2002

Many-to-many invocation: A New paradigm for ad hoc collaborative systems

Alan Kaminsky

Hans-Peter Bischof

Many-to-Many Invocation: A New Paradigm for Ad Hoc Collaborative Systems

Alan Kaminsky
Information Technology Laboratory
Rochester Institute of Technology
ark@it.rit.edu

Hans-Peter Bishof
Department of Computer Science
Rochester Institute of Technology
hpb@cs.rit.edu

IT Lab Technical Report TR-2002-01
February 6, 2002

1 Introduction

This report describes a new and innovative paradigm, Many-to-Many Invocation (M2MI), for building collaborative systems that run in wireless proximal ad hoc networks of fixed and mobile computing devices. M2MI is useful for building a broad range of systems, including multiuser applications (conversations, groupware, multiplayer games); systems involving networked devices (printers, cameras); wireless sensor networks; and collaborative middleware systems.

M2MI provides an object-oriented method call abstraction based on broadcasting. An M2MI-based application broadcasts a series of method invocations, which are received and performed by many target devices simultaneously. An M2MI invocation means "Everyone that's out there, 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 devices 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 *do not need complicated ad hoc routing protocols* that consume memory, processing, and network bandwidth resources.
- 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, codebase 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 employs a *new message broadcasting protocol,* the Many-to-Many Protocol (M2MP), which uses a fundamentally different approach compared to existing ad hoc networking protocols. Instead of routing messages from point to point to the particular destination devices, M2MP broadcasts messages to all nearby devices, taking advantage of the wired or wireless network's inherent broadcast nature. Based on

the message contents, the devices then decide whether and how to process the message.

- 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 report is organized as follows. Section 2 describes the application domain and networking environment at which M2MI is targeted. Section 3 describes the M2MI paradigm at a conceptual level. Section 4 describes how M2MI is implemented. Section 5 compares and contrasts M2MI with related work. Section 6 discusses the current status of M2MI and plans for further work.

2 Target Environment

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:

- Multiuser applications: a chat session, a shared whiteboard, a group appointment scheduler, or a multiplayer game
- 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, 2]

M2MI is designed to take advantage of a wireless proximal ad hoc networking environment. The devices in the system connect to each other using *wireless* networking technology such as IEEE 802.11 or Bluetooth. The devices are located in *proximity* to each other, around the same table or in the same room; every device can hear every other device. Devices come and go as the system is running, and the devices do not know each others' identities beforehand; instead, the devices form *ad hoc* networks among themselves.

M2MI is also intended to run in *small, battery powered* devices with limited memory sizes and CPU capacity. Unlike desktop computers, such devices cannot maintain constant network connections because that would rapidly drain their batteries. To make each battery charge last as long as possible, reducing network utilization is essential.

Lastly, M2MI is intended for running collaborative systems *without central servers.* Relying on servers is unattractive because servers crash, networks go down, and devices move out of range of wireless access points. Also, any one device can't act as a server 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, even if no central server is available.

3 The M2MI Paradigm

The following sections describe M2MP, M2MI, and M2MI-based applications in more detail.

3.1 Many-to-Many Protocol

Designed particularly for the wireless proximal ad hoc networking environment, M2MP has these characteristics:

- *There are no device addresses.* Consequently, devices can enter and leave the network in an ad hoc fashion without having to maintain any routing tables.
- *Messages are broadcast to all devices.* Since wireless radio transmissions are inherently broadcast within a certain proximal area, at the radio level it's just as easy to deliver a message to all devices as to one device.
- *Message delivery is mostly reliable.* Most of the time, a message broadcast by one device is received by all the other devices. However, on rare occasions a message broadcast by one device is not received by some or all of the other devices.

Figure 1 shows the protocol architecture. Applications in each device are layered on top of M2MP, which in turn is layered on top of a broadcast communication medium. Each application on each device registers one or more *message filters* with M2MP (the small gray boxes in Figure 1). An incoming message whose contents match what the

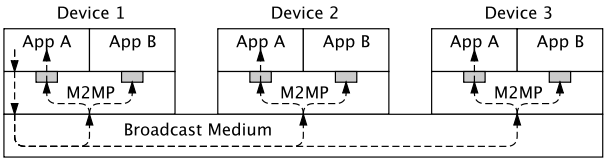

Figure 1: M2MP protocol architecture

application is looking for is allowed to pass through the application's message filter; other incoming messages are not passed through.

When an application on one device sends a message, M2MP breaks the message into packets and broadcasts each packet. Every device receives each packet (barring failures). If a packet's contents show that it is part of a message that matches an application's message filter, the M2MP layer reassembles the original message from the packets and passes the message to the application, otherwise the M2MP layer ignores the packet. If a failure occurs, such as a lost packet, the M2MP layer abandons the message and signals an exception to the application.

