, eliminating the "multiple multiple read" problem.

MELON directly provides three features which are helpful to the whiteboard application: persistent messages, reading only unread messages, and returning messages in a per-host ordering. Message persistence is crucial in MANET applications, where communication with remote nodes is often disrupted and delayed. For a shared whiteboard, every message must be delivered to keep the document synchronized between users. By managing read versus unread messages, MELON easily allows the whiteboard to efficiently fetch only newly-added figures. Finally, MELON guarantees the updates from each host will be retrieved in the order that host initiated them. While this does not provide a global ordering, it does ensure updates from a single host will be in order.

# CHAPTER 7

# Evaluation of MELON

## 7.1 Performance Analysis

### 7.1.1 Operation Speed

To establish a baseline for performance, we measured the time for the **write**, **read**, **store**, and **take** operations directly on a local message storage and compared the results to the LighTS [Bal07] local tuple space implementation used by LIME. In these experiments, all messages are first stored, then either read or removed from the local storage. No network communication is involved.

When comparing **read** and **rd**, we simulate the MELON's feature of only returning unread messages by using a sequential integer ID in the tuples and performing a **rd** operation for each ID. If we did not do this, LighTS would return the same tuple for each **rd** operation.

In both LighTS and MELON, messages are stored in what is essentially an array. Since we are not removing messages, each operation must linearly search the array taking O($nm$) time, where $n$ is the length of the message or tuple and $m$ is the number of stored messages or tuples. Naturally, operations returning messages near the beginning of the array are faster, while the slowest operation returns the last message in the array. In our experiments, this cost did not become apparent until searching 100,000 messages. The average time per operation from 10,000 to 100,000 increased $\tilde{9}$x for LighTS and $\tilde{1}$0x for MELON, with total read time taking just under a minute. It is unsurprising MELON is slightly slower,

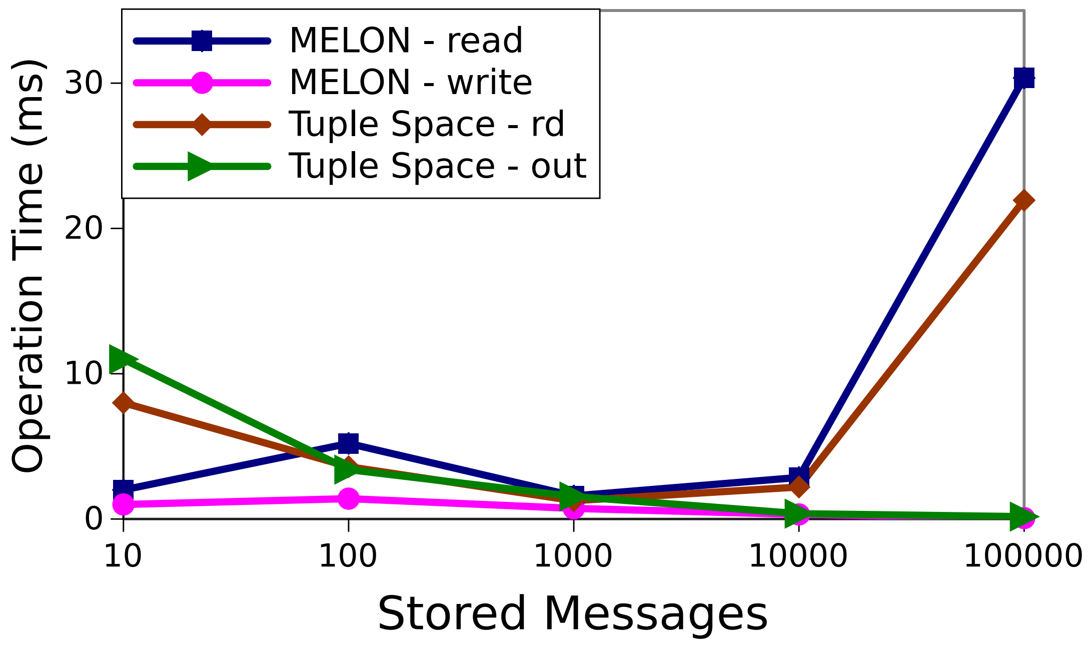


Figure 7.1: Read Speed

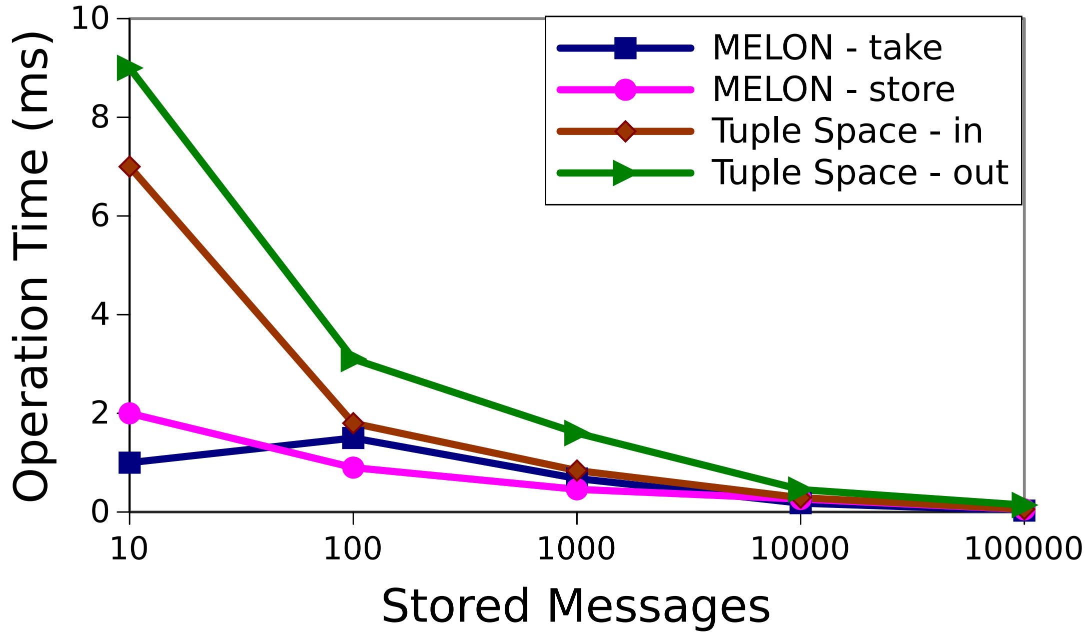


Figure 7.2: Take Speed

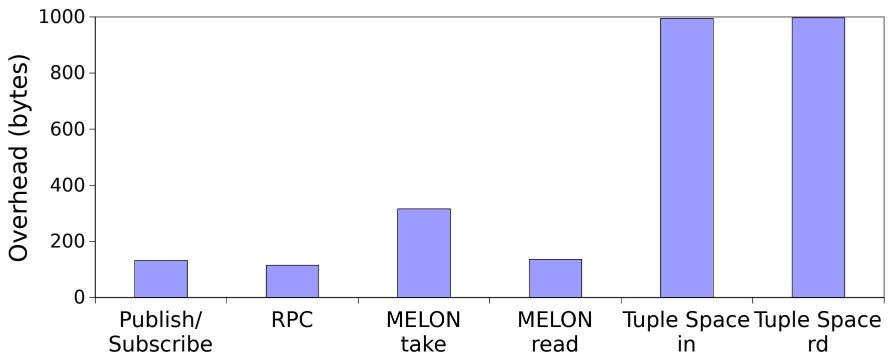


Figure 7.3: Message Overhead

since it must also check that a message is not in the "read" list before returning it.

On the other hand, removing messages is naturally quite fast, since the matching message is always the first message in the store. All **take**/**in** operations require less than 8$ms$ to execute on average. MELON is slightly faster here due to differences in how removal is implemented, although average speed per operation converges as the number of operations performed increases.

