Mobile Application Development with MELON

Justin Collins and Rajive Bagrodia

University of California Los Angeles, Los Angeles CA, USA,
{collins, rajive}@cs.ucla.edu

Abstract. Developing distributed applications for mobile ad hoc network continues to be challenging due to the dynamic and unpredictable nature of MANETs. MELON is a general purpose coordination language designed to provide flexible communication patterns for MANET applications while remaining lightweight. Based on a distributed shared message store, MELON abstracts network communication to an asynchronous exchange of persistent messages. MELON simplifies application development by supporting read-only and remove-only messages, bulk message retrieval, and per-host ordering of messages. In this paper, we review the MELON programming model, demonstrate its utility for writing MANET applications, and quantitatively compare it to traditional distributed computing paradigms in a MANET context. For a shared whiteboard application, we find MELON achieves 100% message delivery with 95% less latency than tuple spaces while also maintaining per-host message ordering.

1 Introduction

Tiny, powerful personal computers are quickly becoming ubiquitous. In the United States, 91% of adults have cell phones, and 56% of those are smartphones[1]. Among teens, 78% have a cell phone, of which 37% are a smartphone[2]. 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. Developing MANET applications presents a number of challenges with the primary difficulty being the unpredictably dynamic infrastructureless wireless network itself. Unlike wired networks, failures in MANETs are commonplace instead of exceptional. Nodes may join and leave the network at any time and the network topology is in constant flux. Networked applications must be aware of and handle the challenging nature of MANETs to be effective.

Several language, middleware, and library solutions to dealing with MANET communication have been proposed to assist in developing MANET applications. The majority of these proposals are based on traditional distributed computing paradigms[3]. Of these, publish/subscribe, remote procedure calls, and tuple spaces are commonly used. However, these communication paradigms were originally designed for stable, wired networks or even interprocedural communication on a single machine. While they have been adapted to MANETs, their original designs are limiting and not well-suited to the MANET environment[4].

To address the need for a communication paradigm designed to operate in MANETs, we have proposed a new coordination language called MELON[5]. MELON provides a flexible, general-purpose communication abstraction which may be completely distributed. In addition to basic message exchange, MELON also implements bulk message retrieval, provides basic message access control, enforces per-host ordering of messages, and supports message streams.

In this paper we describe the MELON paradigm, compare implementing a shared whiteboard application with publish/subscribe, RPC, tuple spaces, and MELON, and finally compare the performance of each whiteboard implementation. We demonstrate MELON is well-suited for developing MANET applications, and while slower than publish/subscribe and RPC, it is the only paradigm to deliver 100% of sent messages. MELON also delivered whiteboard messages with 95% less latency and 60% more in-order messages compared to tuple spaces.

Section 2 reviews the traditional paradigms used in this paper. In Section 3 we discuss the design of MELON, the operations it provides, and our prototype implementation. We compare a whiteboard implemented each paradigm in Section 4 and then measure performance of each paradigm in Section 5 before presenting our conclusions in Section 6.

2 Related Work

This section offers a brief overview of the three paradigms we will compare with MELON. Further surveys of middleware, languages, and communication paradigms for MANET development can be found in [3] and [6].

The *publish/subscribe* paradigm divides processes into publishers and subscribers. In topic-based publish/subscribe, publishers send messages tagged with a topic. 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. It is also possible to handle multiple incoming publications concurrently. 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. 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[7], although some implementations exist[8]. Managing an overlay network of brokers also adds considerable complexity.

Remote procedure calls (RPC) is a distributed programming paradigm which disguises remote communication as local method calls. 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[9].

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.

Tuple spaces, introduced in the Linda[10] coordination language, operate on a distributed shared memory space of 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 nonblocking versions are available.

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. 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 and each process can request tuples by the counter value in order. Multiple processes producing tuples must coordinate to generate consistent counters. A third option proposed in [11] is to introduce a **copy-collect** operation which copies all matching tuples.

Several projects have adapted tuple spaces to MANETs, including L^{2}imbo[12], LIME[13], TOTAM[14], and EgoSpaces[15]. LIME, a popular implementation, relies on explicit join and leave operations to federate distributed tuple spaces. This is at odds with the frequently unexpected disconnections in MANETs. [16] discusses the difficulties LIME encounters with tuple space semantics, including situations that can lead to livelocks. LIME II[17], Limone[18], and CoreLIME[19] are proposed to meet shortcomings in LIME.

