11-2002

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

Alan Kaminsky
Rochester Institute of Technology

Hans-Peter Bischof
Rochester Institute of Technology

Recommended Citation
Alan Kaminsky and Hans-Peter Bischof. 2002. Many-to-Many Invocation: a new object oriented paradigm for ad hoc collaborative systems. In Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02). ACM, New York, NY, USA, 72-73. DOI=http://dx.doi.org/10.1145/985072.985109

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

Alan Kaminsky
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
ark@cs.rit.edu

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

July 16, 2002

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 multiuser applications (conversations, groupware, multiplayer games); 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." An M2MI-based application is built by defining one or more interfaces, creating objects that implement those interfaces in all the participating devices, and broadcasting method invocations to all the objects on all the devices. 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. M2MI synthesizes remote method invocation proxies dynamically at run time, eliminating the need to compile and deploy proxies ahead of time. As a result, 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.

1 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 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 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 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.

The paper 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 gives several examples of ad hoc collaborative systems based on M2MI. Sections 5, 6, and 7 describe the architecture and design of the M2MI software. Section 8 compares and contrasts M2MI with related work. Section 9 discusses the current status of M2MI and plans for further work.

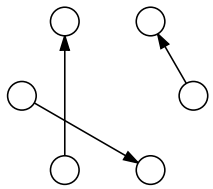
2 Target Environment

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:

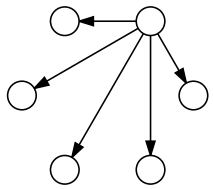
- Multiuser applications: a chat session, a shared whiteboard, a group appointment scheduler, a file sharing application, or a multiplayer game.
- Applications that discover and use nearby networked services: a document printing application that finds printers wherever the user happens to be, or a surveillance application that displays images from nearby video cameras.
- Collaborative middleware systems like shared tuple spaces [14, 7].

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.

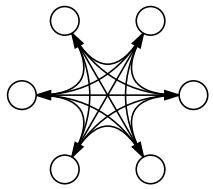
M2MI is also designed to take advantage of a wireless proximal ad hoc networking environment. The devices in the system connect to each other using *wireless* networking technology such as IEEE 802.11 or Bluetooth. The devices are located in *proximity* to each other, around the same table or in the 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, and the devices



One-to-one
(email, web browsing)



One-to-many
(webcasting)



Many-to-many
(chat, groupware)

Figure 1: Communication patterns

do not know each others' 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.

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.

To reduce the amount of network traffic, M2MI takes advantage of the *broadcast* communication made possible by a wireless proximal network. In a collaborative system with n devices where every device sends messages to every other device, if messages had to be sent between individual devices, the number of messages would be proportional to n^2. But since M2MI uses broadcast messaging, the number of messages sent is only proportional to n. This also improves the scalability of M2MI, since the network traffic tends to increase linearly rather than quadratically as more devices join an M2MI-based system.

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.

3 The M2MI Paradigm

Remote method invocation (RMI) [49] 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.

3.1 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.

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

Once an omnihandle is created, calling method Y on the omnihandle for interface X means, "Every object out there that implements interface X, perform method Y." The method is actually performed by whichever objects implementing interface X exist at the time the method is *invoked* on the omnihandle. Thus, different objects could respond to an omnihandle invocation at different times. Figure 3 shows what happens when the statement `allFoos.y();` is executed. Three objects implementing interface `Foo`,

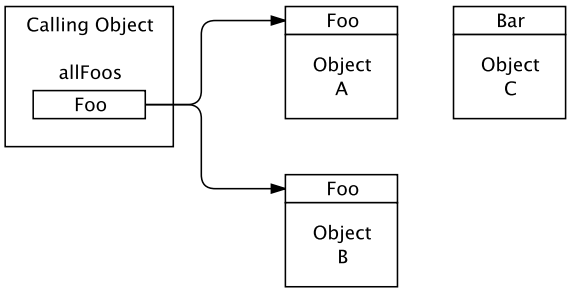


Figure 2: An omnihandle

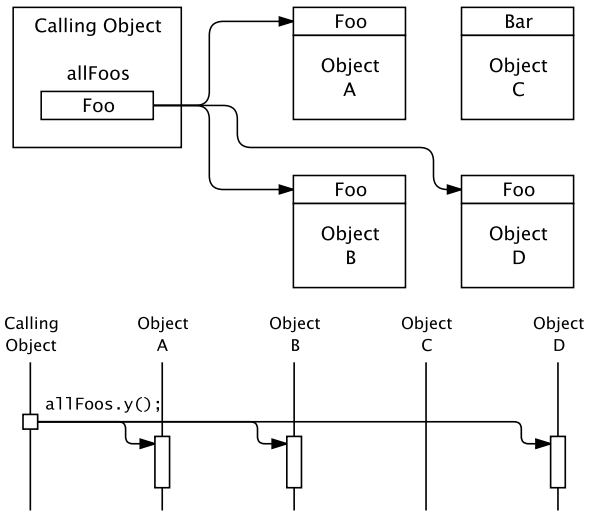


Figure 3: Invoking a method on an omnihandle

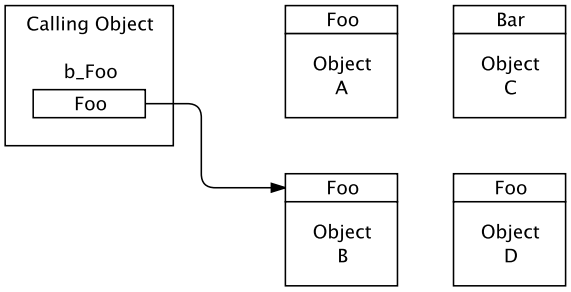


Figure 4: A unihandle

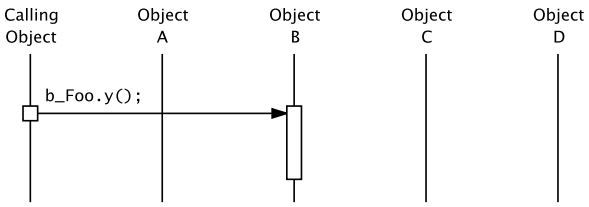


Figure 5: Invoking a method on a unihandle

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.

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.)

3.3 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 [15, 1] explicit in the interface.

3.4 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:

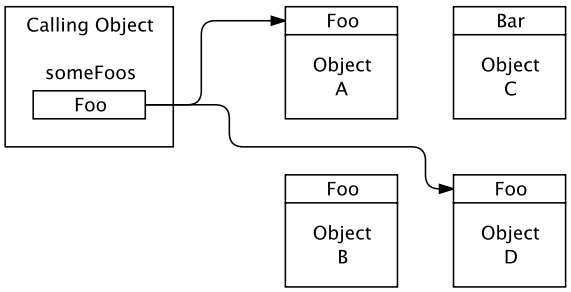


Figure 6: A multihandle

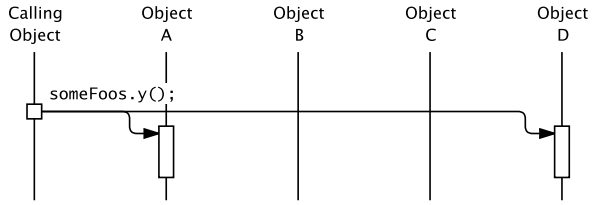


Figure 7: Invoking a method on a multihandle

```
Foo b_Foo = (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 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();
```

