

A New Framework for Building Secure Collaborative Systems in True Ad Hoc Network

Hans-Peter Bischof, Alan Kaminsky, and Joseph Binder

Rochester Institute of Technology, 102
Lomb Memorial Dr, Rochester, NY 14623
{hpb,ark,jsb7834}@cs.rit.edu

Abstract. Many-to-Many Invocation (M2MI) is a new paradigm for building secure collaborative systems that run in true ad hoc networks of fixed and mobile computing devices. M2MI is useful for building a broad range of systems, including service discovery frameworks; groupware for mobile ad hoc collaboration; systems involving networked devices (printers, cameras, sensors); and collaborative middleware systems. M2MI provides an object oriented method call abstraction based on broadcasting. An M2MI invocation means “every object out there that implements this interface, call this method.” M2MI is layered on top of a new messaging protocol, the Many-to-Many Protocol (M2MP), which broadcasts messages to all nearby devices using the wireless network's inherent broadcast nature instead of routing messages from device to device. In an M2MI-based system, central servers are not required; network administration is not required; complicated, resource-consuming ad hoc routing protocols are not required; and system development and deployment are simplified.

Keywords: Collaborative systems, peer-to-peer systems, distributed objects, decentralized key management, ad hoc networking, server-less networking.

Introduction

This paper describes a new paradigm, Many-to-Many Invocation (M2MI), for building secure collaborative systems that run in true ad hoc networks of fixed and mobile computing devices. M2MI is useful for building a broad range of systems, including service discovery frameworks; groupware for mobile ad hoc collaboration.

We also address encryption and decryption of M2MI method invocations and a describe a decentralized key management in ad hoc networks.

M2MI provides an object oriented method call abstraction based on broadcasting. An M2MI-based application broadcasts method invocations, which are received and performed by many objects in many target devices simultaneously. An M2MI invocation means “Everyone out there that implements this interface, call this method.” The calling application does not need to know the identities of the target devices ahead of time, does not need to explicitly discover the target devices, and does not need to set up individual connections to the target devices. The calling device simply broadcasts method invocations, and all objects in the proximal network that implement those methods will execute them.

As a result, M2MI offers these key advantages over existing systems:

- M2MI-based systems do not require central servers; instead, applications run collectively on the proximal devices themselves.
- M2MI-based systems do not require network administration to assign addresses to devices, set up routing, and so on, since method invocations are broadcast to all nearby devices. Consequently,
- M2MI is well-suited for an ad hoc networking environment where central servers may not be available and devices may come and go unpredictably.
- M2MI-based systems allow to decrypt an encrypt method invocations using session keys [9].
- M2MI-based systems do not need complicated ad hoc routing protocols that consume memory, processing, and network bandwidth resources [10]. Consequently,
- M2MI is well-suited for small mobile devices with limited resources and battery life.
- M2MI simplifies system development in several ways. By using M2MI's high-level method call abstraction, developers avoid having to work with low-level network messages. Since M2MI does not need to discover target devices explicitly or set up individual connections, developers need not write the code to do all that.
- M2MI simplifies system deployment by eliminating the need for always-on application servers, lookup services, code-base servers, and so on; by eliminating the software that would otherwise have to be installed on all these servers; and by eliminating the need for network configuration.

M2MI's key technical innovations are these:

- M2MI layers an object oriented abstraction on top of broadcast messaging, letting the application developer work with high-level method calls instead of low-level network messages.
- M2MI uses dynamic proxy synthesis to create remote method invocation proxies (stubs and skeletons) automatically at run time - as opposed to existing remote method invocation systems, which compile the proxies, offline and which must deploy the proxies ahead of time.

This paper is organized as follows: the next chapter describes the target environment for M2MI based systems; the following chapter discusses the M2MI paradigm followed by a chapter showing how M2MI can be used to develop applications and service discovery frameworks. The last two chapters discuss a dynamic fault tolerant key management system.

