Many-to-Many Invocation: A new Framework for Building Collaborative Applications in Ad Hoc Networks

Hans-Peter Bischof
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
+1 585 475-4994
hpb@cs.rit.edu

Alan Kaminsky
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
+1 585 475-6789
ark@cs.rit.edu

ABSTRACT
Many-to-Many Invocation (M2MI) is a new paradigm 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 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, ad hoc networking, wireless networking.

INTRODUCTION
This paper describes a new 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 service discovery frameworks; groupware for mobile ad hoc collaboration; 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 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 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 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.
- 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.

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 related work and future work.

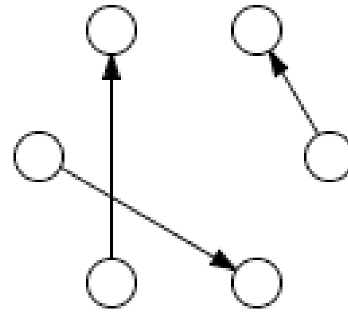
TARGET ENVIRONMENT

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:

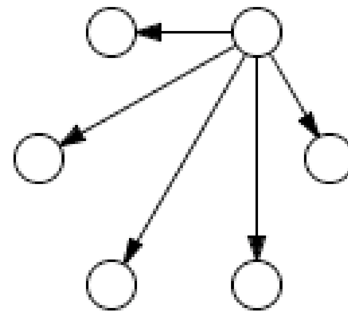
- Applications that discover and use nearby networked services: a document printing application that finds printers wherever the user happens to be, or a surveillance application that displays images from nearby video cameras.
- Collaborative middleware systems like shared tuple spaces [2, 3].
- Groupware 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 webcasting (one-to-many communication), the collaborative systems envisioned here exhibit many-to-many communication patterns (Figure 1). M2MI is designed especially to support applications with many-to-many communication patterns, although it also supports other communication patterns.

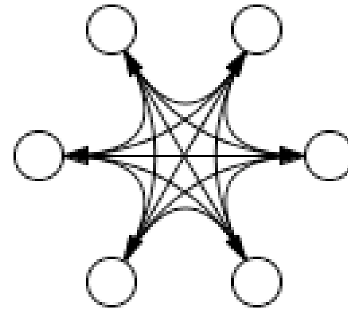
M2MI is also designed to take advantage of a wireless proximal ad hoc networking environment. The devices in the system connect to each other using wireless networking technology such as IEEE 802.11 or Bluetooth. The devices are located in proximity to each other, around the same table or in the same room; every device can hear every other device. Consequently, each transmitted message is immediately received by all the devices without needing to route the message through intermediate devices. Devices come and go as the system is running, the devices do not know each others' identities beforehand; instead, the devices form ad hoc networks among themselves.



One-to-one (email, web browsing)



One-to-many (webcasting)



Many-to-Many (chat, group ware)

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.

M2MI is 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.

Although M2MI is designed to work well in a limited environment of small battery-powered devices, ad hoc wireless networks, and no central servers, M2MI is not confined to that environment. M2MI is perfectly capable of working in an environment of large host computers, wired networks, and central servers.