Retransmitting lost packets is unnecessary, and abandoning the message is acceptable, because we assume the network is mostly reliable. Recovery from an occasional message loss can be done at the application level. Indeed, the messaging layer should not be expected to provide end-to-end delivery or ordering guarantees [3]. This considerably simplifies M2MP.

3.2 Many-to-Many Invocation

Remote method invocation (RMI) [4] 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.

M2MI lets an application invoke a method declared in an interface. The interface's methods must all be declared not to return a value and not to throw any exceptions. The methods can, of course, have arguments.

To call a method in an interface via M2MI, the application needs some kind of "reference" upon which to perform the invocation. In M2MI, a reference is called a *handle*, and there are two varieties, omnihandles and unihandles.

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 certain interface X. Thereafter, calling method Y on the omnihandle for interface X means, "Every object out there that implements interface X, perform method Y."

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.

A *unihandle* for an interface stands for "one particular object out there that implements this interface." When an application exports an object, the M2MI layer returns a unihandle for that object. Thereafter, calling method Y on the unihandle means, "The particular object out there associated with this unihandle, perform method Y."

When an object of a non-primitive type is passed directly as an M2MI method call argument, the object is normally *passed by copy;* manipulations of the argument by the method call recipient do not affect the original object in the caller. However, when a unihandle for an exported object is passed as an M2MI method call argument, the object is effectively *passed by reference;* invocations performed by the method call recipient on the argument (unihandle) come back to the original object via M2MI and thus do affect the original object in the caller. (Primitive types are always passed by copy in M2MI.)

Since M2MI methods do not return anything, the caller cannot get any information back from the called objects *in the same method call.* If the caller needs to get information back, the caller can broadcast a reference to its own object by passing the object's unihandle as an argument to a method invoked on an omnihandle. The called objects can then send information back by performing subsequent method invocations on the unihandle. This leads to a pattern of *asynchronous* method calls and callbacks in an M2MI-based application as shown in the examples below.

3.3 M2MI-Based Systems

3.3.1 Chat: Broadcast Invocations

As an example of an M2MI-based collaborative system, consider a simple chat session. Each user's chat application has an object implementing this interface:

```
public interface Chat {
   public void put (String line);
}
```

The application exports the chat object to the M2MI layer. The application also obtains from the

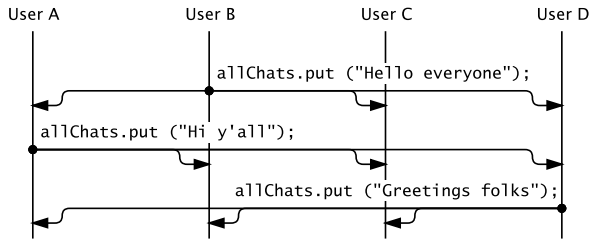

Figure 2: M2MI invocations for a chat application

M2MI layer an omnihandle for interface `Chat` and stores the omnihandle as `allChats`.

Figure 2 shows a sequence of M2MI invocations that might occur when four instances of this chat application run in four nearby devices. To send a line to everyone in the chat session, the application does a method call on the omnihandle:

```
allChats.put ("Line of text");
```

The chat object's implementation of the `put` method adds the line of text to the chat session log displayed on the user's device. Thus, in response to the above omnihandle invocation, all the exported chat objects display the line of text on all the users' devices.

Note that the M2MI-based chat application does not need to find and connect to a central chat server. Neither does the application need to know which other devices are part of the chat session or connect to them. The user's device simply shows up and starts broadcasting `put` invocations. This shows how M2MI simplifies the development of collaborative systems.

3.3.2 IM: Mixed Invocations

As another example, consider a simple instant messaging (IM) system. The IM application needs to discover which users are out there and send messages to individual users (unlike the chat application which sends messages to all users). To discover users, each application broadcasts a unihandle to itself via an omnihandle invocation. To send a message to a particular user, the application invokes a method on that user's unihandle, which was received in a prior broadcast.

Specifically, each user's IM application has an object implementing this interface:

```
public interface InstantMessage {
   public void present (String name,
      InstantMessage participant);
   public void put (String line,
      InstantMessage sender);
}
```

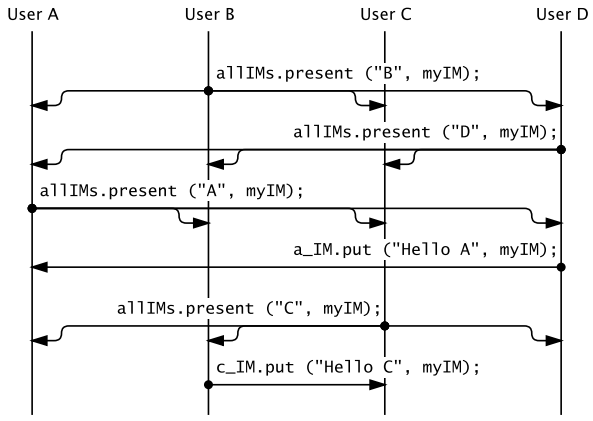

Figure 3: M2MI invocations for an IM application

The application exports the IM object to the M2MI layer. The M2MI layer returns a unihandle to the IM object, which the application stores as `myIM`. The application also obtains from the M2MI layer an omnihandle for interface `InstantMessage` and stores the omnihandle as `allIMs`.