Target Environmnet

M2MI's target domain is ad hoc collaborative systems: systems where multiple users with computing devices, as well as multiple standalone devices like printers, cameras, and sensors, all participate simultaneously (collaborative); and systems where the various devices come and go and so are not configured to know about each other ahead of time (ad hoc). Examples of ad hoc collaborative systems include:

- Applications that discover and use nearby networked services: a document printing application that finds printers wherever the user happens to be, or a surveillance application that displays images from nearby video cameras.
- Collaborative middleware systems like shared tuple spaces [1].
- Groupware applications: a chat session, a shared whiteboard, a group appointment scheduler, a file sharing application, or a multiplayer game.

In many such collaborative systems, every device needs to talk to every other device. Every person's chat messages are displayed on every person's device; every person's calendar on every person's device is queried and updated with the next meeting time. In contrast to applications like email or web browsing (one-to-one communication) or web-casting (one-to-many communication), the collaborative systems envisioned here exhibit many-to-many communication patterns. M2MI is designed especially to support applications with many-to-many communication patterns, although it also supports other communication patterns.

Devices come and go as the system is running, the devices do not know each other's identities beforehand; instead, the devices form ad hoc networks among themselves.

M2MI is intended for running collaborative systems without central servers. In a wireless ad hoc network of devices, relying on servers in a wired network is unattractive because the devices are not necessarily always in range of a wireless access point. Furthermore, relying on any one wireless device to act as a server is unattractive because devices may come and go without prior notification. Instead, all the devices - whichever ones happen to be present in the changing set of proximal devices - act in concert to run the system.

The M2MI Paradigm

Remote method invocation (RMI) [7] can be viewed as an object oriented abstraction of point-to-point communication: what looks like a method call is in fact a message sent and a response sent back. In the same way, M2MI can be viewed as an object oriented abstraction of broadcast communication. This section describes the M2MI paradigm at a conceptual level.

Handles

M2MI lets an application invoke a method declared in an interface. To do so, the application needs some kind of "reference" upon which to perform the invocation. In M2MI, a reference is called a handle, and there are three varieties, omnihandles, unihandles, and multihandles.

Omnihandles

An omnihandle for an interface stands for "every object out there that implements this interface." An application can ask the M2MI layer to create an omnihandle for a cer-

tain interface `X`, called the omnihandle's target interface. (A handle can implement more than one target interface if desired. An omnihandle for interface `Foo`; the omnihandle is named `allFoods` is created by code like this:

```
Foo allFoods = M2MI.getOmnihandle(Foo.class);
```

Once an omnihandle is created, calling method `doSomething` on the omnihandle for interface `AnInterface` means, “Every object out there that implements interface `AnInterface`, perform method `doSomething`.” The method is actually performed by whichever objects implementing interface `AnInterface` exist at the time the method is invoked on the omnihandle. Thus, different objects could respond to an omnihandle invocation at different times. Three objects implementing interface `Foo`, objects `A`, `B`, and `D`, happen to be in existence at that time; so all three objects perform method `y`. Note that even though object `D` did not exist when the omnihandle `allFoods` was created, the method is nonetheless invoked on object `D`.

The target objects invoked by an M2MI method call need not reside in the same process as the calling object. The target objects can reside in other processes or other devices. As long as the target objects are in range to receive a broadcast from the calling object over the network, the M2MI layer will find the target objects and perform a remote method invocation on each one.

Exporting Objects

To receive invocations on a certain interface `X`, an application creates an object that implements interface `X` and exports the object to the M2MI layer. Thereafter, the M2MI layer will invoke that object's method `Y` whenever anyone calls method `Y` on an omnihandle for interface `X`. An object is exported with code like this:

```
M2MI.export(b, Foo.class);
```

`Foo.class` is the class of the target interface through which M2MI invocations will come to the object. We say the object is “exported as type `Foo`.” M2MI also lets an object be exported as more than one target interface. Once exported, an object stays exported until explicitly unexported:

```
M2MI.unexport(b);
```

In other words, M2MI does not do distributed garbage collection (DGC). In many distributed collaborative applications, DGC is unwanted; an object that is exported by one device as part of a distributed application should remain exported even if there are no other devices invoking the object yet. In cases where DGC is needed, it can be provided by a leasing mechanism explicit in the interface.