3 MELON Design

MELON borrows the idea of a distributed shared message store from tuple spaces. The concept of shared message collections which may be safely manipulated by many processes fits well in a MANET context. To send a message, a process using MELON persists it in a globally shared "store". In practice these messages are stored on the client which outputs them in order maintain atomicity of removal without global state or locking. Messages are retrieved by matching against templates describing message content. This decoupling between sending and receiving is beneficial in a MANET context where maintaining connections between hosts can be challenging. With MELON and tuple spaces, disconnections do not cause lost messages or disrupt operations.

Aside from the shared message store, MELON departs significantly from typical tuple spaces in its operations and semantics. In a tuple space, any message may be read or removed by any process, and any matching message may be retrieved in any order. In contrast, MELON divides the messages into two pools:

a remove-only pool, and a read-only pool. Remove-only messages can only be retrieved once and must be removed when retrieved. Read-only messages may never be explicitly removed, only be copied from the message store. Matching messages are returned in FIFO order per host, matching the reliable FIFO-ordered multicast semantics in [9].

MELON contains some additional minor differences from tuple spaces. First, messages are not required by the paradigm to be tuples, but may be implemented as any structure which can be matched by a template (e.g., messages could be unordered tuples with named values instead). Secondly, MELON explicitly acknowledges the storage limitations of mobile devices. In any communication paradigm which persists messages, there is a limitation to how many messages may be stored. MELON attempts to mitigate this limitation by allowing messages to be automatically garbage collected. The alternatives are to not store new messages or to allow applications to exhaust available memory.

A last deviation from tuple spaces is the removal of strict semantics for returning messages. In tuple spaces, the semantics stipulate that if a matching message exists, it must be returned for a retrieval operation. In the reality of MANETs, this semantic cannot be met, so in MELON all retrieval operations are limited to best-effort.

These differences were introduced in MELON to both relieve the application developer of certain responsibilities and to allow the paradigm to operate well in a MANET. For example, read-only messages prevent a badly-behaved process from removing important messages meant to be read by many processes, and FIFO ordering is especially convenient in applications where most messages are generated by a single host and the ordering is important, such as news feeds or streaming video. MELON is also deliberately designed to avoid any global state and enable completely distributed implementations.

3.1 MELON Operations

Processes in MELON communicate by storing messages to a distributed shared message store and retrieving the messages based on templates. In this paper, we assume messages consist of an ordered list of typed values. However, as noted above, MELON does not limit how messages might be constructed. 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.

Operations are split into read-only (**write**/**read**/**read_all**) and take-only (**store**/**take**/**take_all**) operations. Each operation is represented here as a separate function call. **store** and **write** operations return null values as soon as the saved message is available in the message store (essentially immediately). **take** and **read** operations block by default until a matching message is returned, but may be set to nonblocking on a per-call basis. If a nonblocking operation finds no matching messages, an empty set is returned.

Table 1. MELON Operations

Operation	**Return Type**	**Action**
store(*message*)	*null*	Store removable message
write(*message*)	*null*	Store read-only message
take(*template*, *[block = true]*)	*message* or *null*	Remove and return message
read(*template*, *[block = true]*)	*message* or *null*	Copy and return read-only message
take_all(*template*, *[block = true]*)	*array*	Bulk remove messages
read_all(*template*, *[block = true]*)	*array*	Bulk copy read-only messages

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.

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 stored with either operation may be automatically garbage collected.

A **take** operation accepts a message template as the first argument and an optional boolean indicating blocking or nonblocking 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. Once a message has been returned by a **take** operation, it may not be returned by a subsequent operation in any process.

The **read** operation accepts the same arguments but 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 that process. A message may be **read** by any number of processes, but only once per process.

Table 2. Read from multiple processes

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

Table 2 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 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** and **read_all** operations are used to retrieve a group of matching messages instead of a single message. Otherwise, the semantics match **take** and **read**: **take_all** can only remove messages from **store** operations, and **read_all** only returns unread messages from **write** operations. Table 3 demonstrates a use of **read_all**. One or more processes generate news messages containing a news category and headline. 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. Any number of processes can consume the news as readers. The `fetch` method in Table 2 uses **read_all** to return all news items in a given category. Repeated calls to `fetch` will only return news items not previously read.

Table 3. News server and reader

News Server	News Reader

`def report(category, headline)`
`   write [category, headline]`
`end`
	`def fetch(category)`
	`   return read_all([category, String])`
	`end`

By default, all retrieval operations will block the application until at least one matching message is found. The operations can also be performed in nonblocking mode, in which case **take** and **read** return null when no matching message is found, while **take_all** and **read_all** return empty collections.