Figure 3 shows a sequence of M2MI invocations that might occur when four instances of this IM application run in four nearby devices. Each application announces its presence by calling the `present` method on the omnihandle, passing in the user's name and the unihandle to its own IM object:

```
allIMs.present ("User Name", myIM);
```

Executing the `present` method, each IM object stores the user name and the unihandle in an internal list for later use.

To send an instant message to a particular user X, the application looks up the corresponding unihandle in the user list and calls the `put` method on the unihandle, passing in the message text and the unihandle to its own IM object (so the recipient knows who sent the message):

```
x_IM.put ("Line of text", myIM);
```

where `x_IM` is the unihandle for user X's IM object. The `put` method displays the message and the sender on the device's display. Since the invocation is performed on a unihandle, not an omnihandle, only the destined user's IM object executes the `put` method and displays the message; the message does not appear on the other devices' displays.

To show that the user is still present, each instance of the IM application broadcasts a `present` invocation periodically. If the time since the last `present` invocation for a certain user exceeds a threshold, the other IM applications conclude the user has gone away and remove the user from their user lists.

3.3.3 Printing: Service Discovery

As an example of an M2MI-based system involving standalone devices, 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 discover
      (PrintClient client);
   public void print (Document doc);
}
```

The printer exports its print service object to the M2MI layer and saves the object's unihandle as `myPrinter`. The printer is now prepared to process discovery requests and document printing requests.

The document printing application has a print client object that implements this interface:

```
public interface PrintClient {
   public void present (String name,
      PrintService printer);
}
```

The application exports its print client object to the M2MI layer and saves the object's unihandle as `theClient`. The application also obtains from the M2MI layer an omnihandle for interface `PrintService` and saves it as `allPrinters`. The application is now prepared to make print discovery requests and process print discovery responses.

Figure 4 shows the sequence of M2MI invocations that occur when the document printing application goes to print a document with three printers nearby. The application first calls

```
allPrinters.discover (theClient);
```

passing 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 responses.

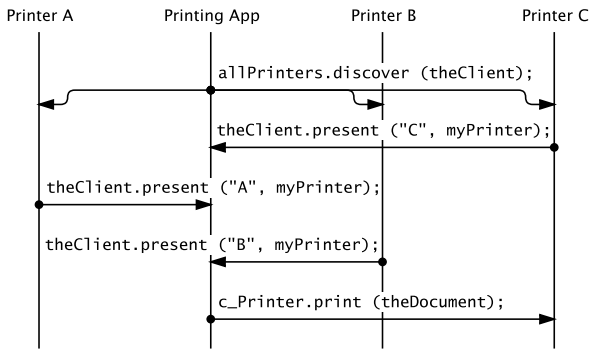

Figure 4: M2MI invocations for a print service

Each printer's `discover` method calls

```
theClient.present ("Printer Name",
   myPrinter);
```

The method is invoked on the print client unihandle passed in as an argument. The method call arguments are the name of the printer and the unihandle to the printer's print service object. After executing all the `present` 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