Unihandles

A unihandle for an interface stands for “one particular object out there that implements this interface.” An application can export an object and have the M2MI layer

return a unihandle for that object. Unlike an omnihandle, a unihandle is bound to one particular object at the time the unihandle is created. A unihandle is created by code like this:

```
Foo b_Foo = M2MI.getUnihandle(b, Foo.class);
```

Once a unihandle is created, calling method Y on the unihandle means, “The particular object out there associated with this unihandle, perform method Y.” When the statement `b_Foo.y()`; is executed, only object B performs the method. As with an omnihandle, the target object for a unihandle invocation need not reside in the same process or device as the calling object.

A unihandle can be detached from its object, after which the object can no longer be invoked via the unihandle:

```
b_Foo.detach();
```

Multihandles

A multihandle for an interface stands for “one particular set of objects out there that implement this interface.” Unlike a unihandle which only refers to one object, a multihandle can refer to zero or more objects. But unlike an omnihandle which automatically refers to all objects that implement a certain target interface, a multihandle only refers to those objects that have been explicitly attached to the multihandle.

The multihandle is named `someFoos`, and it is attached to two objects, A and D. The multihandle is created and attached to the objects by code like this:

```
Foo someFoos = M2MI.getMultihandle(Foo.class);
someFoos.attach(a); someFoos.attach(d);
```

Once a multihandle is created, calling method Y on the multihandle means, “The particular object or objects out there associated with this multihandle, perform method Y.” When the statement `someFoos.y()`; is executed, objects A and D perform the method, but not objects B or C. As with an omnihandle or unihandle, the target objects for a multihandle invocation need not reside in the same process or device as the calling object or each other. A multihandle can be created in one process and sent to another process, and the destination process can then attach its own objects to the multihandle.

An object can also be detached from a multihandle:

```
someFoos.detach(a);
```

M2MI-Based Systems

This section gives one examples showing how M2MI can be used to design a chat application and a print service discovery system. These examples show the elegance of ad hoc collaborative systems based on M2MI. Further examples can be found at [4].

Service Discovery – Printing

As an example of an M2MI-based system involving stand-alone devices providing services, consider printing. To print a document from a mobile device, the user must discover the nearby printers and print the document on one selected printer. Printer discovery is a two-step process: the user broadcasts a printer discovery request via an omnihandle invocation; then each printer sends its own unihandle back to the user via a unihandle invocation on the user. To print the document, the user does an invocation on the selected printer's unihandle.

Specifically, each printer has a print service object that implements this interface:

```
public interface PrintService {
    public void print(Document doc);
}
```

The printer exports its print service object to the M2MI layer and obtains a unihandle attached to the object. The printer is now prepared to process document printing requests. To discover printers, there are two print discovery interfaces:

```
public interface PrintDiscovery {
    public void request(PrintClient client);
}
public interface PrintClient {
    public void report(PrintService printer,
                      String name);
}
```

The client printing application exports a print client object implementing interface `PrintClient` to the M2MI layer and obtains a unihandle attached to the object. The application also obtains from the M2MI layer an omnihandle for interface `PrintDiscovery`. The application is now prepared to make print discovery requests and process print discovery reports.

Each printer exports a print discovery object implementing interface `PrintDiscovery` to the M2MI layer. The printer is now prepared to process print discovery requests and generate print discovery reports

The application first calls

```
printDiscovery.request(theClient);
```

on an omnihandle for interface `PrintDiscovery`, passing in the unihandle to its own print client object. Since it is invoked on an omnihandle, this call goes to all the printers. The application now waits for print discovery reports.

