MELON: A Persistent Message-Based Communication Paradigm for MANETs

Justin Collins and Rajive Bagrodia

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

Abstract. In this paper we introduce MELON, a new communication paradigm tailored to mobile ad hoc networks, based on novel interactions with a distributed shared message store. MELON provides remove-only, read-only, and private messages, as well as bulk message operations. The dynamic nature of MANETs is addressed with persistent messages, completely distributed message storage, and flexible communication patterns. We quantitatively compare a prototype implementation of MELON to existing paradigms to show its feasibility as the basis for new MANET applications. Experiments demonstrate 40% better throughput on average than traditional paradigms, as well as 70% faster local insertion and removal operations compared to an existing tuple space library.

1 Introduction

While smartphones are quickly becoming ubiquitous, most mobile applications continue to use a client-server model rather than communicating through mobile ad hoc networks (MANET). One reason may be the added challenges of developing a MANET application which must communicate with peers over unreliable shifting network topologies. While communicating over a single-hop wireless network (either a WiFi access point or cellular tower) to a central server is simpler, MANETs are useful when communication is between nearby devices (avoiding data charges, for example) or when there is no network infrastructure, such as in disaster recovery situations. Even less dramatic circumstances such as locations without cellular signals can also benefit from MANETs.

To alleviate the application development challenges posed by MANETs, several approaches to middleware and libraries have been proposed. The majority of these proposals are adapted forms of traditional distributed computing paradigms such as publish/subscribe [1], remote procedure calls [2], and tuple spaces [3], instead of MANET-focused paradigms.

In this paper, we introduce a new paradigm called MELON[1]. MELON overcomes frequent network disconnections in MANETs by providing message persistence in a distributed shared message store and can operate entirely on-demand, avoiding coupling between nodes. New communication primitives in MELON

[1] Message Exchange Language Over the Network

offer remove-only, read-only, and private messages, as well as bulk transfer of messages. MELON also simplifies communication by returning messages in per host write order, instead of entirely nondeterministically. We demonstrate that our proposed paradigm is practical by comparing performance of a prototype MELON implementation to canonical implementations of traditional paradigms in a MANET environment. Results show higher throughput with comparable overhead and latency.

The rest of this paper is organized as follows: Section 2 lists the MANET challenges we address; Section 3 discusses the design and operations of MELON; Section 4 provides details of the MELON implementation; in Section 5 we quantitatively compare MELON to existing paradigms; Section 6 references related work; and finally Section 7 presents our conclusions.

2 MANET Application Challenges

Applications which are developed for MANETs must operate in an infrastructureless, unreliable, and dynamic distributed environment. This presents a particular combination of challenges which should be addressed by a development platform. In [4] we identify disconnection handling, addressing and discovery, and flexible communication as important features for MANET development.

Wireless communication can be disrupted in many ways, including competing broadcasts, physical obstacles, and nodal mobility. When combined with an entirely self-organizing network where nodes may join and leave the network at any time, this environment leads to frequent communication disruptions. In traditional networking libraries, disconnections are treated as exceptional events which must be handled by the application. In MANETs, disconnections are so common they should be handled naturally by the programming paradigm.

The dynamicity of MANETs also leads to transience of resources. It is common in distributed programming paradigms to refer to resources independently from the physical location of the resources. This also works well in MANETs, since a resource is likely to be mobile, and several hosts may offer the same type of resource over time.

Before addressing resources an application must discover the resources which are available. The infrastructureless nature of MANETs precludes the use of centralized architectures to maintain a directory of resources. It is also unlikely the location of resources will be known prior to joining a network. Distributed discovery mechanisms are needed to find resources in a MANET environment.

As the primary purpose of a programming paradigm for MANETs is communication between hosts, a general-purpose paradigm should provide flexible communication: both multicast and unicast communication are common in MANET applications. Given the ubiquity of SMS, instant messaging, and direct messages, the paradigm should also support private unicast communication.

It should also be noted that devices in MANETs are often resource constrained. Smartphones have become essentially ubiquitous in many locales but have much less power, CPU, memory, and storage space than a typical desktop

computer. These constraints influence the design of paradigms intended for these devices.