THE M2MI PARADIGM

Remote method invocation (RMI) [15] 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 certain interface X, called the omnihandle's target interface. (A handle can implement more than one target interface if desired.) Figure 2 depicts an omnihandle for interface `Foo`; the omnihandle is named `allFoos`. It is created by code like this:

```
Foo allFoos =
         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 invocationat different times. Figure 3 shows what happens when the statement `allFoos.y();` is executed. 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 `allFoos` was created, the method is nonetheless invoked on object D.

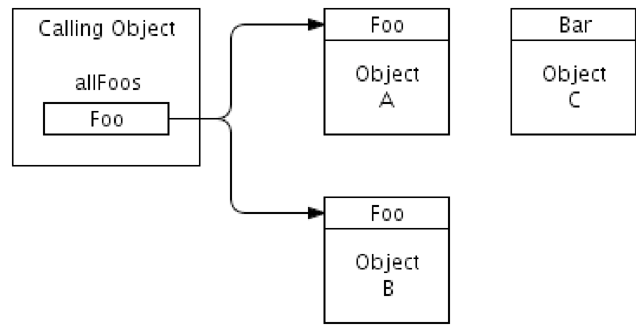
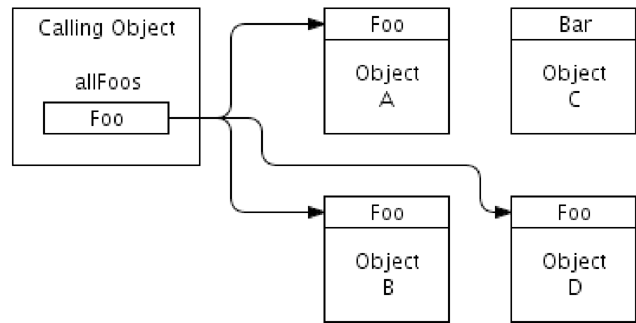


Figure 2: An omnihandle



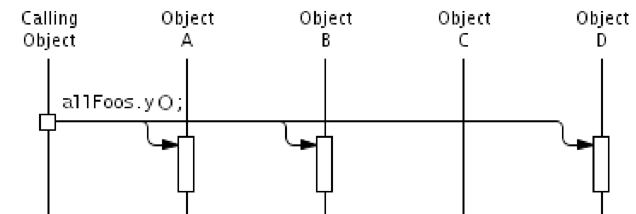


Figure 3: Invoking a method on an omnihandle

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. (M2MI's remote method invocation does not, however, use the same mechanism as Java RMI.)

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. Figure 4 depicts a unihandle for object B implementing interface Foo; the unihandle is named `b_Foo`. It is created by code like this:

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

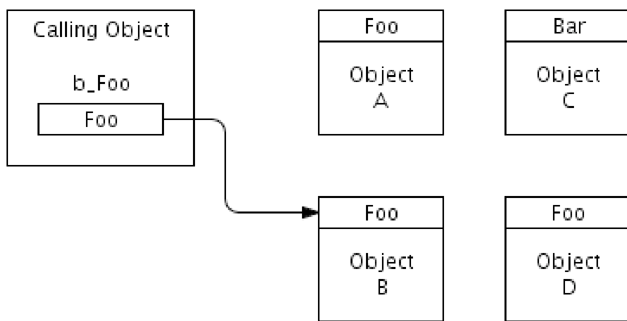


Figure 4: A unihandle

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 shown in Figure 5. 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();
```

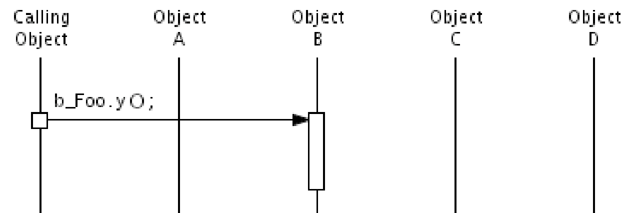


Figure 5: Invoking a method on a unihandle

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.

Figure 6 depicts a multihandle implementing target interface Foo; 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);
```

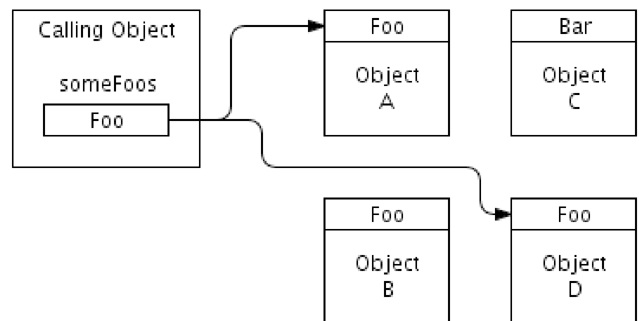


Figure 6: A multihandle

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 shown in Figure 7. 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);
```

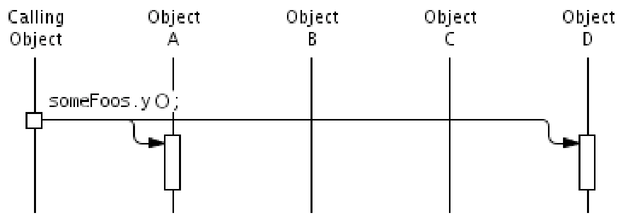


Figure 7: Invoking a method on a multihandle

M2MI-BASED SYSTEMS

This section gives two 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 [7].