3.2 MELON Implementation

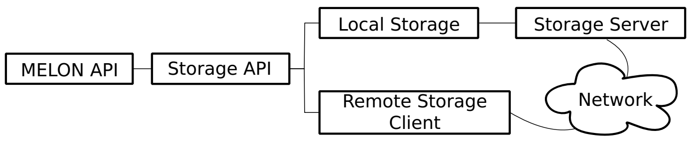

Fig. 1. Paradigm Architecture

We developed a prototype implementation of MELON to validate our design and obtain empirical performance data. The architecture illustrated in Figure 1 is split into five parts. The MELON API is the only interface exposed to the application and provides the six operations described in Section 3. 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.

Local storage is implemented 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 to allow multiple **read** operations to access the array concurrently, but locks the array for **write** operations. The **store/take** array does not permit concurrent operations, as **store** and **take** modify the store. The two arrays may be accessed and modified independently. Searching for matching messages is a linear operation in the prototype implementation. Performance of local operations is explored in [5].

Network communication is handled by ZeroMQ[20], a high performance networking library. For the prototype, the networking was intentionally kept simple. For example, a **read** request queries remote hosts in random order and stops when a matching result is returned. This could possibly be improved using multicast, but it would complicate the implementation by requiring the client to handle multiple asynchronous responses, select one, request the actual message, and handle failure scenarios if the matching message cannot be returned. We traded potential performance gains for simplicity.

For **read** and **read_all** operations, it is necessary to track which messages have been read. Each process maintains its own list of read messages, which it sends with each **read** request. We use a compact sparse bit set to track message IDs and transfer this information efficiently with an average overhead of < 2 bits per ID.

4 Shared Whiteboard Example

A shared whiteboard is a digital document which may be edited and viewed by multiple users concurrently and is commonly given as an example of an application well-suited to MANETs[21]. Shared whiteboards are distributed, real-time, and interactive, which presents some interesting characteristics. Many participants must have access to update the whiteboard, but ordering of changes is very important to maintain a consistent document. Changes should be propagated quickly and reliably so that each user is working with the latest document without missing any updates.

We have implemented a shared whiteboard in MELON along with canonical versions of publish/subscribe, RPC, and tuple spaces using JRuby (a Java implementation of the Ruby language) to compare their features and performance. The programs share 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 to be 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.

4.1 Publish/Subscribe

The publish/subscribe whiteboard in Table 4 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.

Table 4. 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
```

Table 5. 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
```

4.2 RPC

A shared whiteboard implementation using RPC is listed in Table 5. 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.

4.3 Tuple Spaces

Table 7 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, **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.

Table 6. 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
    # Returns only unread figures
    # in per-host order
    figures = @melon.read_all([Figure])

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

Table 7. 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
    # Returns ALL figures
    # in arbitrary order
    figures = @ts.bulk_rd([Figure])

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

4.4 MELON

The MELON whiteboard in Table 6 writes out each figure in a tuple containing just the new figure. It uses the **write** operation since every remote node needs 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 FIFO 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 the host initiated them.

4.5 Summary

All four implementations of the whiteboard have been kept as simple and similar as possible in order to highlight the differences between the paradigms. However, while MELON appears to be as simple to use as the other paradigms, it provides more functionality and guarantees.

Publish/subscribe and RPC are push-based paradigms and allow messages to be received asynchronously by default. However, they do not provide message persistence in their canonical forms. While publish/subscribe is a multicast paradigm by default, RPC must be adapted to perform group communication. Also, RPC must explicitly discover remote objects before invoking remote methods. In both paradigms figures are sent and received singly, although multiple messages may be received concurrently.

Tuple spaces and MELON are pull-based paradigms which provide message persistence. Both tuple spaces and MELON require applications to explicitly use a separate thread to receive messages asynchronously. Tuple spaces do not provide a method to nondestructively read a subset of matching tuples, while MELON does. MELON is also the only paradigm to provide some ordering of messages by definition. Both paradigms allow bulk retrieval of messages, although tuple spaces require an extension to the canonical paradigm. This extension is necessary to allow nondestructive reads of multiple matching tuples and avoid the "multiple read" problem.

5 Quantitative Comparison

For these experiments, we implemented the shared whiteboards as described in Section 4 in the four paradigms using the same codebases as in [5]. To make the comparison as fair as possible, each paradigm shares a considerable amount of common code and utilizes ZeroMQ for network communications.The tuple space implementation uses the LighTS[22] local tuple space library from LIME.

To evaluate the implementations in a MANET environment, we used EXata[23] to provide high-fidelity wireless models and precisely repeatable scenarios while allowing us to run real applications. Our scenario uses 50 nodes with

802.11b radios using AODV moving with random waypoint with a maximum speed of 5m/s in a 500x500 meter area. The two-ray path loss model is used. To measure how the implementations fared in turbulent network conditions, we performed the experiments with increasing levels of packet loss from 1% to 30% as measured by the *ping* command. We used an experiment coordination framework written in MELON itself to manage the applications, running EXata, and collating results.