Each printer's request method calls

```
theClient.report(thePrinter,
                 "Printer Name");
```

The method is invoked on the print client unihandle passed in as an argument. The method call arguments are the unihandle to the printer's print service object and the name of the printer. Since it is invoked on a unihandle, this call goes just to the re-

questing client printing application, not to any other print clients that may be present. After executing all the report invocations, the printing application knows the name of each available printer and has a unihandle for submitting jobs to each printer.

Finally, after asking the user to select one of the printers, the application calls:

```
c_Printer.print(theDocument);
```

where `c_Printer` is the selected printer's unihandle as previously passed to the report method. Since it is invoked on a unihandle, this call goes just to the selected printer, not the other printers. The printer proceeds to print the document passed to the print method.

Clearly, this invocation pattern of broadcast discovery request - discovery responses - service usage can apply to any service, not just printing. It is even possible to define a generic service discovery interface that can be used to find objects that implement any interface, the desired interface being specified as a parameter of the discovery method invocation.

M2MI Architecture

Our initial work with M2MI has focused on networked collaborative systems. In this environment of ad hoc networks of proximal mobile wireless devices, M2MI is layered on top of a new network protocol, M2MP. We have implemented initial versions of M2MP and M2MI in Java. A detailed description of the design and architecture can be found at [4].

M2MI Security

Providing security within M2MI-based systems is an area of current development. We have identified these general security requirements:

- Confidentiality - Intruders who are not part of a collaborative system must not be able to understand the contents of the M2MI invocations.
- Participant authentication - Intruders who are not authorized to participate in a collaborative system must not be able to perform M2MI invocations in that system.
- Service authentication - Intruders must not be able to masquerade as legitimate participants in a collaborative system and accept M2MI invocations. For example, a client must be assured that a service claiming to be a certain printer really is the printer that is going to print the client's job and not some intruder.

While existing techniques for achieving confidentiality and authentication work well in an environment of fixed hosts, wired networks, these techniques will not work well in an environment of mobile devices, wireless networks, and no central servers.

A decentralized key management is necessary in order to achieve the security requirements.

Decentralized Keymanagement in Ad Hoc Networkd

State of the Art

Key management has been the thrust of several research initiatives in the ad hoc networking domain (e.g., [1, 6] et al). Each of these approaches seeks to establish a public key infrastructure within the constraints of ad hoc networks; each approach is discussed below.

“Securing Ad Hoc Networks” [10] was one of the first notable publications to propose a public key management service for ad hoc networks. The service itself encapsulates a public/private key pair K/k . The private key, k , is used to sign other nodes' public keys; the public key, K , is used to verify the signature. The service employs a $(n, t+1)$ threshold scheme to distribute the private key and the digital signing process among n nodes. Each of the n nodes is denoted as a server node, as it has a special role in the signing service. Combiner nodes - which may be a subset of the server nodes or altogether different nodes - are also required to combine each server's partial signature. For example, to sign a certificate, each of the n server nodes must generate a partial signature using its share of the private $(k_1, k_2, \dots, k_n)$ to compute a partial signature of the certificate. Once generated, each server node sends its partial signature to the combiner; the combiner then computes the entire signature. To its credit [10] was quite progressive at its inception, as its design is largely proactive and capable of handling a dynamic network state. Nonetheless, the service has remnants of its wired predecessor, namely, a trusted authority, and specialized server and combiner nodes. Although the threshold scheme employed allows $t < n$ servers to be compromised without sacrificing the service, its largely centralized approach encapsulates relatively few points of failure and attack.

“Providing Robust and Ubiquitous Security Support for Ad Hoc Networks [6] presents a natural extension to [1], wherein the signing service is distributed to any node n in the network. For example, if a network member requires a certificate, it need only be in the proximity of *any* $t+1$ nodes. The service is otherwise similar to [6]. Despite the improved distribution, [6] still requires a trusted party at initialization. Further, because any node in the network may participate in the sharing, a malicious node may masquerade as $t+1$ bogus nodes and reconstruct the private key.