```
x_Printer.print (theDocument);
```

where `x_Printer` is the selected printer's unihandle as previously passed to the `present` 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.

4 Design

We have implemented initial versions of M2MP and M2MI in Java. The sections below briefly describe how M2MP and M2MI are designed.

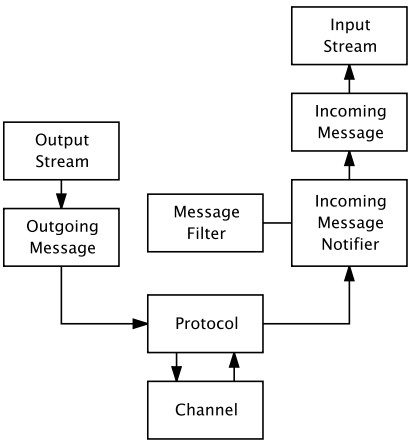

Figure 5: M2MP design

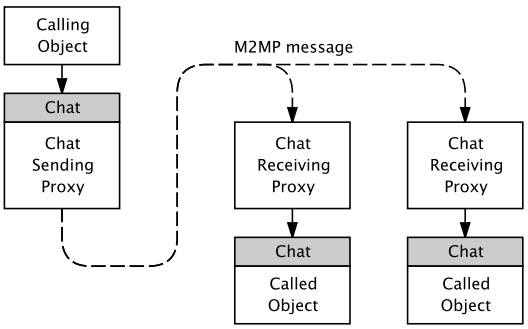

Figure 6: M2MI design

4.1 M2MP Design

Figure 5 shows the principal objects in the M2MP implementation and the patterns of data flow among them. The protocol object forms the core of M2MP. The channel object interfaces the protocol core to the broadcast communication medium. By plugging a different channel object into the protocol core, M2MP can be used with different underlying media. The remaining objects interface the rest of the application to the protocol core.

To send an M2MP message, the application creates an outgoing message object, obtains an output stream from the outgoing message, and writes the message's contents to the output stream.

To receive M2MP messages, the application creates an incoming message notifier object. The application registers one or more message filter objects with the incoming message notifier. The application reads incoming message objects from the incoming message notifier, which returns only those messages that match the message filters. For each incoming message, the application obtains an input stream and reads the message's contents from the input stream.

For outgoing messages, the protocol core breaks the message into packets and broadcasts each packet via the channel. Concurrently, the protocol core receives packets from the channel, matches the packets' contents against the incoming message notifiers' message filters, and (if there is a match) reassembles the incoming message from the packets.

4.2 M2MI Design

Figure 6 shows how M2MI implements a method invocation from a calling object to one or more called objects that implement a certain interface, `Chat` in this example. On the calling side, the calling object invokes a method on a *sending proxy object* which implements the `Chat` interface. The sending proxy converts each method argument to a sequence of bytes using Java's object serialization [5] and broadcasts an M2MP message containing the interface being invoked, the identifier for the called object, the method being invoked, and the method's serialized arguments.

On the called side, when the called object is exported, the M2MI layer creates a *receiving proxy object*, assigns the receiving proxy object a unique identifier, registers two message filters with the M2MP layer, and links the receiving proxy object to the called object. One message filter matches a message containing the interface being invoked plus a wildcard identifier, the other message filter matches a message containing the interface being invoked plus the receiving proxy's identifier. When such a message arrives, the receiving proxy object deserializes the method arguments and invokes the method on the called object.

An omnihandle for an interface is just an instance of the sending proxy class that puts a wildcard object identifier into the M2MP messages the proxy sends. Such a message will match the first message filter registered for every object that implements the interface, and thus will trigger a method invocation on all the called objects.

A unihandle for a particular object is just an instance of the sending proxy class that puts the object's identifier into the M2MP messages the proxy sends. Such a message will match the second message filter registered for the one object with that identifier, and thus will trigger a method invocation on just the one object.

The M2MI layer does not require the sending and receiving proxy classes to be compiled beforehand and downloaded from a codebase server. Instead, the

M2MI layer *synthesizes* the proxy classes on the fly, as needed, by building the requisite binary class files in memory and feeding them directly into a special class loader. The M2MI layer makes use of a Java library for synthesizing Java class files [6].

When a unihandle is passed as a method argument in an M2MI invocation, the unihandle itself is not serialized. Instead, a *handle transport object*, containing instructions for synthesizing the proxy class and initializing the proxy instance at the far end, is serialized and sent in the M2MI invocation message. When deserialized at the far end, the handle transport object causes the M2MI layer to synthesize the sending proxy class if necessary and create a sending proxy class instance initialized with the proper object identifier. The far end now has a copy of the original unihandle (without having had to download the proxy class file). This unihandle then takes the handle transport object's place as the argument for the method call.

5 Related Work

M2MI touches on four areas of related work: ad hoc networking, remote method invocation, distributed systems architecture, and collaborative middleware.

5.1 Ad Hoc Networking

A considerable amount of work has been done on ad hoc networking. This work has concentrated on how to make networking based on host addresses (such as IP addresses) work when hosts move around and do not stay attached to a fixed network segment. Mobile IP [7], for example, is a scheme where a host can move to a different location, obtain a temporary IP address there, and cause traffic sent to the host's permanent address to be forwarded to its temporary address. Many ad hoc routing algorithms have been developed to route messages from source to destination through a network of point-to-point connections where the hosts (including the routers) are mobile and thus the connections between hosts are constantly changing [8, 9, 10, 11, 12, 13]. These routing algorithms tend to be complicated and to utilize substantial memory space (code and data), CPU time, and network bandwidth just to maintain the routing information, in addition to what the actual applications utilize.