Chat

As a first example of an M2MI-based collaborative system, consider a simple chat session: whenever a user types a line of text, the line is displayed on all the users' devices. Each user's chat application has an object implementing this interface:

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

The application exports the chat object to the M2MI layer. The application also obtains from the M2MI layer an omnihandle for interface Chat and stores the omnihandle as `allChats`. Figure 8 shows a sequenceof 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.putMessage ("Hello there");
```

The chat object's implementation of the `putMessage` 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 putMessage invocations. This shows how M2MI simplifies the development of collaborative systems.

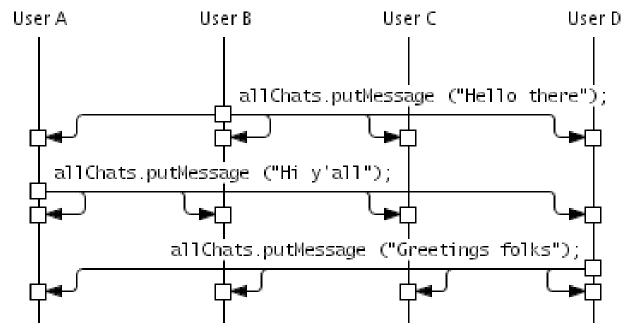


Figure 8: M2MI invocations for a chat application

Note that the M2MI-based chat application does not need to find and connect to a central chat server.

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. Figure 9 shows the sequence of M2MI invocations that occur when the document printing application goes to print a document with three printers nearby.

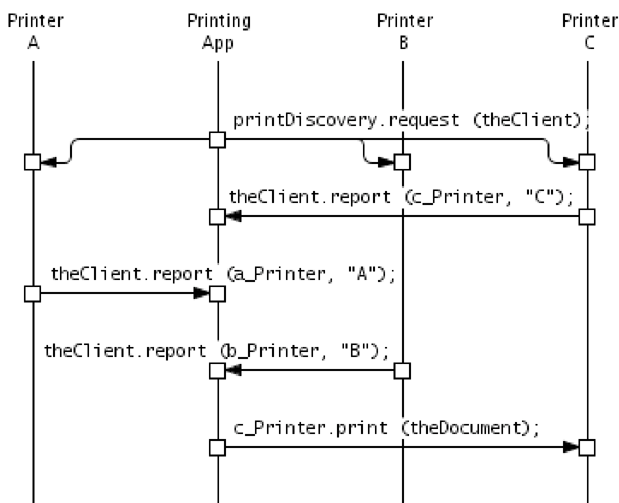


Figure 9: M2MI invocations for a print service

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 requesting 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.

SUMMARY

The examples in this section have shown how objects implementing simple interfaces, coupled with M2MI's ability to invoke methods on many objects at once, coupled with M2MI's ability to invoke methods on many objects at once, can be used to build different kinds of powerful and interesting ad hoc collaborative systems. None of the systems required central servers; none of the systems required knowledge of individual device addresses. All of the systems allowed new devices to join the collaborative group simply by showing up and starting to broadcast M2MI invocations, without needing to perform explicit discovery or group joining protocols. M2MI is thus well suited to ad hoc networks of small mobile devices.

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.

Figure 10 shows the overall architecture of M2MI. When the calling object invokes a target method on a handle, the invocation may have to be performed by target objects in three places: in the same process as the calling object, in different processes in the same device, and in different devices. The invocation travels along different paths to the three destinations.

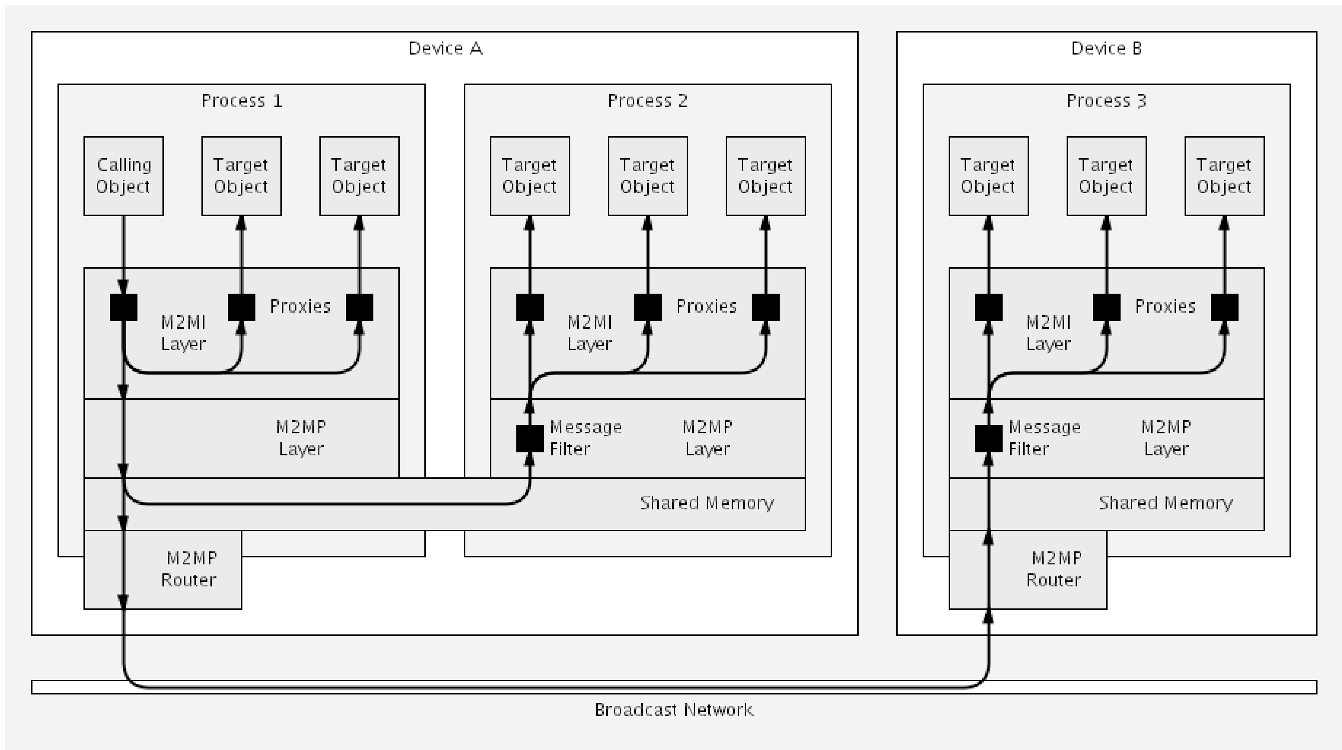


Figure 10: M2MI Architecture

Each process that employs M2MI has a singleton instance of the M2MI layer, and the M2MI layer has an instance of the M2MP layer. When the calling object invokes a target method on a handle, the handle forwards the invocation to the M2MI layer in its own process. The M2MI layer in turn forwards the invocation to the appropriate objects that have been exported to the M2MI layer in that process, if any. No messages are sent out of the process to reach these objects. To reach