Storing messages is faster than removing them for both implementations. In LighTS there is slightly more constant overhead for adding new tuples, so **out** operations are a little slower than **write** and **store** in MELON. However, in reality both implementations are plenty fast for typical applications, since storing a message takes less than 10$ms$ on average, and usually less than 4$ms$.

Overall, MELON performs roughly the same or better than LighTS when performing serial operations.

### 7.1.2 Communication Overhead

For any communication library or framework, the message size added by use of the library is an important factor in determining its usefulness. In these experiments, we measure the number of bytes actually sent over the network, divide by the number of messages sent (in this case, 1000) and subtract the 1KB application payload. This leaves us with the overhead introduced by the paradigm. We compare the overhead of MELON to canonical implementations of publish/subscribe, RPC, and tuple spaces in Figure 7.3.

Publish/subscribe and RPC have very low overhead and provide a good baseline. In the case of publish/subscribe, the only added information to a publication is the topic. Periodic subscription messages are small and infrequent compared to the number of messages sent. For RPC, there is one initial exchange to find the remote object, then later messages only need the object and method names plus the payload itself.

As in the operation speed experiments, we use the LighTS tuple space implementation. The serialized versions of tuples and tuple templates are very large and must be sent for each request. If a simpler data structure were used, overhead would be expected to be similar to MELON's overhead for **take**.

For MELON, **take** and **read** requests must send a message template, so the size of the request is dependent on how many values the template contains. For **read** operations, each request must also send information on previously read messages as described in Section 5.7, which increases as the number of read messages increases.

### 7.1.3 Message Latency

Figures 7.4 and 7.5 show the average latency between a client's request for a message and the receipt of a matching message. The error lines indicate the standard deviation. In these experiments, a single host writes out 1,000 messages with a 1KB payload, and the other hosts concurrently read the messages. Tuple spaces and MELON used the **rd/read** operations to retrieve the messages one at a time, rather than the bulk retrieval with **rdg** or **read_all**. Since publish/subscribe