In our scenario, six real nodes are running the whiteboard application, the rest are simulated and function only as intermediate nodes. For each experiment, each of the six nodes sends 50 whiteboard updates with pauses of 5-10 seconds. This roughly models each user updating their whiteboard at a brisk pace for 4-8 minutes.

5.1 Results

For each implementation, we measured lost messages, 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, 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 globally out of order.

As guaranteed by the paradigm, MELON maintains host ordering in all scenarios while publish/subscribe and RPC do not. For global message ordering, MELON outperforms tuples spaces by 67% and remains within 15% of RPC performance until packet loss reaches 30%. However, this is because MELON continues to achieve 100% message delivery while RPC drops nearly 12% of messages when the network conditions are poor. Reliability does decrease speed, but MELON experiences 95% less latency than tuple spaces while remaining within 3 seconds of RPC in the worst case. While we would like speed, reliability, and perfect ordering of messages for a shared whiteboard, it is better to have the document be eventually consistent than to lose information.

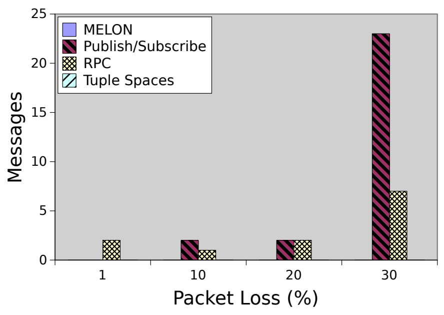

Fig. 2. Host Out-of-Order Messages

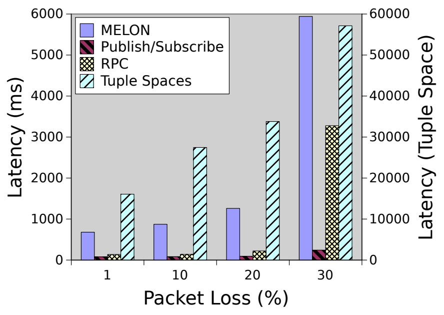

Fig. 3. Message Latency

In our experiments, messages from a single host were generally delivered in the order they were sent as shown in Figure 2. For MELON and tuple spaces, no messages were delivered out of order. Note for 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.

Asynchronous group RPC is used in this application. If one call is delayed, it is possible a subsequent call will complete before a prior one, which is why RPC delivers a small number of messages out of order. Publish/subscribe is fully asynchronous and incoming publications can be processed concurrently. However, even in the worst case publish/subscribe delivers 97.8% of the messages from a host in FIFO order. Like tuple spaces this is the result of the implementation: 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 4. 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 any time 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 3, 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 3.

MELON and RPC provide about the same global ordering, although MELON is more affected when the network conditions worsen. This is likely due to MELON's reliable message delivery (Figure 5), as messages may be delayed significantly by broken routes or network partitioning. In contrast, losing messages improves ordering since a message not delivered cannot be out of order. MELON is the only paradigm to demonstrate 100% message delivery. Tuple spaces are expected to be reliable, but this application requires delivery of large numbers of messages. Since the median latency for tuple spaces reached a full minute, the experiment completed before all messages arrived.

While low delivery rates for publish/subscribe have been observed previously[4], here it performs well in the lossy environment due to quick delivery rates, but still dropped 1.4% of messages when the network connectivity was good. RPC never achieves 100% delivery rates and declines as the network degrades. In group RPC, clients cannot be aware of how many receivers may be available and therefore does not attempt to retry calls after timeouts. Syn-

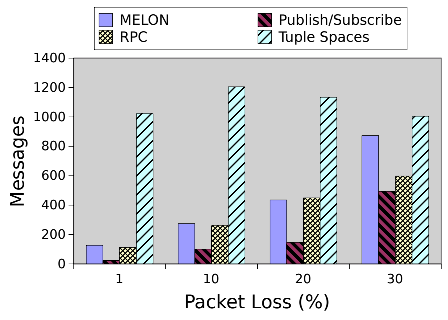

Fig. 4. Global Out-of-Order Messages

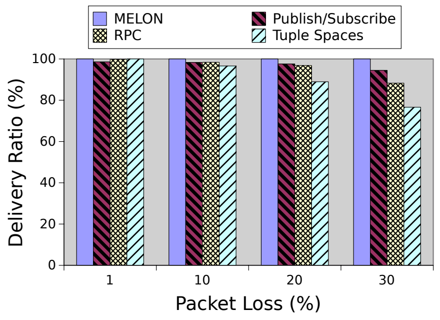

Fig. 5. Delivery Rates

chronous RPC would block the process until the message is delivered, but deliveries would be considerably delayed which is unacceptable for this application.