target objects in other processes, the M2MI layer transmits the invocation in the form of a message (byte stream) to the M2MP layer. All the M2MP layers in the same device share a region of memory. The transmitting M2MP layer deposits the invocation message into the shared memory. The other M2MP layers each obtain a copy of the invocation message from the shared memory and pass the message up to their respective M2MP layers. No messages are sent out of the device onto an external network to reach these objects. Are more detailed description of the design and architecture can be found at [7].

RELATED WORK

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

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 [4], 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 [1, 13, 16, 11, 18,]. 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.

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. This drastically simplifies M2MP, making it more attractive for small battery powered devices.

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 [14] and Java RMI [15]. 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.

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. 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 [12].

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 [16]. Trusting downloaded code is especially problematic for device that is expected never to ``crash."

While using the same proxy-based technique as existing remote method invocation systems, with the handles and the method invokers taking the roles of the sending and receiving proxies, M2MI avoids the existing systems' deployment and security drawbacks. By synthesizing the M2MI proxy classes directly in the devices where they are used, proxy pre-compilation, 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.

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 [9, 10] and JXTA [5] follow 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.

For a more detailed discussion see [7].

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 describedare plans for our ongoing work on M2MI.

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 [6]. The prototype includes several channel implementations that use UDP datagrams to transport M2MP packets. In our continuing work on M2MP, we plan to construct several additional channel implementations. One channel implementation, currently being developed, will transport M2MP directly over a wired Ethernet data link layer, eliminating the unnecessary protocol overheadof the UDP and IP layers in the prototype [8]. This channel implementation will then be extended to run over a wireless (802.11) Ethernet. Another channel implementation will transport M2MP over Bluetooth. The implementations will be written in Java, along with native code where necessary, and tested on a desktop host.

Once these implementations are working, we plan to migrate the M2MP protocol stack, including the shared memory layer, into the Linux operating system kernel. This will reduce the overhead and improve the performance of M2MP.

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.