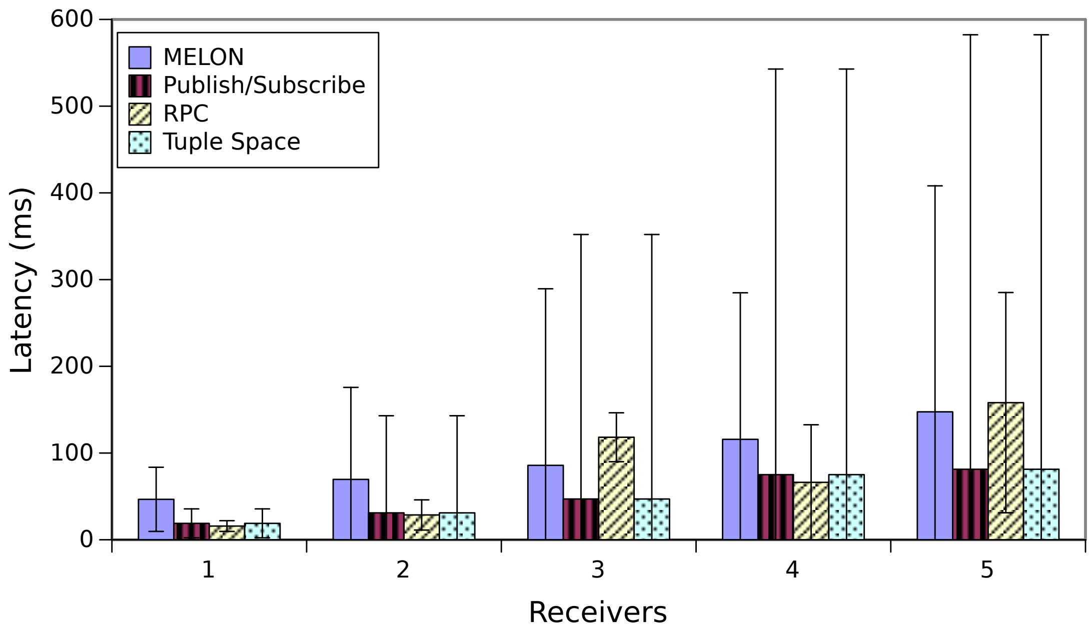


Figure 7.4: Message Latency - Static Scenario

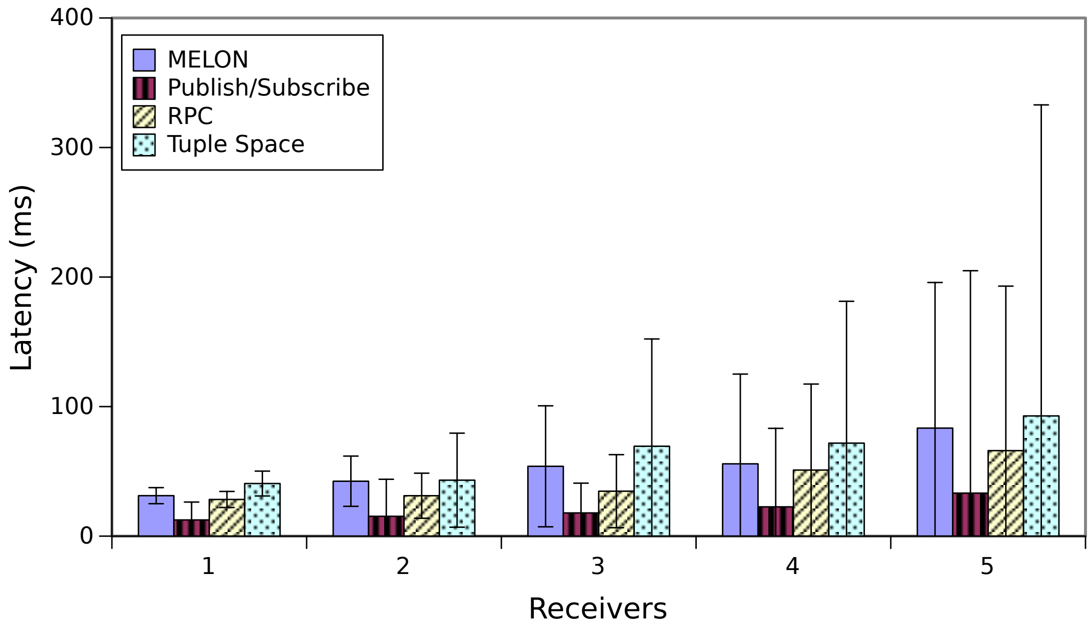


Figure 7.5: Message Latency - Mobile Scenario

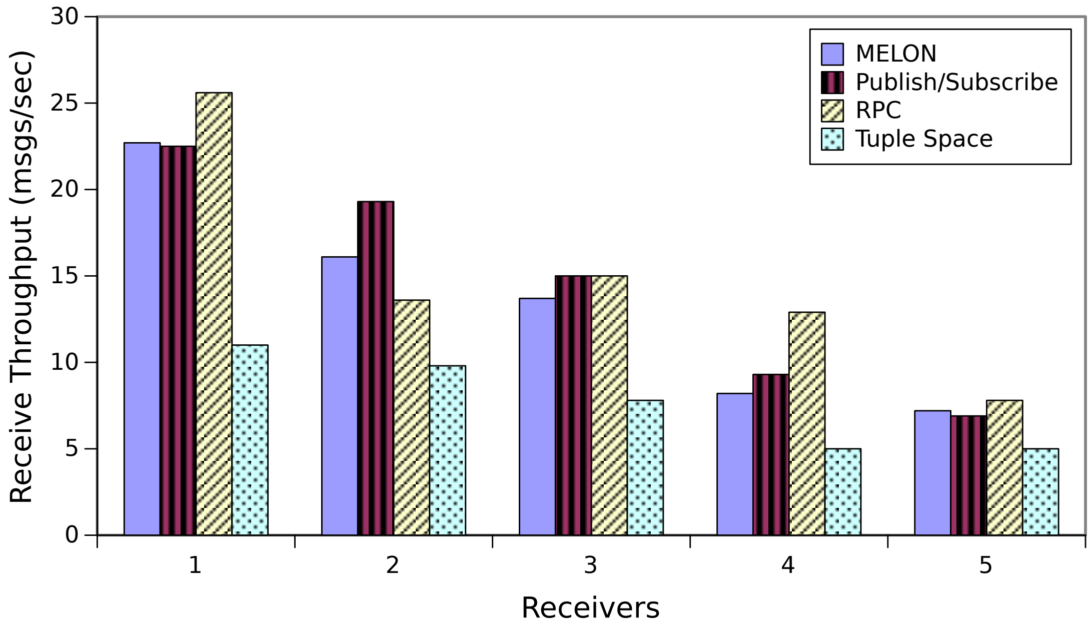


Figure 7.6: Message Throughput - Static Scenario

does not involve a "request" beyond the initial operation, latency was measured as the time elapsed between receiving sequentially numbered publications.

MELON does show higher average latency rates in the static scenario, although for three or more receivers the standard deviation is lower than the other paradigms. In the more realistic mobile scenario, however, MELON latency is about the same or slightly lower than tuple spaces. Comparing the static and mobile scenarios also demonstrates one of the issues in wireless networks: the mobile scenario allows distinct routes to form between hosts with less interference, while the static scenario has many collisions causing more communication delays even though the distance between devices is smaller.

### 7.1.4 Message Throughput

Throughput in these experiments was measured on the receiver side in terms of messages delivered per second. As in the other experiments, 1,000 messages with

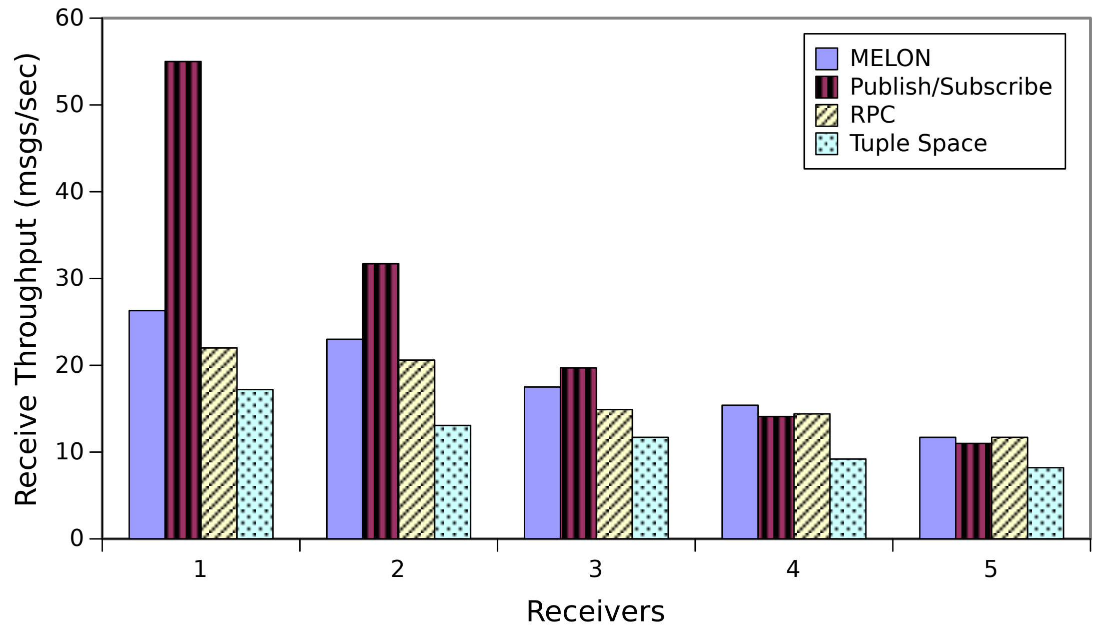


Figure 7.7: Message Throughput - Mobile Scenario

a 1kb payload are output by one host, while the other hosts read the messages one at a time. Figures 7.6 and 7.7 show the average throughput as the number of receivers increases.

Publish/subscribe dominates in these experiments since it is the only push-based paradigm, allowing the sender to publish messages at a high rate without requests or acknowledgments from the receivers. Also, subscribers may receive multiple publications concurrently which increases throughput capacity.

Despite having higher average latency than the other paradigms, MELON demonstrates good throughput in both the static and mobile scenarios. However, throughput for all paradigms drops off dramatically as more receivers are added. Also, all paradigms performed better in the mobile scenario than in the static scenario. This is likely due to two factors: devices were not forced to be more than one hop apart, and the large number of devices allows distinct routes between devices and decreases wireless broadcast collisions.

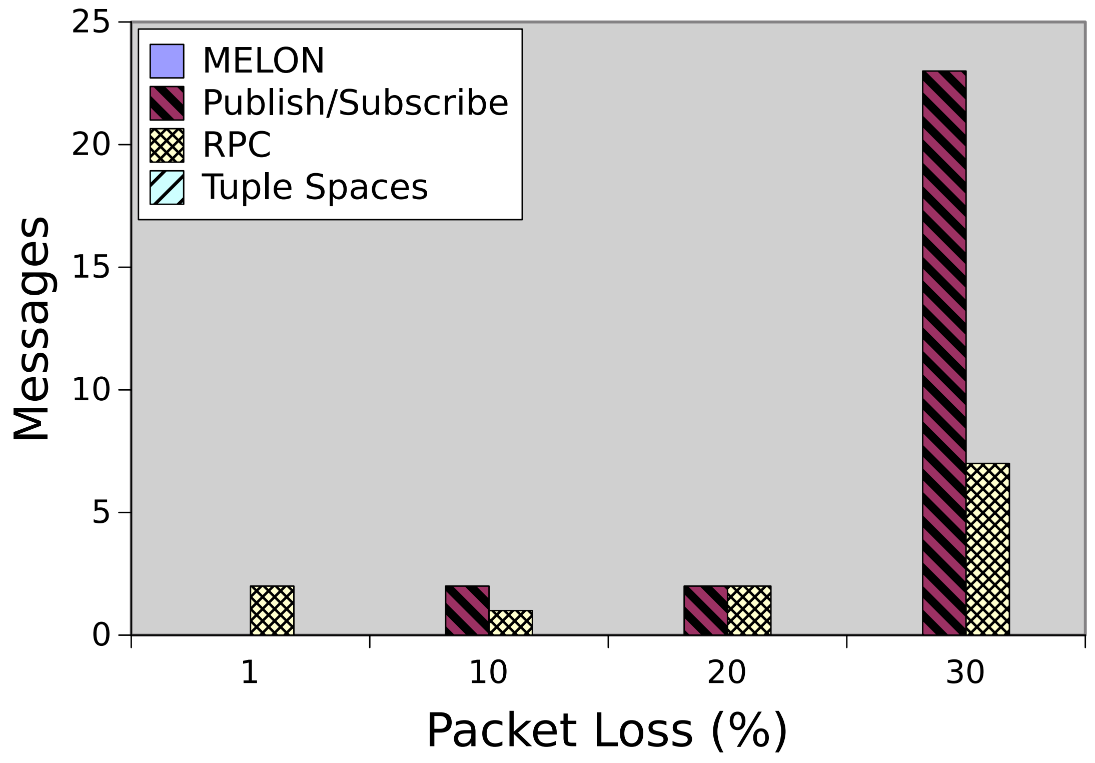


Figure 7.8: Host Out-of-Order Messages

### 7.1.5 Whiteboard Performance

For each implementation, we measured the number of messages lost, the number of messages received out of order, and the message latency. For out-of-order messages, we divided it into two metrics: host out-of-order and global out-of-order. Host out-of-order messages are messages from a single host which are not received in the order sent. Global out-of-order messages are those received before their preceding message. For example, if node A receives a message $m_1$ from node B, then sends $m_2$. If node C receives $m_2$ prior to $m_1$, $m_2$ will be considered out of order.