3.5 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 = (Foo) 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 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);
```

3.6 Characteristics of M2MI Invocations

Methods in interfaces invoked via M2MI can have arguments. When an object of a non-primitive type, including an array 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. An omnihandle or multihandle can also be passed as an M2MI method call argument, and it behaves the same way in the method call recipient as it does in the caller. Primitive types are always passed by copy in M2MI.

M2MI uses Java's object serialization to marshal the method call arguments on the calling side and unmarshal them again on the target side. Accordingly, every non-primitive object passed in as an M2MI method call argument must be serializable, or the invocation will fail.

While M2MI can pass objects as arguments like Java RMI, M2MI does not download the argument objects' classes to the destination as Java RMI does. With M2MI, the destination must already possess the argument objects' classes, or the invocation will fail. If a handle is passed as an argument in an M2MI method call, though, the destination need only possess the handle's target interface or interfaces. (The

destination's M2MI layer already possesses all the classes needed to implement handles.)

Although they can have arguments, methods in interfaces invoked via M2MI must be declared not to return a value and not to throw any exceptions. This is because with potentially more than one object performing the method, there is no single return value or exception to return or throw.

Since an M2MI method does not return anything, the caller cannot get any information back from the called object *in the same method call.* If the caller needs to get information back, the caller can send a reference to its own object by passing the object's unihandle as an argument to a method invoked on a handle. The called object or objects can then send information back by performing subsequent method invocations on the original caller's unihandle. This typically leads to a pattern of *asynchronous* method calls and callbacks in an M2MI-based application as shown in the examples in Section 4; in other words, an *event-driven* application.

For the same reason, an M2MI method invocation does not give any indication of whether the invocation was successfully communicated to the called objects. If an M2MI-based application needs acknowledgments that a method call in fact reached the called objects, the called objects must do separate method invocations back to the calling object. However, some applications can be designed not to need explicit method acknowledgments at all, achieving fault recovery by other means, as shown in Section 4.

Finally, M2MI invocations are *non-blocking.* A method call on a handle returns immediately to the calling object without waiting for all the target objects to execute their methods. Later, when the method invocations are actually performed, every method in every target object is (potentially) executed concurrently by a separate thread. Therefore, every object exported via M2MI must be designed to be multiple thread safe. Furthermore, like any concurrent application, the overall M2MI-based application must be designed to avoid deadlocks, to work properly with any ordering of the concurrent method calls, and so on.

4 M2MI-Based Systems

This section gives several examples showing how M2MI can be used to design a broad range of ad hoc collaborative systems.

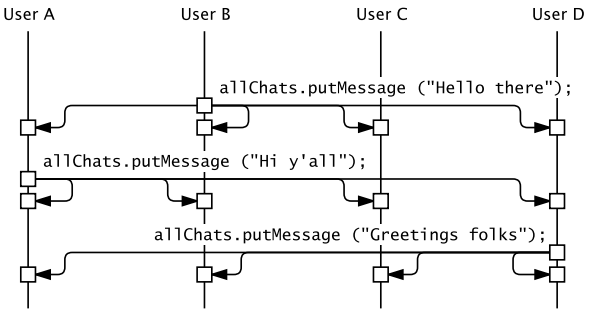


Figure 8: M2MI invocations for a chat application

4.1 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 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.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.

Appendix A gives the actual, working, extremely simple Java code for this M2MI-based chat application.

4.2 Multiple Chat Sessions

Let us add a feature to the chat application: multiple independent simultaneous chat sessions. The user can discover which chat sessions are out there and participate in one of them, or the user can start a new chat session. The user then sees only the messages sent to that chat session, not all the other chat sessions.

To see only the messages for a particular chat session, each user's device has a chat object implementing interface `Chat` as before. Now, however, there is a *multihandle* for interface `Chat` for each separate chat session. To participate in a particular chat session, the application attaches its chat object to the corresponding multihandle. When the user types a line of text, the application invokes `putMessage` on the chat session's multihandle. The chat object processes a `putMessage` invocation exactly as before, by adding the line of text to the chat session log. However, since the invocation is performed on a multihandle instead of an omnihandle, only those chat objects that have been explicitly attached to the multihandle — that is, only those devices participating in the chat session — will respond.

To discover which chat sessions are out there, a new interface is used:

```
public interface ChatDiscovery {
   public void report (Chat session,
      String name);
}
```

The application exports a chat discovery object implementing interface `ChatDiscovery`. Each device with an active chat session periodically invokes `report` on an omnihandle for interface `ChatDiscovery`, passing in the multihandle for the chat session and the name of the chat session. Processing each `report` invocation, the chat discovery object collects the chat sessions in a list and displays them for the user to choose.

If the user decides to participate in an existing chat session, the application obtains the corresponding chat session multihandle from the list and attaches the chat object to the multihandle. If the user decides to start a new chat session, the application creates a new chat session multihandle and attaches the chat object to the multihandle. In either case, the application starts calling `report` periodically.

Figure 9 shows a sequence of M2MI invocations that might occur when four instances of this chat application run in four nearby devices. Users A and C are participating in one chat session named `"AC"`, while users B and D are participating in another named `"BD"`. The corresponding chat session

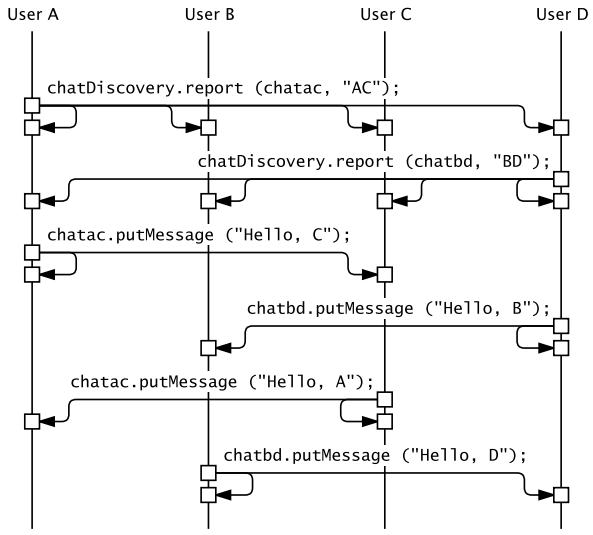


Figure 9: M2MI invocations for multiple chat sessions

multihandles are named `chatac` and `chatbd`. The omnihandle for interface `ChatDiscovery` is named `chatDiscovery`.

As long as there is at least one device participating in a particular chat session, the periodic `report` invocations for that chat session will continue. When the last participant in the chat session vanishes, the periodic `report` invocations cease. If a certain chat session (multihandle) has not been reported for some amount of time, all the chat discovery objects in all the devices remove that chat session from their lists.

A slight modification of the above scheme will reduce the network traffic. It is not necessary for *every* device participating in a particular chat session to perform `report` invocations for that chat session; only one device need do so. Accordingly, each application starts a timeout for a randomly chosen time interval before doing the next `report` invocation. If someone else calls `report` for the application's chat session before the timeout occurs, the application merely restarts the timeout for another randomly chosen time interval without bothering to call `report` itself. But if the timeout occurs before someone else calls `report` for the application's chat session, the application calls `report` and then restarts the timeout for a randomly chosen time interval.

When a new device arrives in an area where chat sessions are in progress, it may be some time before other devices call `report` and the new device discovers the existing chat sessions. To speed up the discovery process, add a method to the `ChatDiscovery` interface:

```
public interface ChatDiscovery {
   public void request();
   public void report (Chat session,
      String name);
}
```

When the chat application starts up, or when it notices that no one has called `report` for a while, the application calls `request` on an omnihandle for interface `ChatDiscovery`. Processing the invocation, the chat discovery objects in the other devices report their respective chat sessions by calling `report` immediately rather than waiting until the next timeout. However, to avoid a *broadcast storm* [35] where all the devices start calling `report`, saturating the network, only one device in each chat session should respond. Accordingly, every chat discovery object starts a timeout for a small nonzero random amount of time. In each chat session, the first chat discovery object to time out calls `report`. Processing that invocation, the newly arrived device adds the reported chat session to its list as usual, while the other chat discovery objects in that chat session refrain from calling `report` and restart their timeouts for the next report, as usual.

4.3 Chat Log Recovery

Let us add another feature to the chat application. Suppose one or more participants step out of the room, taking their devices with them, so their devices go out of range of the proximal wireless network and no longer receive M2MI invocations. When the participants step back into the room, their devices should automatically fill in the gaps in their chat logs with all the messages they missed while they were out of range, as well as displaying new chat messages. In other words, each device should synchronize its chat log with all the other devices. Let us also assume that each device's chat log only needs to hold the most recent n messages; once the chat log fills up with n messages, older messages don't need to be recovered, and each time a new message is added, the oldest message is deleted.

To add this feature, the chat application needs two things: a way to detect that its chat log does not contain all the messages that another device's chat log contains, and a way to obtain copies of the missing chat messages and put them in their proper places in the chat log.

To detect gaps in the chat log, assign a sequence number to each chat message, and change the `Chat` interface to this:

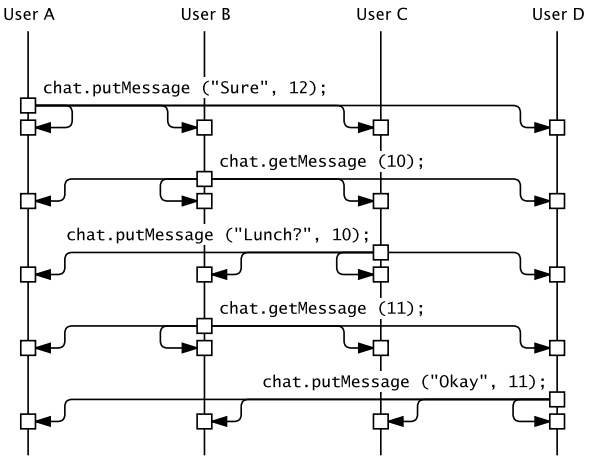


Figure 10: M2MI invocations for chat log recovery

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

When a device calls `putMessage`, it passes in the line of text and a sequence number 1 higher than the most recently received chat message. When a device processes a `putMessage` invocation, the device records the new message and its sequence number in the chat log.

If two or more devices call `putMessage` concurrently, then two or more chat messages will end up with the same sequence number. Let us defer dealing with this situation until later and assume for the moment that only one device at a time ever calls `putMessage`.

If the most recently received sequence number is k, and the chat log can hold at most n messages, then the chat log must always contain messages numbered from $\max(1, k-n+1)$ to k. If, after processing a `putMessage` call, a device notices it doesn't have all the chat messages with sequence numbers in that range, the device must synchronize its chat log. To let it do so, the `Chat` interface needs another method:

```
public interface Chat {
   public void putMessage (String line,
      long seqnum);
   public void getMessage (long seqnum);
}
```

Figure 10 shows the sequence of M2MI invocations to synchronize the chat log in device B. (All devices are participating in the same chat session, and all the method invocations are performed on a multihandle for interface `Chat` named `chat`.) Processing a `putMessage` invocation, device B notices

it doesn't have all the chat messages implied by the new sequence number. So device B calls `getMessage`, specifying the sequence number of one chat message it needs. To avoid a broadcast storm, only one device should respond. Accordingly, every device's `getMessage` method starts a timeout for a small nonzero random amount of time. The first device to time out calls `putMessage`, passing in the text of the requested chat message and its sequence number. Executing the `putMessage` method, device B records the chat message and its sequence number at the proper place in the chat log, while the other devices cancel their timeouts. Device B continues in this way until all the gaps in its chat log are filled.

A newly arrived device must be able to determine the most recent chat message's sequence number, so the device can pass the correct sequence number in subsequent `putMessage` invocations. If some other device calls `putMessage`, that would supply the needed information. But in case no one is calling `putMessage`, the `Chat` interface needs a third method to let the device request the information explicitly:

```
public interface Chat {
   public void putMessage (String line,
      long seqnum);
   public void getMessage (long seqnum);
   public void getLatestMessage();
}
```

A device calls `getLatestMessage` whenever there have been no M2MI invocations in the chat session for a certain length of time. Responding to the `getLatestMessage` invocation, one of the devices (whichever one times out first) calls `putMessage`, passing in the text of the most recent chat message and its sequence number. The requesting device now knows the most recent sequence number and can also start synchronizing its chat log if necessary.

As will be discussed later, the M2MI invocations may be transported by a network protocol that is not totally reliable, so an invocation may occasionally be lost. To recover from a lost invocation, the device starts a timeout after calling `getMessage` or `getLatestMessage`. If `putMessage` is not called before the timeout, the device retries the invocation; if a certain number of retries all time out, the device gives up.

Now let us return to the issue of multiple chat messages with the same sequence number, resulting from multiple devices calling `putMessage` concurrently. We could impose a protocol to guarantee that every chat message gets a unique sequence number, but that seems hopelessly complicated, especially in an ad hoc network where devices can arrive and depart at any time. Instead, we'll relax the restriction and allow multiple chat messages to have the same sequence number. We'll also allow the devices to display chat messages with the same sequence number in any order, as long as they come after the next lower sequence number and before the next higher sequence number.

Consequently, when responding to a `getMessage` or `getLatestMessage` call, a device may possess more than one chat message corresponding to the requested sequence number. In that case the device calls `putMessage` multiple times, with the same sequence number but different message texts each time.

It may also happen that the first device to respond to a `getMessage` or `getLatestMessage` call has a set of chat messages for the requested sequence number that differs from another device's. To handle that case, the other devices monitor the first device's `putMessage` responses. If the other devices detect that they would have responded differently from the first device, the other devices also call `putMessage`.

Finally, it may happen that all the devices' chat logs have the same range of sequence numbers with no gaps, but the chat message texts are different for some sequence number or numbers in different devices. This can happen if the devices separate into multiple groups that are out of wireless range of each other, the chat session continues in each separate group, then the devices come back together again. Since the devices' chat logs all have the same range of sequence numbers, nothing will trigger any device to start a synchronization. To deal with this probably rare case, a device occasionally forces a synchronization by issuing a series of `getMessage` calls for all the sequence numbers in the chat log.

Note that the chat application's log synchronization capability, intended primarily to bring newly arrived devices up to speed, also serves to recover from communication failures. If a device fails to receive a `putMessage` invocation because a network message was lost, on the next `putMessage` invocation the device will detect the missing sequence number and start a synchronization. Even if the network were totally reliable, the chat application would still need the log synchronization capability to deal with newly arrived devices. Therefore, it doesn't make sense to add a lot of code at the network layer to make network communication totally reliable. End-to-end reliability has to be built in at the application level [8].

4.4 Instant Messaging

As another example of an M2MI-based collaborative system, consider a simple instant messaging (IM) system. The IM application needs to discover which

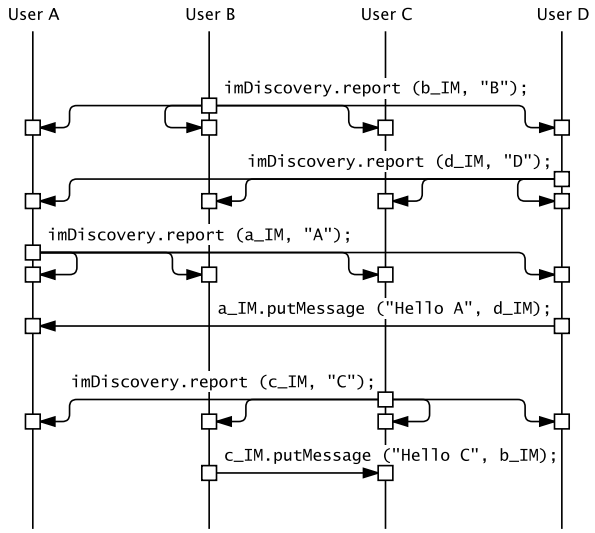


Figure 11: M2MI invocations for an IM application

users are out there and send messages to individual users (unlike the chat application which sends messages to all users in a chat session).

The interfaces for the IM application are quite similar to those for the chat application, interface `IMDiscovery` to discover users and interface `IM` to send messages:

```
public interface IMDiscovery {
   public void request();
   public void report (IM user,
      String name);
}
public interface IM {
   public void putMessage (String line,
      IM sender);
}
```

The IM application exports an IM object implementing interface `IM`. But since instant messages go to only one place, the IM object is attached to a *unihandle* (not a multihandle as in the chat application). The IM application also exports an IM discovery object implementing interface `IMDiscovery`.

Figure 11 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 `report` on an omnihandle for interface `IMDiscovery`, passing in the unihandle to its own IM object and its own user's name. For example, user A's application calls:

```
imDiscovery.report (a_IM, "A");
```

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

To send an instant message to a particular user, the application looks up the corresponding `IM` unihandle in the user list and calls the `putMessage` 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). For example, to send an instant message to user A, user D's application calls:

```
a_IM.putMessage ("Hello A", d_IM);
```

The `putMessage` method displays the message and the sender on the destination device's display. Since the invocation is performed on a unihandle, not a multihandle or omnihandle, only the destined user's IM object executes the `putMessage` 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 `report` invocation periodically on an omnihandle for interface `IMDiscovery`. If the time since the last `report` invocation for a certain user (unihandle) exceeds a threshold, the other IM applications conclude the user has gone away and remove the user from their user lists. To quickly discover which users are present, a device invokes `request` on an omnihandle for interface `IMDiscovery`, and all the IM discovery objects respond by calling `report` immediately.

4.5 Service Discovery — Printing

As an example of an M2MI-based system involving standalone 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);
}
```

In the chat and IM applications, the participating devices all played the same roles, both making discovery requests and generating discovery reports. In the printing application, though, the participating devices do not play the same roles. Some devices are clients which make discovery requests but do not generate discovery reports; other devices are printers which generate discovery reports but do not make discovery requests. Accordingly, in the printing system there is a separate discovery interface for each role.