M2MI-Based Collaborative Applications

We will develop a wide range of collaborative applications using M2MI. Based on the experience gained, codify frameworks and design patterns for developing M2MI-based collaborative applications.

M2MI Tools

We will develop tools to assist in the design, coding, and testing of M2MI-based applications.

ACKNOWLEDGMENTS

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

The following students at the Rochester Institute of Technology have helped us investigate the M2MI infrastructure and have built prototype M2MI-based applications: Adam Bazinet, Joseph Binder, Steve Button, Tom Chang, Dan Clark, Jonathan Coles, Frank Conover, Louie Gosselin, Kiran Hegde, John King, Brian Koponen, Kevin Mooney, Jeffrey Myers, Ravi Nareppa, Jim Papapanu, Tri Phan, Jacob Rigby, Girish Sarma, Anthony Stamp, Evan Teran, Hau San Si Tou, Ken VanderVeer, Merit Wilkinson, and Josh Zatulove.

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

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

REFERENCES

1. S. Basagni, D. Bruschi, and I. Chlamtac. A mobility-transparent deterministic broadcast mechanism for ad hoc networks. *IEEE/ACM Transactions on Networking*, 7(6):799-807, December 1999.
2. N. Carriero and D. Gelernter. How to Write Parallel Programs: A First Course. MIT Press, 1990.
3. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, January 1985.
4. Internet Engineering Task Force. IP Routing for Wireless/Mobile Hosts (mobileip) Working Group. http://www.ietf.org/html.charters/mobileip-charter.html.
5. Project JXTA. http://www.jxta.org/.
6. A. Kaminsky. Many-to-Many Protocol. http://www.cs.rit.edu/~anhinga/m2mp.shtml.
7. 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
8. K. Hedge. M2MP Over Ethernet http://www.cs.rit.edu/~kxh6049/
9. G. Kortuem, S. Fickas, and Z. 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.
10. G. Kortuem, J. Schneider, D. Preuitt, T. G. C. Thompson, S. Fickas, and Z. 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.
11. S.-J. Lee, W. Su, and M. Gerla. Wireless ad hoc multicast routing with mobility prediction. Mobile Networks and Applications, 6(4):351-360, August 2001.
12. G. McGraw and E. W. Felten. Securing Java: Getting Down to Business with Mobile Code. John Wiley & Sons, 1999.
13. S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. 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.
14. Object Management Group. The common object request broker: Architecture and specification, revision 2.4.1, November 2000.

15. R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the Java system. Computing Systems, 9(4):291-312, Fall 1996.
16. B. Schneier. Why digital signatures are not signatures. http://www.counterpane.com/crypto-gram-0011.html, November 2000.
17. A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. Computing Systems, 9(4):265-290, Fall 1996.
18. S.-M. Yoo and Z.-H. Zhou. All-to-all communication in wireless ad hoc networks. In Proceedings of the 39th Annual *ACM Southeast Conferenc*e, pages 180-181, March 2001. http://webster.cs.uga.edu/~jam/acm-se/review/abstract/syoo.ps.