More recently, Hubaux et al have proposed a self organizing public key infrastructure in [1]. Unlike the previous two publications, [1] does not require a trusted authority or any specialized nodes; instead, each node issues its own certificates to other nodes. Each node maintains a limited repository of other nodes' certificates. When a node wishes to validate a certificate of another node, the nodes combine their certificate repositories; the validating node then examines the merged certificate repository for falsified certificates. If none are found, the certificate is accepted; otherwise it is rejected. The primary drawback of [1] is its initialization time. In long-lived ad hoc networks, such overhead may be admissible; it is likely to be prohibitive in more transient settings.

Although each of the above paradigms is effective in its own right, they are all based on a common assumption, namely, point-to-point communication. Public key infrastructures enable nodes with authentic public encryption keys that they may use to establish secure communication with one another. However, many ad hoc networks are collaborative, many-to-many environments. In these settings, public key cryptography is computationally intensive, as each group message must be encrypted $n-1$

times. Group key management paradigms which provide a shared symmetric key that is shared among all group members, have been used throughout the wired networking domain to secure broadcast and many-to-many communication environments; however, very few attempts have been made to adapt group key management infrastructures to an ad hoc setting.

Dominant group key management paradigms include the well-known CLIQUES project [8], Kim et al [5], and several others. Each of these protocols is based on the generalized Diffie-Hellman problem, which requires every network member to contribute to the generation of the shared group key. Because they were developed for wired environments, many of these approaches require point-to-point and broadcast mediums, synchronous messaging, and static network topologies. Unfortunately, the wireless, amorphous, transient, many-to-many nature of ad hoc networks precludes many of the assumptions on which the above protocols were developed. We, therefore, introduce a new approach to key management that can effectively function within the constraints of an ad hoc network environment.

Looking Forward

The ad hoc network environment we envision is transient, dynamic in structure and membership, proximal, and broadcast-based. We also assume that network nodes wish to collaborate, that is, our primary goal is to ensure secure many-to-many communication. As a result, our paradigm is fully decentralized (i.e., it lacks server or otherwise specialized nodes), lightweight, and best-suited for small, spontaneous networks. The first protocol we present is not authenticated; the second is an extension of the first that includes authentication mechanisms.

The nucleus of our first protocol is a tuple-like entity, inspired by Gelerntner's tuples in [2], that is effectively a hash table shared among all members of the group. Each member of the group has an entry in the hash table, which includes that member's contribution to the group key.

The following atomic operations may be performed on the tuple:

- *take()* - removes the tuple from the space, such that no other group member may modify its contents.
- *read()* - reads the current contents of the tuple
- *write()* - writes the tuple into the space, overwriting the previous tuple

Although the tuple spaces are often implemented as a centrally-based service, the tuples used in this context are fully distributed: each member hosts its own entry in the tuple. Nodes may host more than one entry if replication is desired in the interest of availability.

Group Genesis

Group genesis requires two or more parties to be present.

1. Group members agree on a cyclic group, G , of order q , and a generator, α in G ; each member then chooses a secret share, $N_i \in G$.
2. The first member, M_1 , instantiates a Tuple Space and places a new tuple in the space. The tuple initially contains M_1 's contribution and the current cardinal value. M_1 then sends a broadcast message to the group stating that tuple has been created.

3. Upon receipt of the broadcast message, each member attempts to remove the tuple from the space in order to add its contribution. Because *take()* request will withdraw the tuple from the space; the other *take()* will block until the tuple is returned to the space. The member who receives the tuple then adds an entry in the tuple for itself and updates all existing intermediate values and the cardinal value. This step is repeated until $M_2 \dots M_{n-1}$ have written their contributions into the tuple.
4. The last member of the group has special role in the key generation process. The last member is not pre-determined; it is simply the last member to send a *take()* request. M_n first performs a *take()* operation on the tuple. It then exponentiates each intermediate value in the tuple with its secret exponent, S_n , and adds in an intermediate value for itself. Unlike its predecessors, M_n does not update the cardinal value, as the final cardinal value is the group key. Instead, it writes the tuple back into the space with the previous cardinal value and the updated intermediate values. M_n then sends a broadcast message to the group, which informs them of the termination of the key generation phase.