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 12 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

```
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.

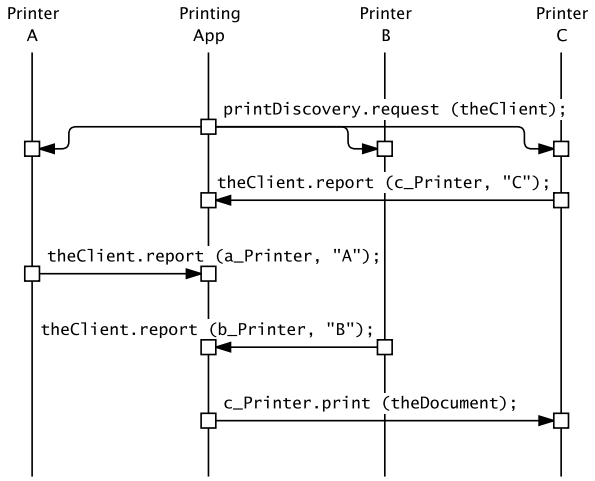


Figure 12: M2MI invocations for a print service

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.

4.6 Advanced Printing

When printing a document, the user may need the printer to have certain features — such as the ability to print multiple copies of a document, or the ability to staple the pages, or having a certain size of paper loaded. Alternatively, the user may need the printer to be in a certain state — such as not jammed, or not too many jobs backed up in the print queue. In such cases, the user wants to discover only the printers that fulfill the user's criteria, not all the printers. Furthermore, when actually printing the document, the user wants to specify the number of copies, stapling, paper size, and other characteristics of the print job in addition to the document itself.

To accomplish this, add some methods to the `PrintDiscovery` interface and to the `PrintService`

interface:

```
public interface PrintDiscovery {
   public void request
      (PrintClient client);
   public void request
      (PrintClient client,
       Attribute attr);
   public void request
      (PrintClient client,
       AttributeSet attrs);
}
public interface PrintService {
   public void print (Document doc);
   public void print (Document doc,
      Attribute attr);
   public void print (Document doc,
      AttributeSet attrs);
}
```

The various printer characteristics — copies, stapling, paper, printer status, print queue status, and so on — are all represented as *attributes.*[1] To discover printers that have, say, ISO A4 paper loaded, the printing application invokes the second `request` method instead of the first, passing in the desired attribute:

```
printDiscovery.request (theClient,
   MediaSize.ISO.A4);
```

While all the printers still execute this method in response to the omnihandle invocation, only those printers that match the attribute — namely, those that have ISO A4 paper loaded — will call back to the client. Likewise, to discover printers that support multiple attributes, the printing application invokes the third `request` method, passing in a set of the desired attributes; only those printers that match all the attributes will respond. Consequently, the client becomes aware of just those printers that match the user's requirements.

To specify job characteristics for the actual print job, the printing application invokes the second or third `print` method instead of the first, passing in the desired attribute or set of attributes:

```
c_Printer.print (theDocument,
   MediaSize.ISO.A4);
```

This example shows that, by defining the appropriate interfaces, service discovery can be tailored specifically for the service.

[1]The Internet Printing Protocol (IPP) [19] defines a rich set of printing attributes. For examples of Java APIs that encapsulate the IPP printing attributes, see the Jini Print Service API [22] and the Java Print API [47].

4.7 File Sharing

As a final example of an M2MI-based collaborative system, imagine a file sharing application. Each user's device has a number of files which the user is willing to share. When the file sharing application runs among a group of proximal devices, the user sees all available shared files — that is, the union of the sets of shared files in all the devices. If a certain file exists on more than one device, that file shows up only once in the application. As devices enter and leave the proximal group, the set of shared files visible on each device grows and shrinks. The user can get information about any shared file, such as its name, its type, its size, a textual annotation, a thumbnail view, and so on. The user can also browse the contents of any shared file — read a text file, view an image file, play a sound file.

When a device leaves the proximal group, from that device's point of view all the shared files disappear except for those stored on the device itself. However, before leaving, the user can tell the file sharing application to *keep* a certain file or files. The application stores a copy of those files on the user's device (which does not change the set of shared files as viewed by all the devices). Now, however, the kept files do not disappear when the device leaves the proximal group.

An ad hoc collaborative file sharing application can be used in many ways. Spectators in public settings like athletic competitions, sporting events, amusement parks, and scenic places can share the digital photos they're all taking. A group of friends can listen to one another's music files, or swap copies of music files.[2] Businesspeople in a meeting can share reports, presentations, contact information, and so on.

To detect whether the same file exists on multiple devices without having to transfer the entire files over the network, the file sharing application uses a *one-way hash of the file's contents* (such as an MD5 hash [43] or SHA-1 hash [34]) to uniquely identify a file. Meta-information about a file, such as its name or type, is not part of the file's contents. Thus, two files with different names but the same contents will have the same IDs (hashes) and will be considered the same file.

Each file sharing application exports an object implementing this interface:

```
public interface FileShare {
   public void available (Hash[] ids);
   public void requestFile (Hash id);
```

[2]Always provided, of course, that the files are legally allowed to be copied.

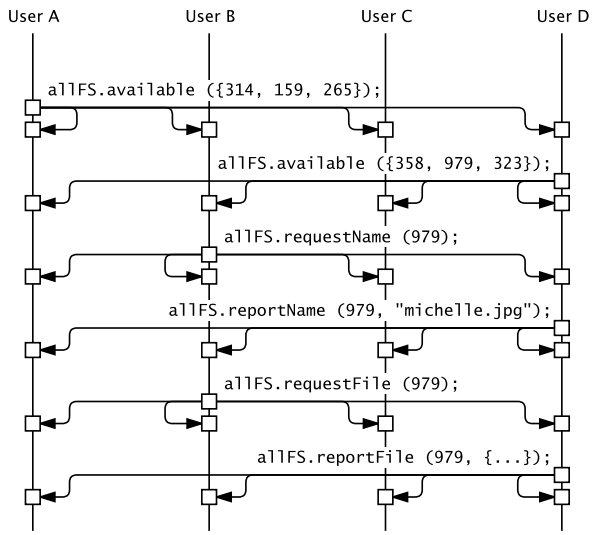


Figure 13: M2MI invocations for a file sharing application