Work has also been done on multicasting and broadcasting messages in an ad hoc network. Again, this work has focused on routing algorithms for delivering messages to certain specified hosts (multicast) or all hosts (broadcast) through a network of point-to-point connections, where the hosts are mobile and the connectivity changes constantly [14, 15, 16, 17, 18]. Work has also focused on *reliable* multicast and broadcast algorithms which ensure either that all intended destinations receive each message (in the same order, for some algorithms), or that none do [19, 20, 21]. All these algorithms require memory space, CPU time, and network bandwidth to maintain group membership and to enforce reliable message delivery and ordering guarantees.

M2MI and M2MP take a fundamentally different approach. Rather than trying to make address-based networking and routing work in an ad hoc mobile environment, M2MP eliminates the device addresses and groups altogether. Instead, all messages go to all devices within the proximal area (taking advantage of the wireless medium's inherent broadcast nature), and each device decides based on the message's contents whether and how to process the message. Also, M2MP does not guarantee reliable message delivery, error recovery being handled if necessary at higher levels in an application-specific fashion. When the device addresses, groups, and delivery guarantees vanish, so do the memory space, CPU time, and network bandwidth needed for the routing, group maintenance, and reliable delivery algorithms. This drastically simplifies M2MP, making it more attractive for small battery powered devices.

A potential problem with M2MI is a *broadcast storm* [15] where one device broadcasting a message causes other devices to broadcast messages, causing further broadcasts, and so on, leading to contention for the medium and diminished throughput. This problem was observed with *correlated* broadcasts resulting from broadcast-based flood routing. Consequently, M2MI-based applications must be designed to avoid correlated broadcasts.

5.2 Remote Method Invocation

Invocation of methods on remote objects is a well-established technique for constructing distributed systems, realized in distributed object systems like CORBA [22] and Java RMI [5]. Such systems use sending and receiving proxy objects (also called *stubs* and *skeletons*) to translate a method call to a message and back again. Typically, the proxy classes are compiled ahead of time from an interface definition file (as in CORBA) or from the actual Java interface (as in Java RMI). The proxy classes are then installed on all devices participating in the distributed application. Java RMI alternatively lets proxy classes be downloaded from a codebase server at run time, eliminating the need to install the proxy classes during

application deployment.

While remote method invocation is indeed useful, existing distributed object system implementations have two drawbacks. First, pre-compiling and deploying the proxy classes in addition to the regular application classes entails additional effort and more opportunities for making mistakes. With Java RMI, if dynamic proxy downloading is used, a codebase (HTTP) server must be provided, various system properties must be set to point to the codebase URL, and a security policy must be put in place to permit connecting to the codebase server. Judging from the frequent pleas for help on RMI-related message boards, many people have trouble getting all this set up correctly. Also, using codebase URLs is problematic in an ad hoc networking environment where there are no predetermined host addresses and where there may not even be any host that can act as a codebase server.

The second drawback is that downloaded code, including downloaded RMI proxy code, poses a major security risk. While the Java virtual machine and security manager defend against some kinds of attacks, they do not defend against others. For example, downloaded code can mount a *denial of service attack* that crashes the system by allocating all available memory or spawning too many threads [23].

Downloaded code can be digitally signed, and the code can be prevented from executing unless it has a valid signature from a trusted source. However, the signature only verifies who created the code, not whether the code is benign. The signature may not even verify who created the code if the signing computer has been compromised [24]. Trusting downloaded code is especially problematic for a *device* that is expected never to "crash."

While using the same proxy-based technique as existing remote method invocation systems, M2MI avoids the existing systems' deployment and security drawbacks. By synthesizing the M2MI proxy classes directly in the devices where they are used, proxy precompilation, codebase servers, and proxy class downloading are all eliminated. This simplifies M2MI-based application development and deployment, especially in an ad hoc networking environment. Since the M2MI layer synthesizes its own proxies, it can ensure that the proxies do only what they're supposed to do and not anything malicious — without needing to place trust in a code signer.

5.3 Distributed Systems Architecture

Figure 7 shows the design centers of several distributed systems architectures compared to the design center of M2MI. Each architecture is classified along three dimensions: whether the architecture is based on centralized servers; whether the hosts or devices are configured with each other's addresses ahead of time or discover each other dynamically at run time; and the communication patterns among the hosts or devices, one-to-one, one-to-many, or many-to-many.

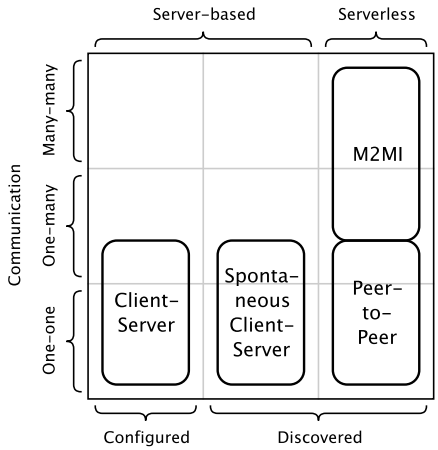

Figure 7: Design centers of distributed systems architectures

