MELON: A Persistent Message-Based Communication Paradigm for MANETs

Justin Collins(✉) and Rajive Bagrodia

University of California, Los Angeles, CA, USA
{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 (e.g., a WiFi access point or cellular tower) to a central server is simpler, MANETs are useful when communication is between nearby devices or when there is no network infrastructure, such as in disaster recovery situations or locations without cellular service.

To alleviate 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, remote procedure calls, and tuple spaces, instead of MANET-focused paradigms [1].

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. MELONs offer remove-only, read-only, and private messages, as well as bulk transfers. MELON also simplifies communication

[1] Message Exchange Language Over the Network.

I. Stojmenovic et al. (Eds.): MOBIQUITOUS 2013, LNICST 131, pp. 716–720, 2014.
DOI: 10.1007/978-3-319-11569-6_59

by returning messages in per host write order. 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 latency.

2 MELON Design

Applications which are developed for MANETs must operate in an infrastructureless, unreliable, and dynamic distributed environment. We consider disconnection handling, addressing and discovery, and flexible communication important features for MANET development.

The design of MELON is built 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 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 do 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.

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 single host generates the majority of the messages, this removes the need to re-order messages in the application.

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, avoiding the multiple read problem [2].

Finally, MELON differentiates messages intended to persist and be read by many receivers from messages expected to be removed from the message store. For example, 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 supports both scenarios.

2.1 MELON Operations

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. Messages saved with a **write** operation cannot be explicitly removed, only copied. Messages saved with a **store** may optionally be directed to a specific receiver. Only the addressee may access a directed message (Table 1).

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.

Table 1. MELON operations

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

A **take** operation removes a message with matching content and returns it to the requesting process. A message may only be returned by a single **take** operation.

read operations also return a message matching a given template, but do not remove the original message. Any number of processes may read the same message, but repeated calls to **read** in the same process will never return the same message. Only messages stored with **write** may be returned by **read**.

MELON also includes the bulk operations **take_all** and **read_all** which mirror the basic operations, except all available matching messages will be returned. 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, all fetch operations will block the calling process until a matching message is available. MELON also provides non-blocking versions of these operations which return a null or empty value instead of blocking.

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.

Table 2. News server and reader

News Server	News Reader

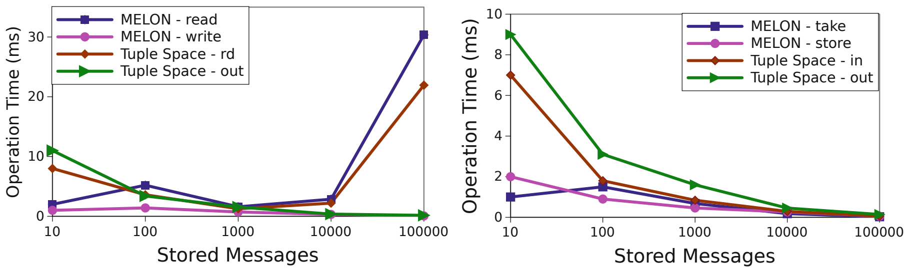
`function report(category, headline) {`
`   write([category, headline])`
`}`
	`function fetch(category) {`
	`   return read_all([category, String])`
	`}`

Table 2 shows a sample application. One or more news servers generate news messages containing a news category and headline. The server uses **write** to disallow removal of news items. Any number of processes can consume the news as readers, using **read_all** to return all news items in a given category. Repeated calls to `fetch` will only return news items not already seen.

3 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. We evaluated applications with the EXata network emulator [3] in order to run real applications and also have precisely repeatable environments with high fidelity network models. The scenario distributed 50 nodes in a 150 m square grid moving with a random waypoint mobility model. Signal propagation is limited to 50 m to match an indoor environment and force multihop routes, and the two-ray model is used for path loss. 802.11b WiFi is used with the AODV routing protocol.

Operation Speed. To establish a performance baseline, 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 [4] local tuple space implementation used by LIME [5].



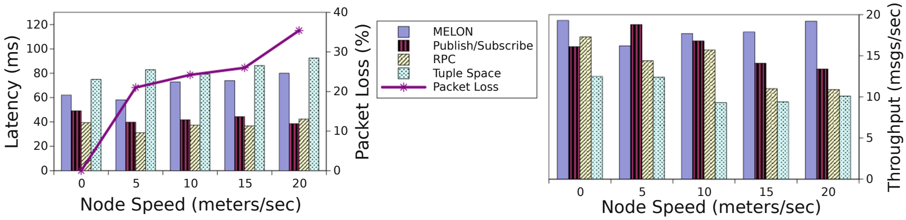
(a) Read Speed (b) Take Speed

Fig. 1. Operation speeds

Since LighTS and MELON search messages linearly, non-destructive reads are most affected by more stored messages. Removing messages is 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. Storing messages is naturally faster than removing for both implementations: storing a message takes less than 10 ms on average, and usually less than 4 ms (Fig. 1).

Message Latency. Figure 2(a) shows the average latency between request and receipt of a message. A single host writes out 1,000 messages with a 1 KB payload, and the other hosts read the messages concurrently. Tuple spaces and MELON use the **rd/read** operations to retrieve the messages singularly. For publish/subscribe, 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 push-based and has very low overhead, it is able to take advantage of the increased



(a) Message Latency (b) Message Throughput

Fig. 2. Communication performance

opportunities for transferring data. On the other hand, MELON and tuple spaces have high overhead and must repeatedly request messages from remote nodes.

Message Throughput. Throughput was measured on the receiver side in messages delivered per second. Figure 2(b) shows the average throughput with varying node speeds. Tuple spaces perform the worst, delivering 12.5–10.1 messages per second. MELON provides the best performance with 19.2–16.2 messages per second. Publish/subscribe performs well at moderate speeds (18.8 msgs/s at 5 m/s), but packet loss reduces the number of delivered messages and throughput drops 29 %–13.4 msgs/s at 20 m/s.

4 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 of 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 acceptable performance in a MANET context.

References

1. Collins, J., Bagrodia, R.: Programming in mobile ad hoc networks. In: WICON '08: 4th International Conference on Wireless Internet, pp. 1–9. ICST (2008)
2. Rowstron, A., Wood, A.: Solving the linda multiple rd problem. In: Hankin, C., Ciancarini, P. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 357–367. Springer, Heidelberg (1996)
3. Scalable Networks: Exata: an exact digital network replica for testing, training and operations of network-centric systems. Technical brief (2008)
4. Balzarotti, D., et al.: The lights tuple space framework and its customization for context-aware applications. Web Intell. Agent Syst. **5**(2), 215–231 (2007)
5. Murphy, A., et al.: Lime: a coordination middleware supporting mobility of hosts and agents. ACM Trans. Soft. Eng. Method. **15**(3), 279–328 (2006)