```
    public void reportFile (Hash id,
       byte[] contents);
    public void requestName (Hash id);
    public void reportName (Hash id,
       String name);
}
```

Figure 13 shows a sequence of M2MI invocations that might occur when four instances of this file sharing application run in four nearby devices. Every device periodically performs an omnihandle invocation of the `available` method, passing in an array of the IDs of the files it has to share. (The array of hashes takes much less time to transmit than the files themselves.) Processing the `available` method, each device adds the specified IDs to its list of available IDs, except for those already in the list. Each device also starts or re-starts a timeout for each of the specified IDs. If a device leaves the proximal group, the device stops doing `available` invocations, the IDs for that device's files time out (unless some other device is reporting they're still available), and each remaining device removes those IDs from its list of available IDs.

To find out further information about a particular file, such as its name, the device performs an omnihandle invocation of the `requestName` method, passing in the desired file's ID. To prevent a broadcast storm if multiple devices possess that file, the `requestName` method in each device that has the file starts a small nonzero random timeout. The first device to time out performs an omnihandle invocation of the `reportName` method, passing in the requested file's ID and name; the other devices if any cancel their timeouts. The requesting device now has the file name. Clearly, any other piece of meta-information can be provided by adding the appropriate `request` and `report` methods to the `FileShare` interface.

In the same way, to get the actual contents of a particular file, the device calls `requestFile` on the omnihandle. One of the devices possessing the file then calls `reportFile` on the omnihandle, passing in the file's contents. Executing the `reportFile` method, the requesting device can display the file, store it locally on the device (to keep the file), and so on.

Besides reducing the time needed to transmit unique file identifiers around the network, using one-way hashes to identify files provides a measure of security. An intruder could try to disrupt the file sharing application by sending some file other than the requested file in a `reportFile` invocation. However, if the reported ID (hash) does not match the actual hash of the reported contents, the recipient knows the contents are not correct and can discard them. While the intruder's bogus `reportFile` invocation does consume network and processing resources, it does not cause the application to behave incorrectly.

Two things need improving in the file sharing application presented so far. First, because all the method invocations were performed on omnihandles, every device received every file's meta-information and contents when requested by any device. In the case of meta-information this behavior is desirable, since every device will likely need to display the meta-information for every shared file. In the case of file contents this behavior is less desirable. When a device which did not request the file's contents executes the `reportFile` method in response to an omnihandle invocation, the device could nonetheless capture the file and save it for possible later use, or the device could simply do nothing. However, if not every device is going to need the contents of every shared file, it would be better not to send the file's contents to every device.

The second problem is that in the `reportFile` method defined above, the entire contents of the file was passed all at once. However, especially for a large file, the receiving device may not have enough buffer space to hold the entire file all at once. Also, if a communication failure occurs while the `reportFile` invocation is traveling through the network, the entire contents will have to be sent again, which wastes bandwidth.

To solve both problems, define two additional interfaces, one for the device sending a file and one for the device receiving it:

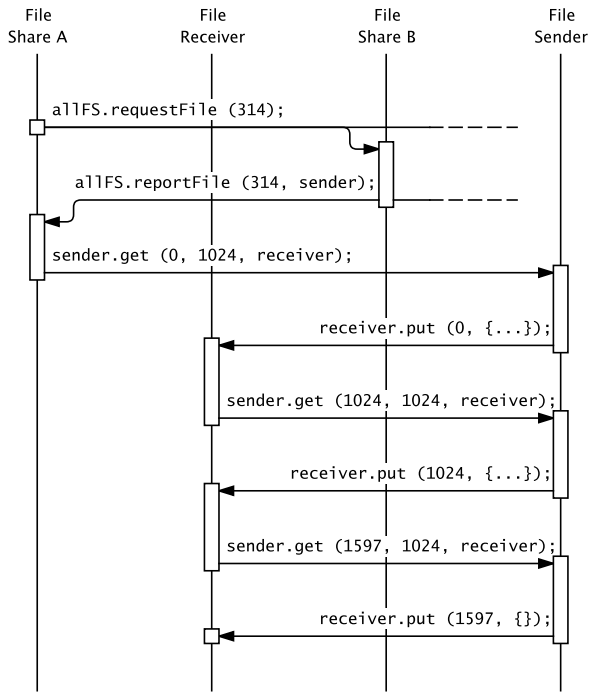


Figure 14: M2MI invocations for transferring a file

```
public interface FileSender {
   public void get (long offset,
      int count, FileReceiver receiver);
}
public interface FileReceiver {
   public void put (long offset,
      byte[] contents);
}
```

Also, change the interface of the `reportFile` method:

```
public interface FileShare {
   public void available (Hash[] ids);
   public void requestFile (Hash id);
   public void reportFile (Hash id,
      FileSender sender);
   public void requestName (Hash id);
   public void reportName (Hash id,
      String name);
}
```

Figure 14 shows the sequence of M2MI invocations that occurs when device A gets a file from device B using the revised interfaces. Device A starts by calling `requestFile` on the omnihandle, passing in the desired file ID. The device which possesses that file, B, executing `requestFile`, creates a `FileSender` object for that file, exports the object to the M2MI layer, and obtains a unihandle. Then B calls `reportFile` on the omnihandle, passing in the file ID and the file sender object's unihandle. A, executing `reportFile`, saves the file sender object's unihandle. Then A creates a `FileReceiver` object for the file, exports the object to the M2MI layer, and obtains a unihandle. Finally, A calls `get` on the file sender object's unihandle to get the first chunk of the file. The arguments to `get` are the offset of the first byte in the chunk, the number of bytes in the chunk (a size that A can conveniently handle), and the file receiver object's unihandle. B, executing `get`, calls `put` on the file receiver object's unihandle, passing in the starting offset of the chunk and the contents of the chunk. A, executing `put`, stores the chunk and calls `get` to obtain the next chunk. This sequence of alternating `get` and `put` calls continues until the entire file has been transferred. B signals the end of the file by calling `put` with a zero-length chunk.

To recover from communication failures, A starts a timeout after calling `get`. If the chunk does not arrive in a `put` call before the timeout, A retries the `get`; if a number of retries all fail, A gives up.

Once the entire file has been transferred (or an unrecoverable failure has happened), A can destroy the file receiver object. B can destroy the file sender object once a certain amount of time has elapsed with no `get` calls.

4.8 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, 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.

5 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 15 shows the overall architecture of M2MI. When the calling object invokes a target method on a handle, the invocation may have to be performed

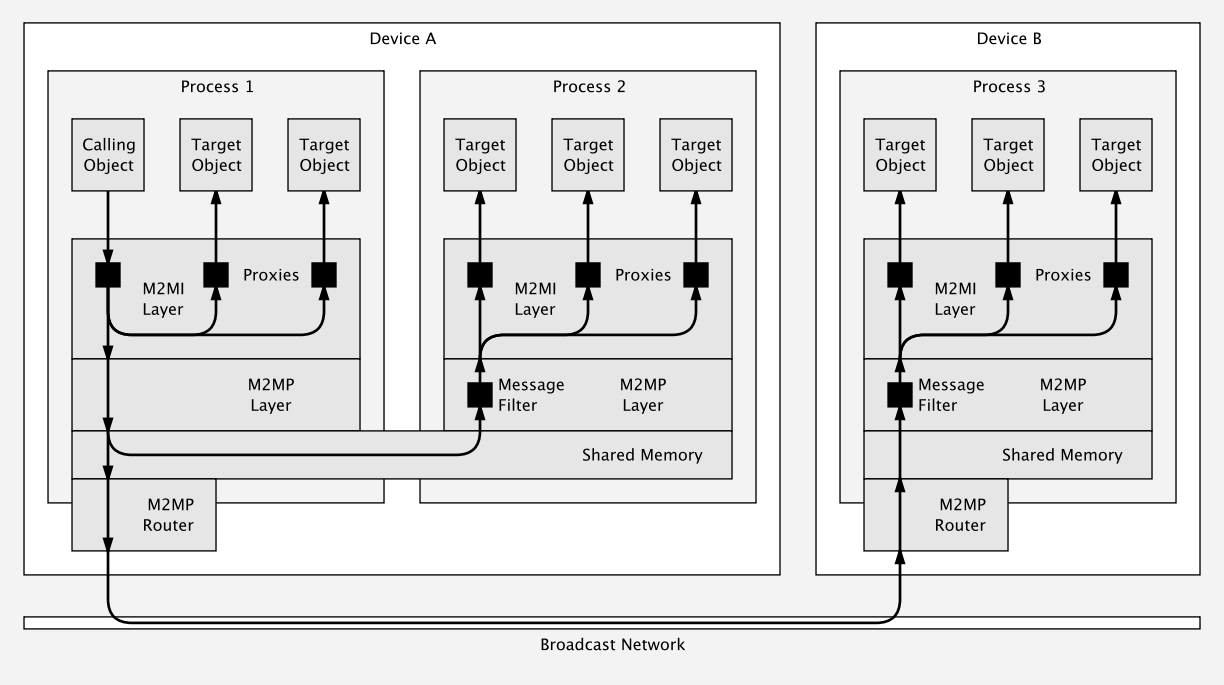


Figure 15: M2MI Architecture

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.

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.

Before reaching the M2MI layer, however, the invocation message must pass through a *message filter* in the M2MP layer. Only invocation messages destined for target objects exported in the M2MI layer in that process will pass through the message filter; messages destined for target objects in other processes will be discarded.

To reach target objects in other devices, an *M2MP router* in each device listens to the M2MP shared memory and transmits each outgoing message on the external broadcast network. The message is broadcast to all the devices in the proximal network. The M2MP router also listens to the external network and injects each incoming message into the M2MP shared memory, whence the message is processed in the same way as messages originating in the same device.

Thus an M2MI invocation is broadcast through the M2MI layer to all target objects in the same process; is broadcast through the shared memory to all target objects in other processes in the same device; and is broadcast through the external network to all target objects in other devices. The M2MP message filters weed out irrelevant messages, letting the M2MI layers devote processing resources only to the relevant messages.

In a device that does not have multiple processes, such as a small handheld device, the M2MI architecture is simpler. The shared memory and M2MP router are omitted. The M2MP layer sends outgoing messages directly to the external network and receives incoming messages directly from the external network.

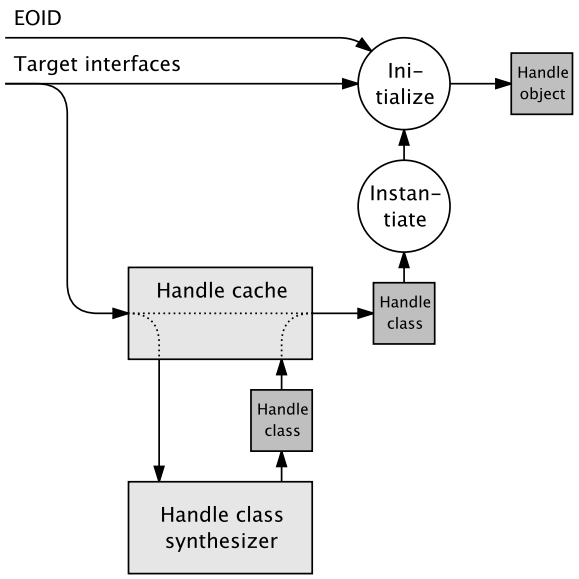


Figure 16: Creating a handle object

6 M2MI Design

This section describes the design of the M2MI layer at a high level. The M2MI API is described in the documentation that accompanies the M2MI software [25].

6.1 Handles

An M2MI handle object encapsulates two pieces of information: a list of the fully-qualified names of the handle's target interfaces (one or more), and a 128-bit *exported object identifier* (EOID). For an omnihandle, the EOID is a wildcard (zero). For a unihandle, the EOID is a globally unique nonzero value that designates a single exported object. For a multihandle, the EOID is a globally unique nonzero value that designates a particular set of exported objects.

A handle object's class implements all of the handle's target interfaces, providing an implementation for every method in every target interface and all superinterfaces thereof. A handle object can thus be cast to any of its target interfaces, and any method in any target interface can be invoked on a handle object.

When a handle object needs to be created (Figure 16), the M2MI layer first *synthesizes* the handle class. The M2MI layer uses Java reflection to identify all the target methods, creates a byte array containing a binary class file with implementations for all the target methods, loads the class file byte array into a special class loader, and gets back the handle class. To do this, the M2MI layer employs the RIT Classfile Library [27], a general purpose library for synthesizing Java class files. The M2MI layer then stores the handle class in a cache. The next time a handle is needed for the same target interfaces, the M2MI layer gets the handle class from the cache instead of synthesizing it again. Having obtained the handle class, the M2MI layer creates an instance of it and stores the proper target interface names and EOID in the newly created handle object.

An alternative to synthesizing handle classes would be to implement handles using Java reflection's dynamic proxies (class `java.lang.reflect.Proxy`). Measurements on several M2MI-based applications showed, however, that the applications ran 5 to 30 percent faster when the M2MI layer was implemented with synthesized classes than when the M2MI layer was implemented with dynamic proxies.

6.2 Exporting Objects

An object can be exported to the M2MI layer by calling `M2MI.export`, giving the object and the target interface or interfaces. In response, the M2MI layer adds the object to the *interface export map*, which maps the fully-qualified name of a target interface to a set of objects that have been exported as that target interface. For each target interface and each superinterface thereof, the object is added to the corresponding set in the interface export map. This lets the object be invoked by an omnihandle as described later.

An object can also be exported to the M2MI layer by calling `M2MI.getUnihandle`, giving the object and the target interface or interfaces. In response, the M2MI layer adds the object to the interface export map as before. The M2MI layer also adds the object to the *EOID export map*, which maps an EOID to a set of exported objects associated with that EOID. The M2MI layer generates a new EOID, adds the (EOID, object) mapping to the EOID export map, and returns a unihandle for the target interfaces initialized with that EOID. This lets the object be invoked by that unihandle as described later, as well as by an omnihandle. The M2MI layer will ensure that that EOID only ever maps to one object.

Finally, an object can be exported to the M2MI layer by first calling `M2MI.getMultihandle` to get a multihandle for a certain target interface or interfaces, then calling `attach` on the multihandle giving the object. To create a multihandle, the M2MI layer generates a new EOID and returns a multihandle for the target interfaces initialized with that EOID. When an object is attached to the multihandle, the

M2MI layer adds the object to the set of objects associated with the multihandle's EOID in the EOID export map, and the M2MI layer adds the object to the interface export map as before. This lets the object be invoked by that multihandle as described later, as well as by an omnihandle.

6.3 Performing an M2MI Invocation

An M2MI invocation starts when the calling object invokes a target method in a target interface on a handle. The target method uses Java object serialization to serialize the method's arguments, if any, into a byte array. The target method then passes the following information to the M2MI layer: the handle's EOID (initialized when the handle was created), the fully-qualified name of the target interface, the target method's name, the target method's descriptor, and the serialized arguments (Figure 17).