The *client-server* architecture is based on a central server whose address (or URL) must be known ahead of time. Most client-server systems use one-to-one communication (e.g. email, web browsing); some use one-to-many communication (e.g. webcasting). While collaborative applications can be and have been built using a client-server architecture, a collaborative application's many-to-many communication pattern doesn't match the client-server architecture's design center. As a result, the application tends to communicate in a "star" pattern where each user's device sends messages to the server and the server then copies the messages to the other devices. In a proximal network with a broadcast medium, sending a separate copy of each message to each device wastes network bandwidth. Also, if the server goes down or becomes inaccessible, the application can no longer operate, even though the devices can communicate with each other directly. Finally, needing to know the server's address ahead of time is problematic in an ad hoc network.

The *spontaneous client-server* architecture eliminates the need for preconfigured addresses by providing a discovery mechanism. Jini Network Technology [25] is a good example. A lookup service runs on one or more server hosts. Clients and services discover the lookup service using a multicast protocol. Services

upload their own proxy objects to the lookup service. Clients download the desired service proxy objects from the lookup service. Clients then invoke methods on the service proxy objects to communicate directly with the services. While this architecture does not require server addresses to be known ahead of time, applications are more complicated to develop because they must discover and interact with the lookup service in addition to their normal functions. Since the architecture still relies on central servers, there's still a mismatch for collaborative applications. Also, Jini in particular relies on downloaded code, which poses a security risk as discussed earlier.

Keeping the spontaneous discovery of services while eliminating the central servers results in a *peer-to-peer* architecture. M2MI is a peer-to-peer architecture oriented around one-to-many and many-to-many communication (although it also supports one-to-one communication). Unlike an application in a client-server architecture, an M2MI-based collaborative application runs collectively in all the participating devices, not on a central server. Thus, an M2MI-based application will not stop operating because a server crashed or became inaccessible. Like a spontaneous client-server architecture, M2MI discovers services dynamically rather than configuring servers' addresses statically. But unlike a spontaneous client-server architecture, M2MI has no central lookup services, and the application does not have to explicitly discover its partners before it can start interacting with them. Rather, the application just goes ahead and broadcasts M2MI method invocations, and whichever partners are out there will respond. This simplifies development and deployment of M2MI-based applications.

5.4 Collaborative Middleware

A number of middleware frameworks for building collaborative applications in ad hoc networks of mobile devices are under investigation. Some frameworks, such as Proem [26, 27] and JXTA [28], follow a *protocol-centric* paradigm in which a standard set of message formats (nowadays typically XML-based) is defined to let devices discover each other, exchange data and events, and otherwise interact with each other. Since the message formats are programming language neutral, applications can be written in different languages to run on heterogeneous platforms and still collaborate. In contrast, M2MI uses only one message "format," that of a method invocation, and overlays that with an object-oriented abstraction in which applications interact by calling methods in interfaces rather than by sending messages. Since M2MI uses dynamic proxy synthesis which the Java platform makes possible, it would be difficult to run M2MI in a heterogeneous environment where some devices lack a Java virtual machine. This, however, is becoming increasingly less of a restriction as more and more devices, including handheld computers, personal digital assistants (PDAs), and cellphones, are shipped with Java.

Other frameworks follow a *data-centric* paradigm. In one.world [29], data are stored in tuples, and applications interact by reading and writing each other's tuples and sending each other events consisting of tuples. Lime [30] is based on "transiently shared tuple spaces" in which each device has a local tuple space, nearby devices merge their local tuple spaces into a shared global tuple space, and applications interact by reading and writing tuples in the shared space.

Different middleware frameworks offer different levels of abstraction. M2MI offers a low-level, method call oriented abstraction. A shared tuple space offers a high-level, data oriented abstraction. In fact, M2MI can be used to implement various high-level middleware frameworks. Applications can then be implemented using the high-level middleware or using M2MI directly. M2MI simplifies the development of high-level middleware frameworks as well as applications in a collaborative ad hoc environment.

6 Status and Future Work

The M2MI paradigm is a work in progress. The sections below describe the current status of M2MP, M2MI, M2MI-based collaborative systems, and security in the M2MI framework. Also described are plans for our ongoing work on M2MI.

6.1 Many-to-Many Protocol

The M2MP protocol has been defined and a prototype protocol stack has been written in Java. The prototype runs on desktop hosts. The prototype code, including a detailed description of the M2MP packet format, is available [31]. The prototype includes a channel implementation (see Section 4.1) that uses UDP datagrams to transport M2MP packets and that uses an IP multicast group as the broadcast communication medium; thus, the "proximal" area consists of all devices listening to the multicast group.

In our continuing work on M2MP, we plan to:

- Construct a channel implementation to transport M2MP directly over a wired Ethernet data link layer (eliminating the unnecessary protocol