In our experiments, messages from a single host were generally delivered in the order they were sent as shown in Figure 7.8. For MELON and tuple spaces, no messages were delivered out of order. However, it should be noted that for

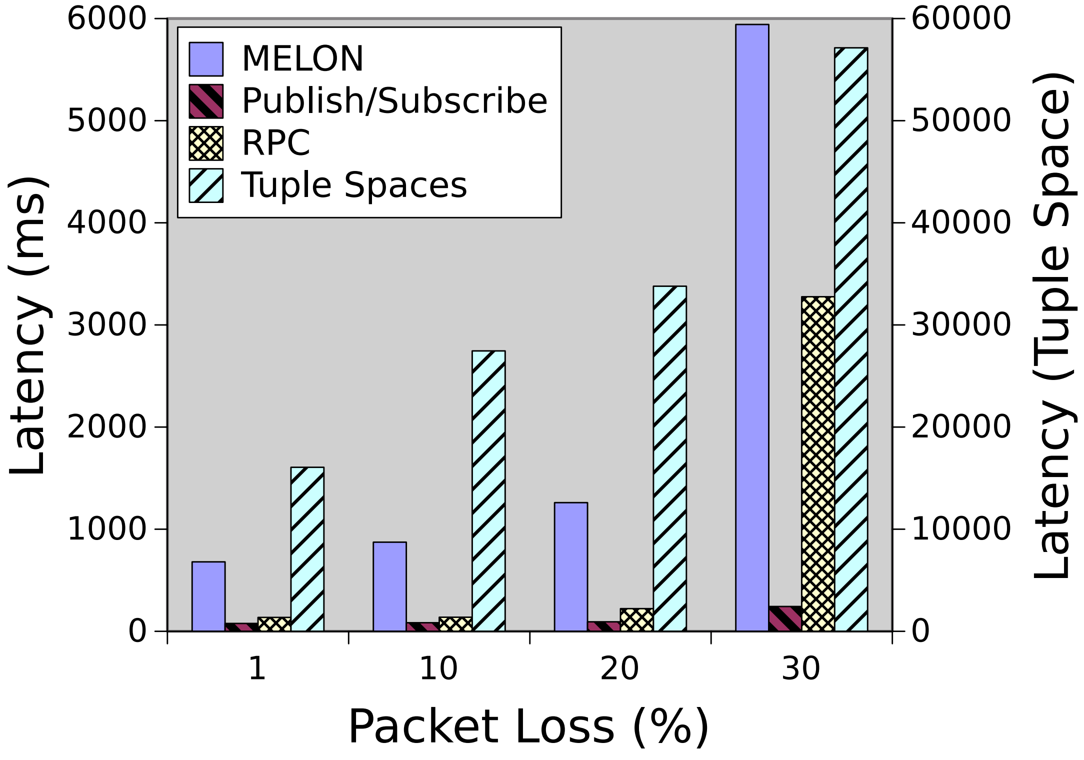


Figure 7.9: Message Latency

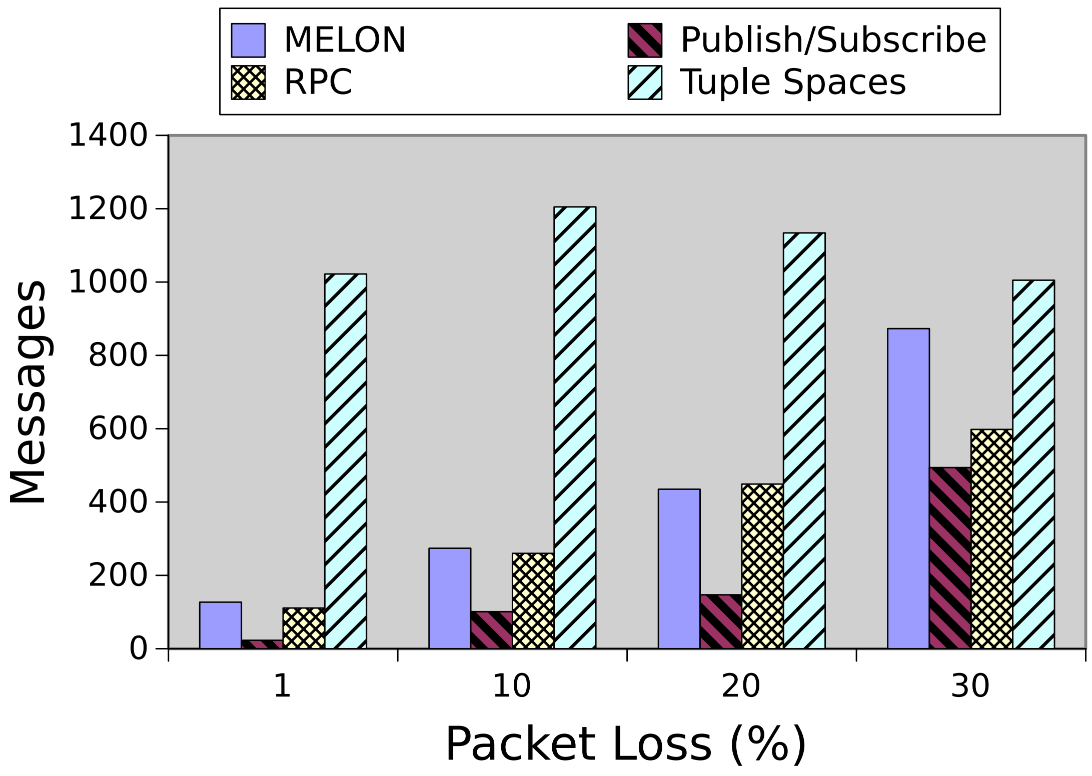


Figure 7.10: Global Out-of-Order Messages

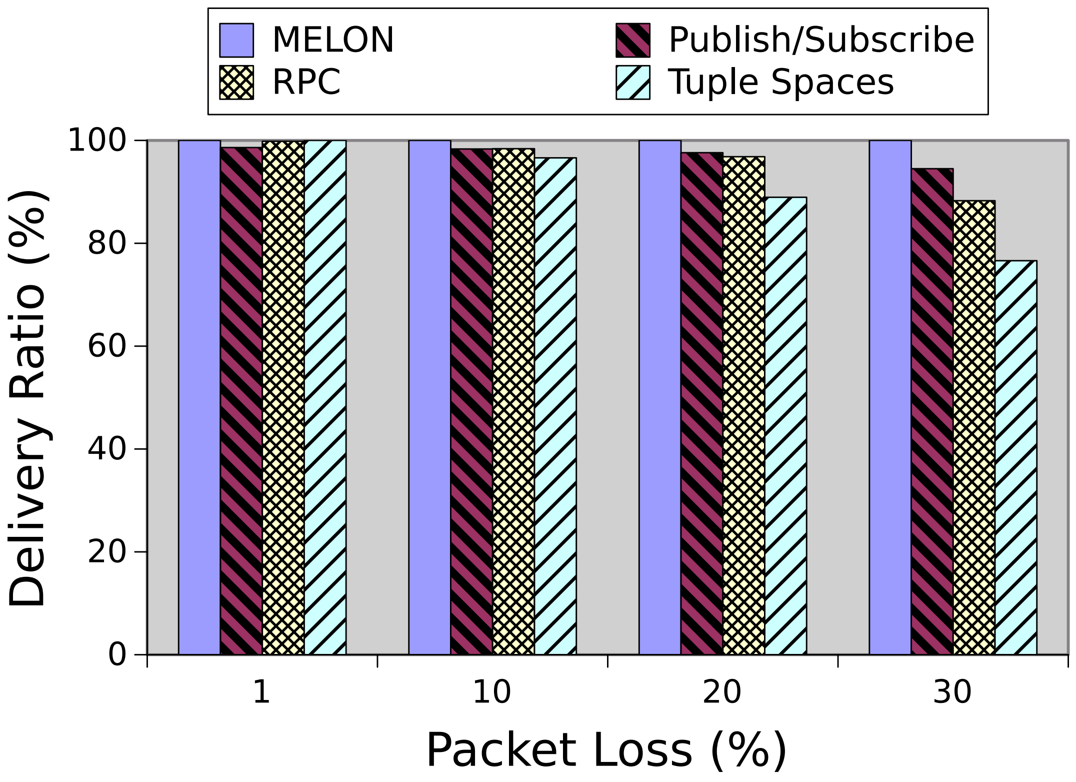


Figure 7.11: Delivery Rates

tuple spaces this is an accident of the implementation, whereas in MELON it is guaranteed. In LighTS, tuples are sequentially stored locally in an array in the order they are output, then returned in that same order when they are matched. Tuple spaces in general do not return matched messages in any particular order.

In this application, RPC is used in an asynchronous manner since it is providing group communication. If one call is delayed, it is possible a subsequent call will complete before a prior one, which explains why RPC delivers a small number of messages out of order. Publish/subscribe is fully asynchronous and incoming publications can even be processed concurrently. However, even in the worst case publish/subscribe delivers 97.8% of the messages from a host in the order they were sent. Unfortunately, like tuple spaces this is just the result of a single implementation and the RPC and publish/subscribe paradigms make no promises about the ordering of messages.

Unlike per-host ordering, many messages were delivered out of order from a global perspective as can be seen in Figure 7.10. This is entirely expected, since none of the paradigms provide a global ordering. Enforcing a global ordering in an unreliable network is not feasible, since nodes may become unavailable at anytime while continuing to output messages. However, the global ordering remains important for a shared whiteboard.

Our results show publish/subscribe performs the best for this metric. Indeed, ordering is largely dependent on deliveries completing quickly before later messages overtake them. As shown in Figure 7.9, publish/subscribe is an extremely quick method for delivering messages, so it excels in ordering as well. Conversely, tuple spaces fare the worst, delivering 67% of messages out of order. Again, because tuple spaces provide no way of controlling which matches messages are returned or in what order, the whiteboard implementation must transfer large amounts of tuples in order to nondestructively read all matching messages. This is extremely slow, as reflected in Figure 7.9.

MELON and RPC perform about the same for global ordering, although MELON is more affected when the network conditions worsen. This is likely due to MELON's reliable message delivery (Figure 7.11), since some messages may be delayed significantly by broken network routes or network partitioning. In contrast, losing messages can improve ordering since a message not delivered cannot be out of order. Of the paradigms compared, MELON is the only one to demonstrate 100% message delivery. Tuple spaces would also be expected to be reliable, but again in this application it is required to deliver large amounts of messages. Given that the median latency for tuple spaces reached a full minute, the experiment completed before some messages arrived.

While we have seen publish/subscribe have low delivery ratios in the past [CB10], here it performs well in the lossy environment due to its quick delivery rates, but still dropped 1.4% of messages when the network connectivity was good. RPC performs predictably, slowing losing more messages as the network degrades. Again, we are using group RPC, which means the application is not aware of how many receivers may be available and therefore does not retry to complete calls if a host cannot be reached for a period of time. Fully synchronous RPC would block the process until the message is delivered. However, that would also delay deliveries considerably which is not acceptable for a whiteboard application.

Median time between sending and receiving a message is reported in Figure 7.9. Since tuple spaces are so much slower, the results are aligned with the right-hand y-axis which is an order of magnitude higher. Publish/subscribe was extremely quick, which is expected since it requires no message confirmations nor active discovery of remote hosts. RPC was also quite fast until it was slowed down along with the other paradigms by the 30% packet loss.