3 MELON Design

The design of MELON is centered around a distributed shared message store. Each device in the network may host any number of applications which access and contribute to the shared message store. Each application hosts a local message store which may be accessed by any other local or remote application. Applications request messages (which may be local or remote) using message templates.

By communicating through a shared message store, the concept of connections between hosts is eliminated and thus disconnections are no longer an application layer concern. Hosts suddenly leaving the network does not disrupt an application and applications do not need to handle operations failing from intermittent network connectivity or physical wireless interference. The application is insulated from these issues by the semantics of the operations.

Messages are sent and received asynchronously by storing and retrieving them from the shared message store, removing the need for a persistent connection. This provides temporal decoupling between hosts, since messages can still be delivered even after prolonged disconnections.

The dynamic network topology of MANETs creates a challenge which maintaining any type of logical or overlay network, so MELON does not rely on a particular network structure. Discovery of available messages is performed on-demand for each operation. While this does increase the amount of communication required for each operation, it avoids global state and allows the network to change at any time.

MELON also provides spatial decoupling by matching messages based on content, instead of 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.

The shared wireless communication medium in MANETs is well-suited to 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 also require unicast communication. While unicast communication can be accomplished by storing regular messages in MELON, these can be disrupted by a process removing a message intended for a different receiver. It is also possible to eavesdrop on messages by reading but not removing a message. For applications such as instant messaging, it is important to have private unicast communication. In MELON, messages may be directed to a specific receiver to ensure the messages are only taken by the intended recipient.

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.

Finally, MELON differentiates between messages intended to persist and be read by many receivers versus messages expected to be removed from the message store. For example, messages in a news feed would have many readers but messages should not be removed. In contrast, in a job queue each job should be removed by exactly one worker. MELON provides operations to support both scenarios.

3.1 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 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.

3.2 Operation Details

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 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.

Operation	Return Type	Action
store(*message*, *[address]*)	*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

Table 1: MELON Operations

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 address is provided, then only the host with a matching identity can remove the message.

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. Once a message has been returned by a **take** operation, it is removed from the message store and may not be returned by a subsequent operation in any process.

The **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 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: News server and reader

News Server	**News Reader**

`function report(category, headline) {`
`  write([category, headline])`
`}`
	`function fetch(category) {`
	`  return read_all([category, String])`
	`}`

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 seen.

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.

4 MELON Implementation

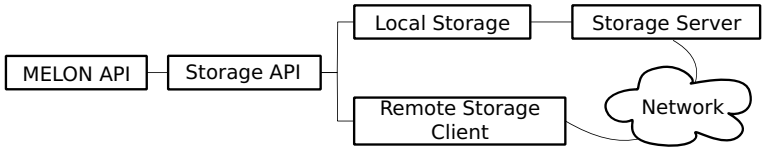


Fig. 1: Paradigm Architecture

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 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.

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 in parallel, 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.

Network communication is handled using ZeroMQ [5], 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, 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 trade 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 transfer this information efficiently. We measure this overhead in Section 5.3.

5 Quantitative Evaluation

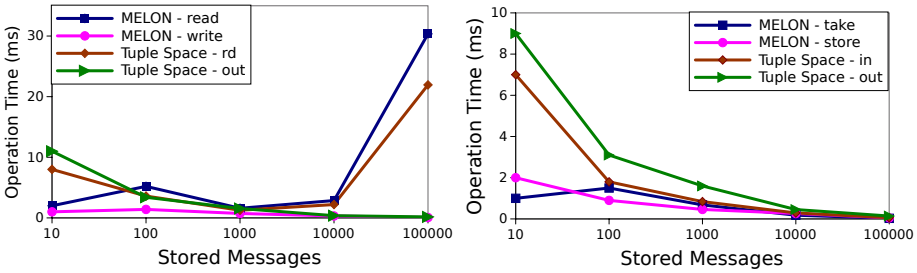
To determine if MELON is a feasible solution for actual MANET applications, we chose to compare its performance to canonical implementations of publish/subscribe, RPC, and tuple spaces. If a prototype implementation of MELON performs at least as well as existing paradigms, then it is likely to be useful in actual practice. In the experiments below, we compare speed of operations, message overhead, throughput, and latency in a MANET context.

5.1 Experimental Setup

In order to judge the performance of MELON, we used applications written using the prototype MELON implementation and evaluated them using the EXata network emulator [6]. Using an emulator allowed us to run real applications but also have precisely repeatable environments with high fidelity network models.