overhead of the UDP and IP layers in the prototype).

- Extend the M2MP-over-Ethernet channel implementation to run over a wireless (802.11) Ethernet.
- Construct a channel implementation to transport M2MP over Bluetooth.

The implementations will be written in Java and tested on a desktop host.

6.2 Many-to-Many Invocation

An initial prototype of M2MI has been written in Java. The prototype runs on desktop hosts. The prototype code is available [32]. At present the prototype does not synthesize proxy classes dynamically. Rather, an offline proxy compiler is used to create the proxy classes, which must then be statically incorporated into the application. A library for synthesizing Java classes, the RIT Classfile Library (RCL) [6], has been written.

In our continuing work on M2MI, we plan to:

- Convert the M2MI prototype to use RCL to synthesize proxy classes dynamically.
- Measure the M2MP protocol core, M2MP channel, and M2MI implementations' memory and CPU utilization. Redesign and reimplement them as necessary to reduce the memory and CPU requirements to a level suitable for a small mobile wireless device.
- Study the available PDAs and select one or more with Java capability and 802.11 or Bluetooth wireless connectivity.
- Port the M2MP protocol core, M2MP channel, and M2MI implementations to the selected PDA platform or platforms.
- Test interoperation of M2MP and M2MI from PDA to desktop host and from PDA to PDA.

6.3 M2MI-Based Systems

Initial prototypes of two collaborative applications, chat and whiteboard, have been constructed using M2MI. The prototypes run on desktop hosts. The prototype code is available [32].

From our initial investigations we are getting an inkling of a general paradigm for building collaborative systems using M2MI. Some elements of the paradigm are perceptible, such as participant discovery (see Section 3.3.2) and service discovery (see Section 3.3.3).

We plan to build up experience with and to codify the M2MI paradigm by developing a number of M2MI-based collaborative systems. The systems we plan to develop include:

- Full-featured chat and instant messaging, enabling spontaneous conversations in quiet spaces like libraries and museums
- Full-featured collaborative groupware, including presentation, shared whiteboard, note taking, document authoring by multiple simultaneous authors, file and information sharing, and calendar scheduling features
- Specialized applications for communication in noisy environments such as engine rooms, airfields, flight decks, meeting halls, and restaurants
- Multiplayer games
- Document system utilizing dynamically discovered print services, allowing users to find nearby printers and print from their devices wherever they happen to be
- Surveillance system utilizing dynamically discovered video cameras, allowing users to display images from nearby cameras wherever they happen to be
- Lightweight shared tuple space middleware framework like that of Lime [30]

Each system will be tested on a mixture of desktop and PDA platforms with wired and wireless connectivity.

As we gain experience building M2MI-based systems we plan to flesh out the collaborative system paradigm, devise reusable design patterns, and construct class libraries for building collaborative systems using the paradigm.

6.4 M2MI Security

Providing security within M2MI-based systems is an area for future work. As a starting point, 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, and central servers, it is not clear which techniques would work well in an environment of mobile devices, wireless networks, and no central servers.

Consider, for example, an M2MI-based chat application that supports *closed sessions* where only certain users are allowed to participate. To achieve confidentiality, all the M2MI invocations can be encrypted using a key known only to the chat session members. Ideally, a user should be able to arrive where a closed chat session is taking place, prove that he or she is a member of the group (authentication), obtain the encryption key being used at that time (session key exchange), and start participating in the session. However, authentication and session key exchange systems such as Kerberos [33] rely on central servers that may not be available in an ad hoc device environment.

Building blocks such as the following may be more attractive for M2MI-based applications. Public key exchange protocols, such as Diffie-Hellman key exchange [34], do not require a central server. However, the parties in the exchange must be authenticated to prevent intruder-in-the-middle attacks. Authentication schemes based on zero-knowledge proofs of identity [35, 36, 37, 38] also do not require interacting with a central server. Furthermore, serverless techniques for proving group membership rather than individual identity, such as one-way accumulators [39], eliminate the need to maintain group membership lists on all devices and so may be more attractive in an ad hoc networking environment where all devices are not present all the time. Variations of such schemes based on elliptic curves are especially attractive for small devices, since to obtain a given level of security elliptic curve based algorithms typically require much less storage and processing than algorithms based on integers in a finite field [40].

To begin our investigation of M2MI security, we plan to:

- Conduct a literature search to identify cryptographic algorithms for achieving confidentiality and authentication that are suited for an environment of mobile devices, wireless networks, and no central servers.
- Define modified cryptographic algorithms where the existing algorithms are not well suited for an environment of mobile devices, wireless networks, and no central servers.
- Define elliptic curve based variants of the cryptographic algorithms where necessary (to reduce memory and processing requirements in small devices).
- Analyze how to extend the M2MI infrastructure to provide confidentiality and authentication.