Logically, delivery rates and latency are directly related. With reliable delivery some messages may be very late, increasing overall latency. On the other hand, dropped messages do not count towards the latency metric, so a lossy communication paradigm can appear to be very fast. MELON errs on the side of reliability, and therefore is a bit slower as the network becomes less reliable and more delivery attempts are required. There is an additional trade-off that pull-based paradigms like MELON and tuple spaces must make, which is the frequency of the pull attempts. Publish/subscribe and RPC may send as soon as a message is ready, but MELON and tuple spaces must continually poll to receive messages. Faster polling results in faster message delivery, but higher overall network usage, collisions and resource monopolization.

## 7.2 MELON Suitability for MANET Applications

In the previous chapters, we have noted three important features which we believe should be provided by a general purpose communication paradigm for applications operating in a mobile ad-hoc network: *disconnection handling*, *resource addressing and discovery*, and *flexible communication*. This section examines how well MELON implements these features.

### 7.2.1 Disconnection Handling

MELON does not have a concept of connections exposed to the application layer. Applications are not affected by disrupted or lost connections, because MELON effectively hides these disconnections in its operations. Since MELON provides spatial decoupling, not only does the application not maintain connections to specific hosts, it is completely unaware of the location of any resources to begin with.

Besides intermittent disconnections which may disrupt communications but from which applications can quickly recover, prolonged disconnections where a node is unreachable for some time (but eventually returns) are also common. The message persistence in MELON provides temporal decoupling between nodes, so that a process on one node may output a message which may be read at any later time, possibly much later. This approach adapts well to the MANET environment, where the network topology may be constantly changing. It allows messages to be received opportunistically instead of only having a chance of receiving the message at the time of sending.

Finally, it is possible to provide some message replication in MELON as discussed in Section 5.8 in order to deal with permanent disconnections between nodes. Message replication allows messages to potentially cross network partitions by copying messages to nodes which may then travel to parts of the network unreachable by the original sender. It also increases availability of messages and can improve performance if a cached message is available at a node closer than the original sender.

### 7.2.2 Resource Addressing and Discovery

Resources in MELON are messages, which may be read or retrieved by matching their content with message templates. It is never necessary or even possible to know where messages physically reside, nor any type of identifier (e.g., an IP address) for the host. In MANETs, this is an important feature because it allows resources to be independent of physical location.

In some traditional distributed systems, there is a centralized directory or index of resources. In MANETs, any type of centralized system is difficult to maintain, since the nodes may leave the network at any point and without any warning. A decentralized index is possible, but quickly becomes challenging since information needs to be migrated as nodes join and leave, and it is difficult to have a decentralized index in small networks, such as communication between just two nodes.

Instead of relying on an index or directory, MELON searches for messages entirely on-demand. When a message is requested, each available remote node is queried until the request can be fulfilled. This allows MELON to be entirely distributed and avoids the complexity of maintaining a decentralized index or overlay network.

### 7.2.3 Flexible Communication

To be of use in a wide range of applications, a communication paradigm needs to provide a number of communication patterns. At a minimum, both multicast and unicast communications should be straightforward.

MELON's **write/read** operations provide "enforced" multicast communication, in the sense that not only can any number of processes read the messages, but the messages also remain available for reading until garbage collected. Messages exchanged via **store/take** are similarly available to any process, but only one process may ultimately receive the message. This could be considered something like anycast. MELON also makes a provision for private unicast communication. When a message is sent with a **store** operation, it may also be addressed to a specific receiver. This receiver is the only process which may **take** the message.

MELON also supports receiving messages in bulk, which is a more efficient method for transferring many messages in a single operation. Because MELON only retrieves each message once, it can safely be used to gather all matching messages in bulk without requiring the shared message store to be static.