Upon receipt of the broadcast message, each member *read()*s its intermediate value and uses it to compute the group key.

Member Addition – *join()*

A *join()* operation denotes the addition of a single group member. Semantics for *join()* entail a modification of the group key, such that the new member's share is included in the group key. The steps required for *join()* follow.

1. M_{n+1} *take()*s the tuple out of the space, adds its intermediate value, updates each existing intermediate values, and *write()*s the tuple back into the space.
2. M_{GC} performs a *take()* on the tuple, updates the cardinal value, *write()*s the tuple back into the space, and notifies all group members that the key generation is complete.

Following a *join()* operation, the new member becomes new group controller (i.e., $M_{n+1} = M_{GC}$).

By default, join does not ensure forward or backward secrecy. In many scenarios, this may be admissible; however, a simple extension to the join operation can ensure forward and backward secrecy. The revised protocol requires the existing group controller, M_n , factor its secret, S_n out of the existing cardinal and intermediate values, choose a new secret, S_n , and exponentiate each intermediate value with it.

Member Removal - *leave()*

Leave entails the removal of a group member's contribution to the group key, thereby prohibiting it from decrypting subsequent group messages. The following protocol assumes that the departure is voluntary. If the departure is not voluntary, the first step is clearly omitted, however, the excluded member is still unable to derive the group key.

1. The departing member, M_p , factors its contribution out of each entry in the tuple.
2. The group controller, M_{GC} , chooses a new secret S_{GC} and exponentiates each entry in the tuple with it.

Conclusion

We present a dynamic, fault--tolerant symmetric key management system. Unlike other key management paradigms, our approach does not require a specific order in which contributions are collected, nor does it rely on a trusted or centralized entity to combine the partial keys.

References

1. S. Capkun, L. Buttyan, and J. Hubaux. Self-organized public-key management for mobile ad hoc networks, 2002.
2. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, January 1985.
3. Internet Engineering Task Force. IP Routing for Wireless/Mobile Hosts (mobileip) Working Group. <http://www.ietf.org/html.charters/mobileip-charter.html>.
4. A. Kaminsky, Hans-Peter Bischof. Many-to-Many Invocation: A new object oriented paradigm for ad hoc collaborative systems. 17th Annual ACM Conference on *Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, Onward! track, November 2002, to appear. Preprint: <http://www.cs.rit.edu/~anhinga/publications/publications.shtml>
5. Yongdae Kim, Adrian Perrig, and Gene Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In Proceedings of the 7th ACM conference on Computer and communications security, pages 235-244, 2000.
6. H. Luo and S. Lu. Ubiquitous and robust authentication services for ad hoc wireless networks, 2000.
7. Michael Steiner, Gene Tsudik, and Michael Waidner. CLIQUES: A new approach to group key agreement. In Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS98), pages 380-387, Amsterdam, 1998. IEEE Computer Society Press.
8. Jefferson S. Tuttle. Security in an Ad Hoc Network using Many-to-Many Invocation, <http://www.cs.rit.edu/~jst1734>
9. A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265-290, Fall 1996.
10. S.-M. Yoo and Z.-H. Zhou. All-to-all communication in wireless ad hoc networks. In Proceedings of the 39th Annual ACM Southeast Conference, pages 180-181, March 2001. <http://webster.cs.uga.edu/~jam/acm-se/review/abstract/syoo.ps>.
11. Lidong Zhou and Zygmunt J. Haas. Securing ad hoc networks. *IEEE Network*, 13(6):2430, 1999.