Median time between sending and receiving a message is reported in Figure 3. 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, as it requires no message confirmations nor active discovery of remote hosts. RPC was also quite fast until it was disrupted 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. Since dropped messages do not count towards median latency, a lossy communication paradigm can appear very fast. MELON provides better reliability and therefore is a bit slower as the network becomes less reliable and more delivery attempts are required. Frequency of the pull attempts are another trade-off that pull-based paradigms must make. 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 delivery, but higher overall network usage, collisions and network monopolization.

6 Conclusions

Traditional distributed computing paradigms were not designed to operate in dynamic, self-organizing MANETs where disconnections and topology changes are frequent. In this paper we have introduced the MELON coordination language and compared it qualitatively and quantitatively to traditional distributed computing paradigms which have been adapted to MANETs.

In our shared whiteboard experiments, MELON was the only paradigm to deliver 100% of sent messages. It also maintained 100% host ordering of messages with 95% less latency than tuple spaces and 67% more globally in-order messages, without additional complexity for the application developer. These results indicate MELON can serve as a practical approach for distributed communication in MANET applications while providing persistent messages, reliable FIFO-ordered multicast, efficient bulk retrieval, and simple message streaming.

References

1. Aaron Smith. *Smartphone Ownership - 2013 Update.* Pew Internet & American Life Project, June 2013.
2. Mary Madden and et. al. *Teens and Technology 2013.* Pew Internet & American Life Project, March 2013.
3. Justin Collins and Rajive Bagrodia. Programming in mobile ad hoc networks. In *WICON '08: 4th Intl. Conf. on Wireless Internet*, pages 1–9. ICST, 2008.
4. Justin Collins and Rajive Bagrodia. A quantitative comparison of communication paradigms for manets. In *7th ICST Conf on Mobile and Ubiq. Sys. (Mobiquitous)*, 2010.
5. Justin Collins and Rajive Bagrodia. Melon: A persistent message-based communication paradigm for manets. In *10th ICST Conf on Mobile and Ubiq. Sys. (Mobiquitous)*, 2013.
6. S Hadim and et al. Trends in middleware for mobile ad hoc networks. *Journal of Communication*, 1(4), 11-21 July 2006.
7. Patrick Th. Eugster and et al. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
8. Gianpaolo Cugola and Gian Pietro Picco. Reds: a reconfigurable dispatching system. In *Proc. of the 6th international workshop on Software engineering and middleware*, pages 9–16. ACM, 2006.
9. Maarten van Steen Andrew S. Tanenbaum. *Distributed Systems*, pages 375–381, 390. Prentice Hall, 2002.
10. David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
11. Antony Rowstron and Alan Wood. Solving the linda multiple rd problem. In *Coordination Languages and Models*, pages 357–367. Springer, 1996.
12. Stephen Paul Wade. *An investigation into the use of the tuple space paradigm in mobile computing environments.* PhD thesis, Citeseer, 1999.
13. Amy Murphy and et al. Lime: A coordination middleware supporting mobility of hosts and agents. *ACM Trans on Soft. Eng. and Meth.*, 15(3):279–328, July 2006.
14. Boix Gonzalez and et. al. Programming mobile context-aware applications with totam. *Journal of Systems and Software*, 2013.
15. 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.
16. Bogdan Carbunar and et al. Lime revisited. In *Mobile Agents*, pages 54–69. Springer, 2001.
17. 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.
18. Chien-Liang Fok and et al. A lightweight coordination middleware for mobile computing. In *Coordination Models and Languages*, pages 135–151. Springer, 2004.
19. Bogdan Carbunar and et al. Corelime:: A coordination model for mobile agents. *Electronic Notes in Theoretical Computer Science*, 54:17–34, 2001.
20. Pieter Hintjens. *ZeroMQ: Messaging for Many Applications.* O'Reilly, 2013.
21. DN Rewadkar and Smita Karve. Spontaneous wireless ad hoc networking: A review. *International Journal*, 3(11), 2013.
22. Davide Balzarotti and et al. The lights tuple space framework and its customization for context-aware applications. *Web Intelli. and Agent Sys.*, 5(2):215–231, 2007.
23. Scalable Networks. Exata: An exact digital network replica for testing, training and operations of network-centric systems. Technical brief, 2008.