In all its operations, MELON enforces ordering at the host level. In other words, messages from a single host are received in the order they were sent. While this does not impose an ordering across hosts, it is still very useful in instances where messages are streamed or when messages are aggregated per host.

# CHAPTER 8

# Conclusions

In this dissertation, we have presented an investigation into communication paradigms for applications operating in mobile ad hoc networks. Starting with an qualitative survey of existing projects to support communication MANETs, we then compared performance of representative projects using real applications in an emulated MANET environment. We showed the wireless and mobile environment is dramatically different from a static wired network and communication libraries must be adapted to the MANET environment. We determined *disconnection handling*, *resource addressing and discovery*, and *flexible communication* to be important issues to address for MANET communication libraries.

We found most projects were based on traditional distributed computing paradigms, but it was not possible to assert any conclusions regarding the underlying paradigms, since the project implementations compared used different languages and were of varying quality. In order to study the paradigms' performance in a quantitative manner, we implemented our own versions of three commonly-used paradigms (publish/subscribe, RPC, and tuple spaces) with as much shared code as possible. This allowed us to fairly compare the paradigms using real applications.

After empirically investigating the traditional paradigms, we found message persistence and connectionless operations to be especially beneficial for MANET applications. RPC relies too heavily on synchronous communication between hosts and is not well-suited for group communication with an unknown set of participants or unreliable remote hosts, although it does provide reliable unicast communication. Publish/subscribe is fast and light, but does not provide reliable delivery or convenient unicast communication. Implementing a system of brokers to provide persistence and manage subscriptions is complex in a MANET. Tuple spaces, not even originally designed for distributed systems, do provide message persistence and flexible communication, but are hampered by strict semantics and lack of support for message streams. None of the three paradigms provided private unicast communication, a common requirement for modern mobile applications.

While adapting existing research and solutions to new problem domains is a valid first step, it became clear in this research that traditional distributed computing paradigms, were not designed for nor suited to the challenges of MANETs. Therefore, we proposed a new communication model for MANETs called MELON. MELON provides message persistence, basic access controls, on-demand operations, bulk message retrieval, FIFO message ordering, and support for simple message streaming. We believe MELON is well-suited to supporting MELON applications while offering a simple and easily implemented set of operations.

We have examined several case studies using MELON to implement applications and found it well-suited to provide convenient and efficient communication for MANET applications. To evaluate MELON, we again implemented the traditional paradigms in order to share a common codebase with MELON and provide a fair comparison. As part of the evaluation, we implemented an experiment coordination framework implemented using MELON itself. In our experiments, the prototype implementation of MELON performed well in comparison to the existing paradigms, suggesting it is a viable approach to MANET communication.

## 8.1 Future Work

This dissertation presents only the initial design of a MANET communication paradigm. Although the prototype implementation performed well, it is implemented naively. In particular, it does not attempt to leverage multicast communication, which may improve performance. Very little investigation has been performed in determining appropriate policies for message replication and garbage collection.

Security is a growing concern, especially for wireless communication. In this dissertation, we defined private communication as that which cannot be accessed by unauthorized nodes from within the communication paradigm. However, this ignores the reality of how easily it is to eavesdrop on wireless communication. Naturally encrypted connections are desired, but verifying identity of nodes in a decentralized network is a challenge. Relatedly, while MELON includes a proposal for "directed messages" which can only be read by their addressee, we have not provided any mechanism for assigning and verifying identities.

As mobile devices become more accessible and open, we hope to see MELON adapted to work on consumer mobile devices such as smartphones to enable the development of more decentralized applications working together in mobile ad hoc networks.

# APPENDIX A

# Full Application Examples

The following sections provide a demonstration of working code using the current MELON library implementation in JRuby.

## A.1 News Server & Reader Applications

Listing A.1: News Reader

```
require "melon"

# Initialize MELON
melon = Melon.with_zmq

# Initialize potential topics
topics = ["Politics", "Sports", "Business", "Technology", "Local"]

i = 0

loop do
  # Choose a topic
  topic = topics.sample

  # Generate headline for topic
  message = [topic, "#{topic} news item #{i+=1}"]

  # Write message
  melon.write message
```

```ruby
  # Pause for a few seconds
  sleep rand(5)
end
```

Listing A.2: News Reader

```ruby
require "melon"

unless ARGV[2]
  abort "news_reader.rb ADDRESS PORT TOPIC"
end

address = ARGV[0]
port = ARGV[1]
topic = ARGV[2]

# Initialize MELON
melon = Melon.with_zmq

# Add address of a news server
melon.add_remote port, address

# Set the template to retrieve a headline for the
# given topic
template = [topic, String]

# Read all relevant messages as they become available
loop do
  puts melon.read_all template
end
```

## A.2 Chat Application

This example splits the implementation of a chat application into a library which provides most of the functionality and a small script to set up the environment for the user.

Listing A.3: Chat Library

```
# Encapsulate chat communication
class Chat
  def initialize name
    @name = name
    @melon = Melon.with_zmq
  end

  def add_remote port
    @melon.add_remote port
  end

  # Write out a chat message
  def chat message
    @melon.write [@name, message]
  end

  # Get all unseen messages
  def read_messages
    @melon.read_all [String, String]
  end

  # Print out all messages except our own
  def print_messages messages
    messages.each do |name, message|
      unless name == @name
        puts "\n<#{name}> #{message}"
      end
    end
  end

  # Read and show messages in a separate thread
  def monitor
    Thread.new do
      loop do
        print_messages read_messages
      end
    end
  end

  # Main loop for chatting
  def start
    monitor

    # Send chat messages from user
    loop do
      print "? "
      message = gets.strip

      unless message.empty?
        chat message
      end
    end
  end
end
```

The chat client in Listing A.4 is a simple script to set up a chat client that talks to other local chat clients. This makes it simple to try on a single machine.

Listing A.4: Local Chat Client

```
require "melon"
require "chat"
```

```
# Get user's name
print "Name: "
name = gets.strip

# Initialize chat library
chat = Chat.new name

# Add processes running on the same machine but different ports
loop do
  print "Remote port: "
  chat.add_remote gets.strip.to_i

  print "Add another (y/n)? "
  break unless gets.downcase.start_with? "y"
end

# Start chatting
chat.start
```

# APPENDIX B

# MELON Prototype Implementation Details

This appendix describes the prototype implementation of MELON in more detail.

## B.1 Local Storage

Each application manages its own local storage, implemented as the `LocalStorage` class. Each instance of `LocalStorage` consists of two dynamic arrays: one for read-only messages and one for take-only messages. The class provides methods for storing, retrieving, and reading messages from these arrays.

When a new message is added to the store a new `StoredMessage` object is created. The local storage generates a incremental ID for the new `StoredMessage`. Once it obtains an exclusive lock for the appropriate array, the message is added to the end of the array.

Retrieving a take-only message involves scanning the take-only array for a matching message. If one is found, the scan stops. The matching message is deleted from the array and the message is returned. If not matching message is found, the method returns `nil`. When scanning, the mutex associated with the take-only array is locked.

Finding a read-only message is a little more involved because the local storage must avoid returning any messages which have already been read. As it scans, it only attempts matching messages which are not included in the provided `read_messages` data structure. Unlike the take-only removal, many reads may

Table B.1: LocalStorage Methods

| **Method** | **Input** | **Output** | **Description** |
|---|---|---|---|
| store | message | | Stores a take-only message |
| write | message | | Stores a read-only message |
| find_and_take | template | message | Removes matching take-only message |
| take_all | template | messages | Removes all matching take-only messages |
| find_unread | template, read_messages | message | Returns matching read-only message |
| find_all_unread | template, read_messages | messages | Returns all matching read-only messages |

occur at the same time. For this reason, a readers/writers lock is used for accessing the read-only array. This allows multiple readers but only a single writer to access the array.