We use a single scenario with 50 nodes distributed in a 150m square grid which move using a random waypoint mobility model. Signal propagation is limited to 50m in order to match an indoor environment and force multihop routes, and the two-ray model is used for path loss. Every run of the mobile scenario uses the same random seed so the mobility pattern is identical. 802.11b WiFi is used with the AODV [7] routing protocol. In the mobile experiments, node speed is varied from 0 to 20 meters/second with a 30s pause time, resulting in increasing packet loss as measured with the ping application.

5.2 Operation Speed



(a) Read Speed

(b) Take Speed

Fig. 2: Operation Speeds

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 [8] 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 LighTS and MELON messages are stored in arrays. Operations linearly search the arrays in $O(nm)$ time, where n is the length of the message and m is the number of stored messages. Operations returning messages near the beginning of the array are fastest, while the slowest operation returns the final message. In our experiments, this cost did not become apparent until searching 100,000 messages when average time per operation increased $\tilde{9}$x for LighTS and $\tilde{1}0$x for MELON, with total read time taking just under a minute. MELON is slightly slower since it also compares messages to the "read" list.

On the other hand, removing messages is fast since the matching message is always the first message in the store. All **take**/**in** operations require less than $8ms$ 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 naturally 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 fast enough for typical applications, since storing a message takes less than $10ms$ on average, and usually less than $4ms$.

5.3 Communication Overhead

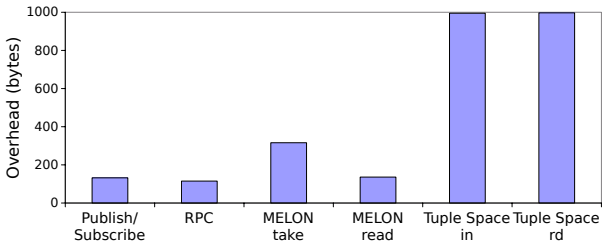


Fig. 3: Message Overhead

For any communication library or framework, increased message size is an important factor in determining usefulness. We average the number of bytes actually sent over the network per message and subtract the 1KB application payload to determine overhead as shown in Figure 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 a topic. 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.

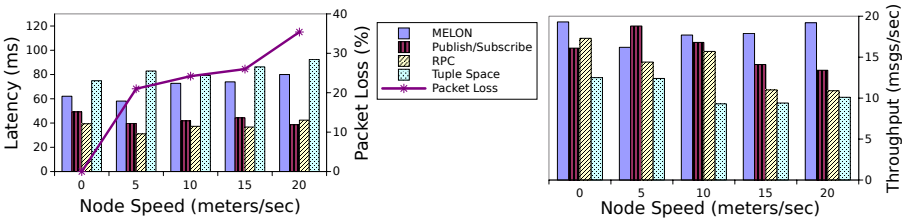
We again use the LighTS tuple space implementation to determine overhead. The serialized versions of tuples and tuple templates are very large and must be sent for each request. If simpler data structures were used, overhead would be expected to be similar to MELON's overhead for **take**.

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, which increases as the number of read messages increases.

5.4 Message Latency

Figure 4a shows the average operation latency between a client's request for a message and the receipt of the message. In these experiments, a single host writes out 1,000 messages with a 1KB payload, and the other hosts read the messages concurrently. Tuple spaces and MELON use the **rd**/**read** operations to retrieve the messages one at a time. As publish/subscribe does not involve a "request" beyond the initial operation, latency was measured as the time elapsed between receiving sequential publications.

In these experiments, MELON and tuple spaces were the most affected by the increase in node speed and packet loss, as well as having the highest latency when nodes were at rest. MELON latency increased 29% and tuple spaces increased 24%. In contrast, RPC only increased 7% and publish/subscribe actually had the lowest latency at the highest node speed. Since publish/subscribe is strictly push-based and has very low overhead, it is able to take advantage of the increased opportunities for transferring data that occur when nodes move quickly in a confined space while being less affected by packet loss. On the other hand, MELON and tuple spaces have high overhead and must continually attempt to request messages from remote nodes.