The M2MI layer needs to set up *method invoker* objects that will ultimately perform the invocations on the target objects. A method invoker is a `Runnable` object whose `run` method is customized to invoke a certain method in a certain interface. The M2MI layer *synthesizes* the appropriate method invoker class for the given target interface name, target method name, and target method descriptor. The M2MI layer saves the method invoker class in a cache to be retrieved the next time a method invoker is needed for the same target method and interface.

The M2MI layer next needs to find the target objects for the invocation that have been exported in the M2MI layer's process. For an omnihandle invocation, the M2MI layer uses the interface export map to map the target interface name to the set of target objects. For a unihandle or multihandle invocation, the M2MI layer uses the EOID export map to map the EOID to the set of target objects.

For each target object, the M2MI layer creates an instance of the method invoker class. The method invoker object is initialized with a reference to the target object and a reference to the byte array containing the serialized arguments. The method invoker object is then placed in a work queue for further processing on a separate thread.

Finally, the M2MI layer uses the M2MP layer to send an outgoing invocation message as detailed in Section 7.6. The M2MP layer is responsible for broadcasting the invocation message to other processes and/or devices. However, if the invocation was performed on a unihandle for an object exported in the M2MI layer's process, the M2MI layer does not send an outgoing invocation message (because there are no other target objects that need to be invoked).

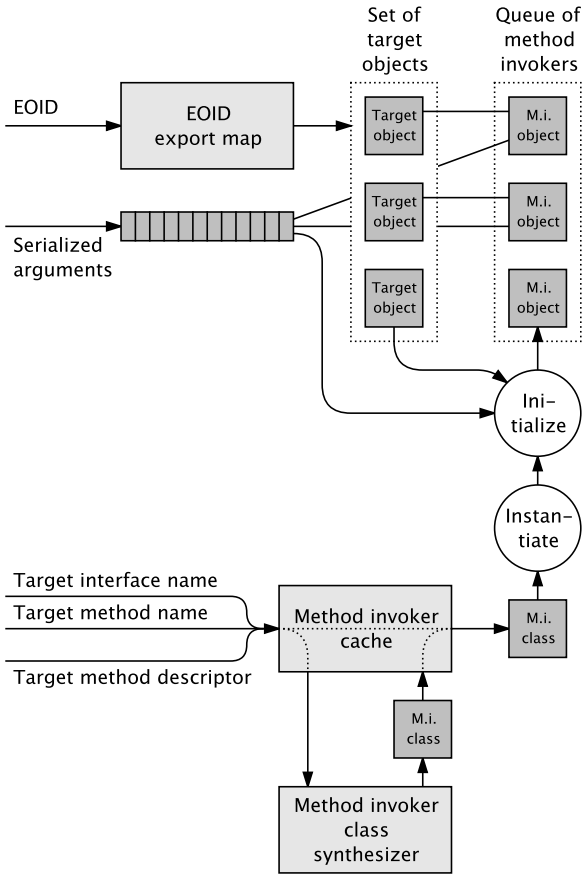


Figure 17: Performing an M2MI invocation, part 1

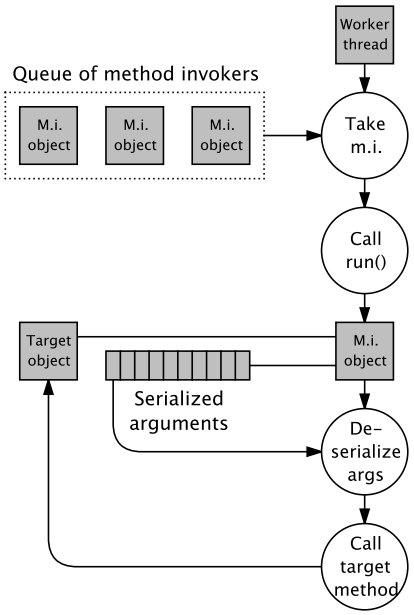


Figure 18: Performing an M2MI invocation, part 2

At this point the original call on the handle returns to the calling object.

The M2MI layer has a pool of one or more worker threads to process the work queue (Figure 18). (The number of worker threads needed depends on the application and is established when the M2MI layer is initialized.) Concurrently, a worker thread takes a method invoker object from the head of the work queue and calls the method invoker's `run` method. The `run` method deserializes the method arguments from the byte array with which the method invoker was initialized. The `run` method then invokes the target method on the target object with which the method invoker was initialized, passing in the deserialized arguments. Once the target method returns, the worker thread goes on to the next method invoker in the queue, blocking if necessary until one is added to the queue.

Since each method invoker separately deserializes the arguments from the byte array of serialized arguments, each target method gets its own copies of the arguments separate from the original calling object's arguments and separate from the other target objects' arguments. This enforces M2MI's argument pass-by-copy semantics.

6.4 Serialization of Handles

A handle object can be passed as an argument in an M2MI invocation just like any other object. However, handle objects must be treated specially during serialization and deserialization to ensure they work properly when reconstituted at the destination, which might in be a different process or device from the calling object. To do this, M2MI uses Java object serialization's *object replacement* capability [46].

Each handle class includes a `writeReplace` method. When a handle is serialized, the serialization system notices the `writeReplace` method and calls it. The `writeReplace` method returns a *handle transport object* initialized with the handle's EOID and target interface list. The serialization system then serializes this handle transport object rather than the original handle.

The handle transport class includes a `readResolve` method. When the handle transport object is deserialized at the destination, the serialization system notices the `readResolve` method and calls it. The `readResolve` method tells the destination's M2MI layer to create a handle for the target interfaces and EOID stored in the handle transport object. The M2MI layer creates the handle as usual, synthesizing the handle class if necessary. The `readResolve` method returns the handle, and the serialization system returns the handle as the result of the deserialization (instead of the handle transport object). Thus, the destination ends up with a handle for the same target interfaces and EOID as the original handle.

7 M2MP Design

This section describes the design of the M2MP layer at a high level. The M2MP API is described in the documentation that accompanies the M2MI software [25]. After describing the M2MP design, this section also describes how the M2MI layer uses the M2MP layer.

7.1 Assumptions

Intended particularly for the wireless proximal ad hoc networking environment, M2MP's design is based on these assumptions:

- *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.
- *A message's relevancy is determined by its contents.* A device decides which incoming messages to process by examining the initial bytes of each message.
- *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.

7.2 Outgoing Messages

When an application on one device sends an M2MP message, the application writes a stream of bytes with the message's contents to the M2MP layer. The M2MP layer breaks the byte stream into a sequence of fragments, wraps each fragment in a packet, and broadcasts each packet. An M2MP packet consists of these fields:

- Message ID (4 bytes)
- Fragment number and last packet flag (4 bytes)

- Message fragment (N bytes)
- Checksum (2 bytes)

The maximum length of an M2MP packet is 508 bytes. This number is chosen so an M2MP packet can be transmitted as a single datagram without fragmentation by various underlying transport protocols. Thus, the maximum length of each message fragment is 498 bytes.

To let the receiving devices reassemble packets sent simultaneously by many transmitting devices into the proper messages, every packet of an M2MP message carries the same value in the message ID field. Each device generates message IDs for successive messages by stepping through a random permutation of the set of 32-bit integers. Each device seeds its permutation generator with unique information including the device's system clock value and the device's physical layer address (such as an Ethernet MAC address or a Bluetooth device address). Thus, each device generates a different permutation of the integers, and there is a negligible probability that two devices will generate the same message ID at the same time. In this way, packets from different messages can be distinguished without the devices having to coordinate with each other.

The fragments of a message are numbered starting with 0, and the fragment number field identifies which fragment the packet contains. The last packet flag is 0 in all packets except the last, where it is 1.

The message fragment field holds the message fragment itself. Each message fragment except possibly the last is 498 bytes long. The length of the message fragment, N, is not carried within the packet. Instead, the overall packet length is obtained from the next lower protocol layer used to transport the packet, and this determines N.

Finally, the checksum field contains a simple 16-bit ones complement sum of the rest of the packet and is used to detect alteration of the packet during transit. (This checksum is unable to detect certain kinds of attack and must be strengthened as discussed in Section 9.1.)

7.3 Incoming Messages

To receive incoming messages, an application must register one or more *message filters* with the M2MP layer. Each message filter has a *message prefix,* a fixed byte string. If an incoming message's initial bytes match the message prefix of a registered message filter, the M2MP layer passes the message up to the application that registered the message filter. Otherwise, the M2MP layer discards the message, and the application never sees it. An application that uses M2MP, such as M2MI, designs the contents of its M2MP messages to take advantage of M2MP's message filtering capability and weed out irrelevant messages before they ever reach the application.

The M2MP layer processes each incoming packet as follows. If the checksum is not correct, the packet is discarded as corrupt. If it is the first packet of a message (fragment number is 0), the M2MP layer compares the message fragment to all the registered message filters' message prefixes using an efficient trie search. If there is no match, the packet is discarded as irrelevant. But if there is a match, the M2MP layer creates a new incoming message associated with the packet's message ID and forwards the message to the application that registered the matching message filter. The application reads a stream of bytes containing the message's contents, beginning with the message fragment in the first packet. If there are further packets in the message (last packet flag is 0), the M2MP layer starts a timeout to wait for the next packet.