Bulk operations are essentially the same, except all matching messages are returned instead of just the first matching.

### B.1.1 Messages

In the prototype, messages are arrays which may contain any assortment of values. Templates are also arrays, but if a value is a class it will be matched against the class of the value in the message. Please note MELON may be implemented with any message scheme that allows for matching based on some kind of templates, this was just a simple approach used in the prototype implementation.

### B.1.2 Stored Messages

When a message is stored, it is saved in a `StoredMessage` which holds an ID and a *copy* of the message. This prevents issues if the message is modified after being stored.

The `StoredMessage` class implements a simple message matching algorithm. First, if the message is not the same length as the template, clearly they do not match. Otherwise, the template and the message are compared value by value. If the template value is a class, the message value is checked to see if it is an object of the same class or a subclass. Otherwise, the values are compared via equality. The matching aborts on the first mismatch.

`StoredMessage` also implements the mapping of process ID and message ID to a single integer as described in Equation 5.1, resulting in the identifier used for tracking the message in the read message data structures.

## B.2 Remote Storage Client

All network communication in the prototype is implemented using ZeroMQ.

Each remote storage client, implemented in the `RemoteStorage` class, is associated with a particular remote node specified by an IP address and port. `RemoteStorage` offers exactly the same retrieval API as local storage, so the MELON API implementation may treat local storage and remote storage in the same manner.

When a method is called on a `RemoteStorage`, it serializes the method name, template, and read messages (if a read-only action) and sends them to the remote node. If the remote node is unavailable, the communication times out and the method returns `nil`. Choosing a timeout value presents a trade-off. Lower values allow operations to return faster if the remote node is unavailable, but then the communication is less reliable. Higher values are slower, but may provide greater opportunities for connections. In our implementation we allowed 1.5 seconds for sending to the remote node and 5 seconds for receiving the reply.

When a response is received, it is deserialized and returned to the caller.

## B.3 Remote Storage Server

The storage server as implemented in the `StorageServer` class, is the most complicated piece of the prototype because it must manage multiple concurrent connections. To do so, it manages a thread pool of workers to accept connections. As remote nodes connect, the connections are handed to the workers over an inter-process connection using ZeroMQ. The worker then deserializes the message and calls the appropriate method on the local storage. Assuming the message is valid, the result from the local storage is again serialized and returned to the remote node.

In the implementation, the server provides a ZeroMQ "ROUTER" socket to which the remote nodes connect using a "REQ" socket. Internally, the server provides a "DEALER" socket which each of the workers connects to via a "REP" sockets. Then the sever connects the "ROUTER" socket to the "DEALER" via a queue. This allows multiple incoming requests to be handed off to the workers. Figure B.1 shows the relationship between the different components.

## B.4 MELON API

The application developer only every needs to interact with the MELON API. The implementation of the API maintains a set of servers (local and remote storage) and the set of read messages. The API provides the six MELON operations and a method to add remote servers. The prototype implementation does not provide a mechanism for discovering remote nodes, so these must be added manually.

For `store/write`, the API saves the message directly to the local storage.

For the retrieval operations, the API iterates over the storage servers (local

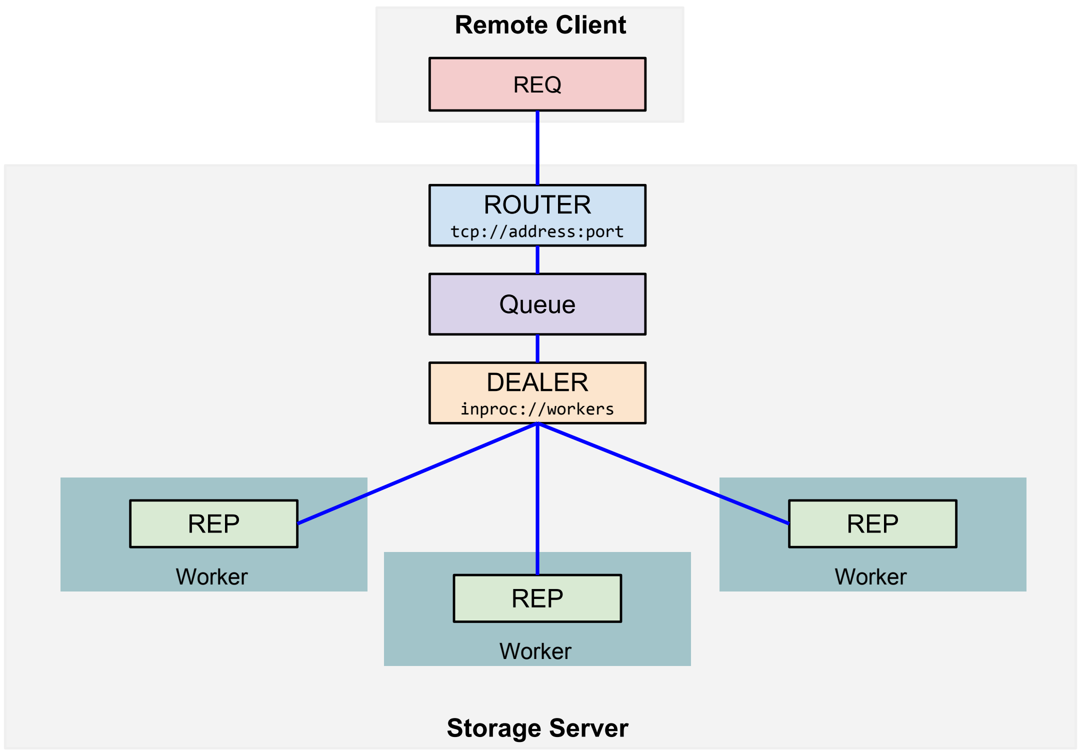


Figure B.1: ZeroMQ Server Setup

and remote) in a random order for each call, invoking the operation on each in turn. When retrieving a single message, the iteration aborts when a matching message is returned. In bulk operations, connections to all servers are attempted and the results returned in one array. If an operation is blocking and no messages are found, there is a brief delay (currently 1 second) then the iteration resumes. If the operation is non-blocking, the servers are only iterated through once. Again, for the prototype implementation remote servers are contacted one at a time, not multicasted.

# References

[Aa09] H. Artail and et al. "The design and implementation of an ad hoc network of mobile devices using the LIME II tuple-space framework." *Wireless Comm., IEEE*, **16**(3):52–59, 2009.

[Bad08] Nadjib Badache. "A distributed mutual exclusion algorithm over multi-routing protocol for mobile ad hoc networks." *IJPEDS*, **23**(3):197–218, 2008.

[Bal07] Davide Balzarotti et al. "The LighTS tuple space framework and its customization for context-aware applications." *Web Intelli. and Agent Sys.*, **5**(2):215–231, 2007.

[Ca01a] Bogdan Carbunar and et al. "CoreLime:: A Coordination Model for Mobile Agents." *Electronic Notes in Theoretical Computer Science*, **54**:17–34, 2001.

[Ca01b] Bogdan Carbunar and et al. "Lime revisited." In *Mobile Agents*, pp. 54–69. Springer, 2001.

[CB10] Justin Collins and Rajive Bagrodia. "A Quantitative Comparison of Communication Paradigms for MANETs." In *7th Intl ICST Conf on Mobile and Ubiquitous Systems (Mobiquitous)*, 2010.

[CDM06] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez, Theo D'Hondt, Theo D'Hondt, and Wolfgang De Meuter. "Ambient references: addressing objects in mobile networks." In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 986–997, New York, NY, USA, 2006. ACM.

[Cer08] Matteo Ceriotti et al. "Data sharing vs. message passing: synergy or incompatibility?: an implementation-driven case study." In *SAC '08: Proc. of the ACM Symp. on Applied Computing*, pp. 100–107, 2008.