(a) Message Latency (b) Message Throughput

Fig. 4: Paradigm Performance

5.5 Message Throughput

Throughput was measured on the receiver side in messages delivered per second. 1,000 messages with a 1kb payload are output by one host, while the other hosts

read the messages one at a time. Figure 4b shows the average throughput with varying node speeds. Packet loss is identical to Figure 4a.

Tuple spaces perform the worst, varying from 12.5 to 10.1 messages per second. Given the large overhead, tuple spaces is more affected by packet loss than the other paradigms. MELON provides the best performance in this scenario, delivering between 19.2 and 16.2 messages per second. Both tuple spaces and MELON actually improve slightly as node speed increases from 5 to 20 meters per second. Although higher node speed causes more disconnections, it also increases the likelihood nodes will be near each other during the experiment.

Publish/subscribe performs well at moderate speeds (18.8 msgs/s at 5 m/s), but since lost messages are lost forever packet loss reduces the number of delivered messages, throughput drops down 29% to 13.4 msgs/s at 5 m/s.

6 Related Work

The concept of a distributed shared message store is based on the idea of tuple spaces introduced by the Linda [3] language. Tuple spaces have been adapted to MANETs in LIME [9], MESH*mdl* [10], TOTA [11], and EgoSpaces [12].

The original version of LIME relies on explicit join and leave operations to federate distributed tuple spaces, which is at odds with the frequently unexpected disconnections in MANETs. [13] discusses the difficulties LIME encounters when attempting to implement tuple space semantics, including situations that can lead to livelocks. LIME II [14], Limone [15], and CoreLIME [16] are projects intended to address the shortcomings in the original LIME, such as global locks on federated spaces.

Further surveys of middleware, languages, and communication paradigms for MANET development can be found in [4] and [17]. We also compare MANET performance of traditional paradigms in [18].

7 Conclusion

MELON is a new communication paradigm designed for MANET application and middleware development. It provides a unique combination of new features for interacting with a distributed shared message store, including separation between read-only messages and removable messages, private messages, bulk message operations, and tracking of read messages. In this paper we used real applications to compare MELON performance to existing communication paradigms and demonstrated the new paradigm has acceptable overhead and performance in a MANET context, as well as being useful for general purpose applications.

There are several aspects of MELON which can be explored in future work including message replication, garbage collection, and secure communication. Message replication is very useful in MANETs to overcome network partitioning and increase availability. On the other side, garbage collection of old (and

replicated) messages is necessary to keep the MELON storage requirements low for small devices. While MELON does offer direct communication, encrypting private communications is necessary for full security against eavesdropping.

References

1. Patrick Th. Eugster and et al. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
2. Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.
3. David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
4. Justin Collins and Rajive Bagrodia. Programming in mobile ad hoc networks. In *WICON '08: Proc. of the 4th Annual International Conference on Wireless Internet*, pages 1–9. ICST, 2008.
5. Pieter Hintjens. *ZeroMQ: Messaging for Many Applications.* O'Reilly, 2013.
6. Scalable Networks. Exata: An exact digital network replica for testing, training and operations of network-centric systems. Technical brief, 2008.
7. Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.
8. 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.
9. Amy Murphy and et al. Lime: A coordination middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3):279–328, July 2006.
10. Klaus Herrmann and et al. Meshmdl event spaces - a coordination middleware for self-organizing applications in ad hoc networks. *Pervasive Mob. Comput.*, 3(4):467–487, 2007.
11. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. *2nd IEE Annual Conf. on Pervasive Computing and Communications*, pages 263–273, 14-17 March 2004.
12. Christine Julien and G-C Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *Software Engineering, IEEE Transactions on*, 32(5):281–298, 2006.
13. Bogdan Carbunar and et al. Lime revisited. In *Mobile Agents*, pages 54–69. Springer, 2001.
14. 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 Communications, IEEE*, 16(3):52–59, 2009.
15. Chien-Liang Fok and et al. A lightweight coordination middleware for mobile computing. In *Coordination Models and Languages*, pages 135–151. Springer, 2004.
16. Bogdan Carbunar and et al. Corelime:: A coordination model for mobile agents. *Electronic Notes in Theoretical Computer Science*, 54:17–34, 2001.
17. S Hadim and et al. Trends in middleware for mobile ad hoc networks. *Journal of Communication*, 1(4), 11-21 July 2006.
18. 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.