7 Acknowledgments

Jim Waldo inspired the idea for M2MI when he said "Everyone that's out there, call this method" during a discussion about M2MP.

We would like to thank Jeffrey Lasky and Amy Murphy for their comments on earlier drafts of this report.

This research was supported by grants from Sun Microsystems and Xerox Corporation.

References

[1] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[2] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[3] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 44–57, December 1993.

[4] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

[5] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, Fall 1996.

[6] RIT Classfile Library. `http://www.cs.rit.edu/~anhinga/rcl.shtml`.

[7] Internet Engineering Task Force. IP Routing for Wireless/Mobile Hosts (mobileip) Working Group. `http://www.ietf.org/html.charters/mobileip-charter.html`.

[8] Charles E. Perkins and Pravin Bhagwat. DSDV routing over a multihop wireless network of mobile computers. In Tomasz Imielinski and Henry F. Korth, editors, *Mobile Computing*, pages 183–206. Kluwer Academic Publishers, 1996.

[9] David B. Johnson, David A. Maltz, and Josh Broch. DSR: the Dynamic Source Routing protocol for multihop wireless ad hoc networks. In Charles E. Perkins, editor, *Ad Hoc Networking*, pages 139–172. Addison-Wesley, 2001.

[10] Charles E. Perkins and Elizabeth M. Royer. The ad hoc on-demand distance-vector protocol. In Charles E. Perkins, editor, *Ad Hoc Networking*, pages 173–219. Addison-Wesley, 2001.

[11] Zygmunt J. Haas and Marc R. Pearlman. ZRP: a hybrid framework for routing in ad hoc networks. In Charles E. Perkins, editor, *Ad Hoc Networking*, pages 221–253. Addison-Wesley, 2001.

[12] J. J. Garcia-Luna-Aceves and Marcelo Spohn. Bandwidth-efficient link-state routing in wireless networks. In Charles E. Perkins, editor, *Ad Hoc Networking*, pages 323–350. Addison-Wesley, 2001.

[13] J. J. Garcia-Luna-Aceves and Marcelo Spohn. Transmission-efficient routing in wireless networks using link-state information. *Mobile Networks and Applications*, 6(3):223–238, June 2001.

[14] Stefano Basagni, Danilo Bruschi, and Imrich Chlamtac. A mobility-transparent deterministic broadcast mechanism for ad hoc networks. *IEEE/ACM Transactions on Networking*, 7(6):799–807, December 1999.

[15] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*, pages 151–162, August 1999.

[16] Jeffrey E. Wieselthier, Gam D. Nguyen, and Anthony Ephremides. Algorithms for energy-efficient multicasting in static ad hoc wireless networks. *Mobile Networks and Applications*, 6(3):251–263, June 2001.

[17] Sung-Ju Lee, William Su, and Mario Gerla. Wireless ad hoc multicast routing with mobility prediction. *Mobile Networks and Applications*, 6(4):351–360, August 2001.

[18] Seong-Moo Yoo and Zhong-Hua 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`.

[19] Elena Pagani and Gian Paolo Rossi. Reliable broadcast in mobile multihop packet networks. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, pages 34–42, September 1997.

[20] Elena Pagani and Gian Paolo Rossi. Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks. *Mobile Networks and Applications*, 4(3):175–192, October 1999.

[21] Dah Ming Chiu, Miriam Kadansky, Joe Provino, Joseph Wesley, Hans-Peter Bischof, and Haifeng Zhu. A congestion control algorithm for tree-based reliable multicast protocols. Technical Report TR-2001-97, Sun Microsystems, June 2001. `http://research.sun.com/nova/cgi-bin/smli_tr-2001-97.pdf`.

[22] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.4.1.* November 2000.

[23] Gary McGraw and Edward W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

[24] Bruce Schneier. Why digital signatures are not signatures. `http://www.counterpane.com/crypto-gram-0011.html`, November 2000.

[25] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[26] Gerd Kortuem, Stephen Fickas, and Zary Segall. Architectural issues in supporting ad-hoc collaboration with wearable computers. In *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing at the 22nd International Conference on Software Engineering*, June 2000. `http://www.cs.washington.edu/sewpc/papers/kortuem.pdf`.

[27] Gerd Kortuem, Jay Schneider, Dustin Preuitt, Thaddeus G. C. Thompson, Stephen Fickas, and Zary Segall. When peer-to-peer comes face-to-face: Collaborative peer-to-peer computing in mobile ad hoc networks. In *Proceedings of the 2001 International Conference*

on Peer-to-Peer Computing (P2P2001), August 2001. http://www.cs.uoregon.edu/research/wearables/Papers/p2p2001.pdf.

[28] Project JXTA. http://www.jxta.org/.

[29] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE-01-06-01, University of Washington, Department of Computer Science and Engineering, June 2001. http://one.cs.washington.edu/papers/tr01-06-01.pdf.

[30] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'01)*, pages 524–533, April 2001.

[31] Many-to-Many Protocol. http://www.cs.rit.edu/~anhinga/m2mp.shtml.

[32] Many-to-Many Invocation Compiler. http://www.cs.rit.edu/~anhinga/downloads.shtml.

[33] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). Internet Request for Comments (RFC) 1510, September 1993.

[34] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[35] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[36] L. Guillou and J. Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *Advances in Cryptology — EUROCRYPT '88, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 123–128, May 1988.

[37] J. Quisquater, L. Guillou, and T. Berson. How to explain zero-knowledge protocols to your children. In *Advances in Cryptology — EUROCRYPT '89, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 628–631, April 1989.

[38] C. Schnorr. Efficient signature generation for smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[39] John Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology — EUROCRYPT '93, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 274–285, May 1993.

[40] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.