[CMB07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. "AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks." In *SCCC '07: Proceedings of the XXVI Intern. Conf. of the Chilean Society of Comp. Sci.*, pp. 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[CP06] Gianpaolo Cugola and Gian Pietro Picco. "REDS: a reconfigurable dispatching system." In *Proc. of the 6th international workshop on Software engineering and middleware*, pp. 9–16. ACM, 2006.

[CPV97] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. "Designing distributed applications with mobile code paradigms." In *Proceedings of the 19th international conference on Software engineering*, pp. 22–32. ACM, 1997.

[CPV07] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. "Is code still moving around? Looking back at a decade of code mobility." In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pp. 9–20. IEEE Computer Society, 2007.

[Da12] Suddhasil De and et al. "A new tuple space structure for tuple space based mobile middleware platforms." In *India Conference (INDICON), 2012 Annual IEEE*, pp. 705–710. IEEE, 2012.

[Den09] Mieso K. Denko et al. "Enhanced cross-layer based middleware for mobile ad hoc networks." *J. Netw. Comput. Appl.*, **32**(2):490–499, 2009.

[ER01] Keith Edwards and Tom Rodden. *Jini example by example.* Prentice Hall PTR, 2001.

[Eug03] Patrick Th. Eugster et al. "The many faces of publish/subscribe." *ACM Comput. Surv.*, **35**(2):114–131, 2003.

[Fa04] Chien-Liang Fok and et al. "A lightweight coordination middleware for mobile computing." In *Coordination Models and Languages*, pp. 135–151. Springer, 2004.

[Ga13] Boix Gonzalez and et. al. "Programming mobile context-aware applications with TOTAM." *Journal of Systems and Software*, 2013.

[GC92] David Gelernter and Nicholas Carriero. "Coordination languages and their significance." *Commun. ACM*, **35**(2):97–107, 1992.

[HAM06] S. Hadim, J. Al-Jaroodi, and N. Mohamed. "Middleware issues and approaches for mobile ad hoc networks." *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, **1**:431–436, 8-10 Jan. 2006.

[HHM07] Klaus Herrmann, Klaus Herrmann, Gero Mühl, Gero Mühl, Michael A. Jaeger, and Michael A. Jaeger. "MESHMdl event spaces - A coordination middleware for self-organizing applications in ad hoc networks." *Pervasive Mob. Comput.*, **3**(4):467–487, 2007.

[Hin13] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications.* O'Reilly, 2013.

[JM96] David B. Johnson and David A. Maltz. "Dynamic Source Routing in Ad Hoc Wireless Networks." In *Mobile Computing*, pp. 153–181. Springer US, 1996.

[JR06] Christine Julien and G-C Roman. "Egospaces: Facilitating rapid development of context-aware mobile applications." *IEEE Trans on Soft. Eng.*, **32**(5):281–298, 2006.

[KB02] Alan Kaminsky and Hans-Peter Bischof. "Many-to-Many Invocation: a new object oriented paradigm for ad hoc collaborative systems." In *OOPSLA '02: 17th Conf. on Object-Oriented Programming, Systems, Langs, and Apps.*, 2002.

[Lan98] DannyB. Lange. "Mobile objects and mobile agents: The future of distributed computing?" In Eric Jul, editor, *ECOOP98 Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pp. 1–12. Springer Berlin Heidelberg, 1998.

[Leg05] Simone Leggio et al. "Session initiation protocol deployment in ad-hoc networks: a decentralized approach." In *2nd Intl. Workshop on Wireless Ad-hoc Networks (IWWAN)*, 2005.

[Lie09] Yao-Nan Lien et al. "A MANET Based Emergency Communication and Information System for Catastrophic Natural Disasters." In *ICDCSW '09: Proc. of the 29th IEEE Intl. Conf. on Distributed Computing Systems Workshops*, pp. 412–417, 2009.

[Ma13] Mary Madden and et. al. *Teens and Technology 2013.* Pew Internet & American Life Project, March 2013.

[MC02] René Meier and Vinny Cahill. "STEAM: Event-Based Middleware for Wireless Ad Hoc Network." In *ICDCSW '02: Proc. of the 22nd Intern. Conf. on Distributed Computing Systems*, pp. 639–644, 2002.

[Mica] Microsoft. http://msdn.microsoft.com/en-us/netframework/.

[Micb] Sun Microsystems. http://java.sun.com/javame/.

[Mur06] Amy L. Murphy et al. "Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents." *ACM Trans. on Software Engin. and Methodology*, July 2006.

[MZ04] M. Mamei and F. Zambonelli. "Programming pervasive and mobile computing applications with the TOTA middleware." *Perv. Comp. and Comm., 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pp. 263–273, 14-17 March 2004.

[Net08] Scalable Networks. "Exata: An Exact Digital Network Replica for Testing, Training and Operations of Network-centric Systems." Technical brief, 2008.

[NKS05a] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. "Programming ad-hoc networks of mobile and resource-constrained devices." *SIGPLAN Not.*, **40**(6):249–260, 2005.

[NKS05b] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. "Programming ad-hoc networks of mobile and resource-constrained devices." *SIGPLAN Not.*, **40**(6):249–260, 2005.

[PR99] Charles E. Perkins and Elizabeth M. Royer. "Ad-hoc On-Demand Distance Vector Routing." In *WMCSA '99: Proc. of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, p. 90, 1999.

[Qua] Qualcomm. http://brew.qualcomm.com/brew/.

[Ra13] Carlos Rodríguez-Domínguez and et. al. "Designing a Communication Platform for Ubiquitous Systems: The Case Study of a Mobile Forensic Workspace." In *New Trends in Interaction, Virtual Reality and Modeling*, pp. 97–111. Springer, 2013.

[RK13] DN Rewadkar and Smita Karve. "Spontaneous Wireless Ad Hoc Networking: A Review." *International Journal*, **3**(11), 2013.

[RW96] Antony Rowstron and Alan Wood. "Solving the Linda multiple rd problem." In *Coordination Languages and Models*, pp. 357–367. Springer, 1996.

[SA14] Eduardo da Silva and Luiz Carlos P Albini. "Middleware proposals for mobile ad hoc networks." *Journal of Network and Computer Applications*, **43**:103–120, 2014.

[SL04] Mee Young Sung and Jong Hyuk Lee. "Desirable Mobile Networking Method for Formulating an Efficient Mobile Conferencing Application." In *Embedded and Ubiquitous Computing*, 2004.

[Smi13] Aaron Smith. *Smartphone Ownership - 2013 Update*. Pew Internet & American Life Project, June 2013.

[SSH07] Jing Su, James Scott, Pan Hui, Jon Crowcroft, Eyal De Lara, Christophe Diot, Ashvin Goel, Meng How Lim, and Eben Upton. *Haggle: Seamless networking for mobile applications*. Springer, 2007.

[Sus04] et.al. Sushil K. Prasad, Vijay Madisetti. "SyD: a middleware testbed for collaborative applications over small heterogeneous devices and data stores." In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pp. 352–371, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[Szu06] Matthew Szudzik. "An elegant pairing function." In *Wolfram Science Conference*, 2006.

[TMB01] M. Takai, J. Martin, and R. Bagrodia. "Effects of wireless physical layer modeling in mobile ad hoc networks." In *MobiHoc '01: Proc. of the 2nd ACM Intl. Symp. on Mobile Ad Hoc Networking & Computing*, 2001.

[TS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems*, pp. 700–701. Prentice Hall, 2002.

[VB04] M. Varshney and R. Bagrodia. "Detailed models for sensor network simulations and their impact on network performance." In *MSWiM '04: Proc. of 7th ACM Intl. Symp. on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2004.

[Wad99] Stephen Paul Wade. *An investigation into the use of the tuple space paradigm in mobile computing environments.* PhD thesis, Citeseer, 1999.