If an incoming packet is not the first packet of a message (fragment number is greater than 0), the M2MP layer looks for an in-progress message associated with the packet's message ID. If there is none, the packet is discarded as irrelevant. If there is a message in progress, but the packet's fragment number is not the next expected fragment number, the packet is discarded as out of sequence. Otherwise, the M2MP layer cancels the timeout and feeds the packet's message fragment to the application reading the message. If there are further packets in the message, the M2MP layer restarts the timeout to wait for the next packet.

If a failure occurs in the middle of a message, such as a lost packet or a corrupted packet, the M2MP layer will time out waiting for the packet with the expected next fragment number to arrive. If the timeout occurs, the M2MP layer abandons the message and signals an exception to the application reading the message. The M2MP layer neither acknowledges nor retransmits packets.

Retransmitting lost packets is unnecessary, and abandoning the message is acceptable, because we assume the proximal 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 [8]. This considerably simplifies M2MP.

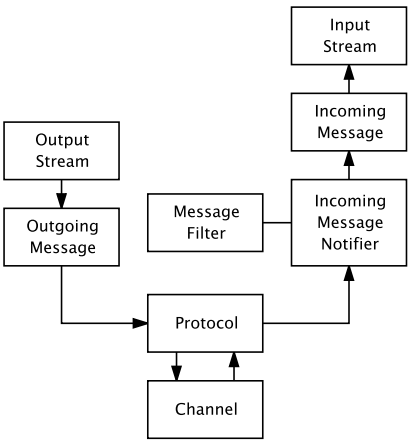


Figure 19: Objects in the M2MP API

7.4 M2MP API

Figure 19 shows the principal objects in the M2MP API 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 underlying layer. By plugging a different channel object into the protocol core, M2MP can be used with different underlying protocols. 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 one of the message filters. For each incoming message, the application obtains an input stream and reads the message's contents from the input stream.

7.5 Underlying Layers

The layer shown in Figure 15 underneath the M2MP layer is presently implemented by a channel object that uses the Internet protocol stack in lieu of shared memory (Figure 20). The channel wraps each outgoing M2MP packet in a UDP datagram and sends the datagram to a designated well-known port on the local host IP address, 127.0.0.1. Concurrently, the channel reads UDP datagrams containing incoming M2MP packets from its own separate port on the local host IP address. A separate M2MP router process receives datagrams from the well-known port

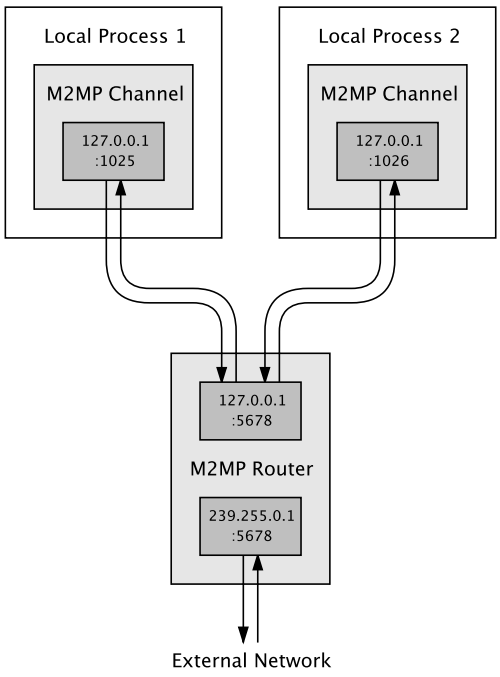


Figure 20: M2MP channels and M2MP router

and sends copies of them to all the channels' ports. Although it is a roundabout way of achieving interprocess broadcast, this scheme can be implemented in pure Java without needing nonportable native code libraries or operating system kernel modifications.

Additionally, the M2MP router process sends a copy of each datagram from a local process to an external network address, and concurrently receives incoming datagrams from an external network address and copies them to the local processes. The external network address can be a unicast IP address, in which case M2MP messages are tunneled between just two devices. Alternatively, the external network address can be a multicast IP address, in which case M2MP messages are broadcast to all devices that have joined the multicast group.

Ideally, M2MP would be supported directly by the operating system with its own protocol stack, including a true shared memory layer, and would not have to incur the additional overhead of the Internet protocol stack. Adding M2MP support to the operating system kernel is an area of future work.

7.6 M2MI's Use of M2MP

Having described the M2MP layer's design, we can now describe how the M2MI layer uses the M2MP layer.

When a calling object calls a target method on a

handle, the M2MI layer sends an outgoing invocation as an M2MP message, and the M2MP layer broadcasts this message to all processes and devices. The invocation message contains these items:

- Magic number, `"M2MI"` in ASCII (4 bytes)
- Hash code of the key used to find the target objects in the interface export map or EOID export map (4 bytes)
- Target interface name (UTF-8 string)
- Target method name (UTF-8 string)
- Target method descriptor (UTF-8 string)
- Length of the serialized arguments (4 bytes)
- The serialized arguments themselves, if any

In an M2MI invocation message, the message prefix used for message filtering consists of the first 8 bytes: the magic number and the key's hash code. Whenever a target object is exported — that is, whenever a target object is associated with a certain key in either the interface export map or the EOID export map — the M2MI layer registers a message filter with the corresponding message prefix. Likewise, whenever a target object is unexported, the M2MI layer deregisters the message filter with the corresponding message prefix. The M2MP layer's trie data structure allows the message prefixes to be stored and searched efficiently even if many objects are exported.

When an incoming M2MP message containing an M2MI invocation arrives at the M2MP layer, the M2MP layer compares the message's initial bytes to the registered message prefixes. If the magic number doesn't match, the message was not generated by an application using M2MI, and the message can be discarded. If the key's hash code doesn't match, then the invocation is not destined for any target object exported in this process, and the message can be discarded.

If both the magic number and the key's hash code match, the M2MP layer passes the invocation message on up to the M2MI layer. The M2MI layer skips over the message prefix (which is there only for efficient message filtering in the M2MP layer) and extracts the target interface name, target method name, target method descriptor, and serialized arguments. The M2MI layer then proceeds to process the invocation in exactly the same way as an invocation originating within its own process.

8 Related Work

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

8.1 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 [21], 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 [39, 23, 40, 18, 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 [2, 35, 48, 31, 50]. 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 [37, 38, 9]. 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 de-

livery, 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* [35] 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.

8.2 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 [36] and Java RMI [42]. 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 [32].

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 [44]. 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, 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 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.

8.3 Distributed Systems Architecture

Figure 21 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.

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

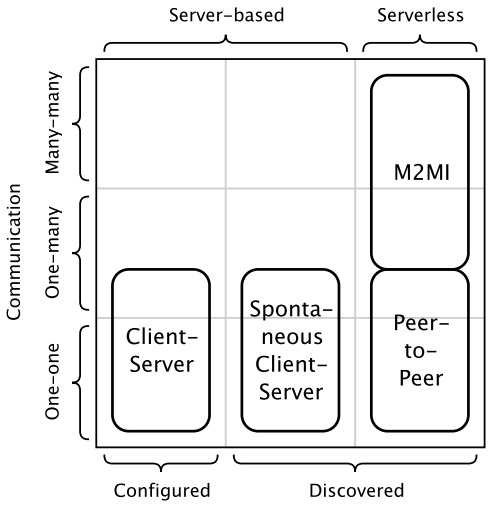


Figure 21: Design centers of distributed systems architectures

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 [1] 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.

8.4 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 [29, 30] and JXTA [24], 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 [16], 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 [33] 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.

9 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.

9.1 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 [26]. The prototype includes several channel implementations that use UDP datagrams to transport M2MP packets (see Section 7.5).

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 overhead of the UDP and IP layers in the prototype [20]. 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.

The way the M2MP packet format is presently defined, an adversary could disrupt a multi-packet M2MP message by injecting a packet with the correct next fragment number but a bogus message fragment. The M2MP layer would have no way of knowing that this packet is not authentic and would pass the bogus data up to the application. This attack is of especial concern in a wireless network, which is arguably easier for an intruder to access than a wired network.

To defend against a packet insertion attack, we plan to replace the checksum with a *message authentication code* (MAC), which is a one-way hash of the packet's contents that requires a key to compute or verify. Each packet uses a different randomly-chosen key. The key needed to verify a packet's MAC is carried in the *next* packet; a final empty packet carries the last key. To conduct a packet insertion attack, an adversary would have to determine the key from a packet's contents and MAC, so as to put the correct key in the next packet; but this is computationally infeasible. An initial version of this scheme has been implemented [4].

9.2 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 [25]. The M2MI prototype uses the M2MP prototype [26] for messaging and the RIT Classfile Library [27] for dynamic proxy synthesis. It builds on an earlier prototype that used an offline proxy compiler [5].

In our continuing work on M2MI, we plan to port the RCL, M2MP protocol core, M2MP channel, and M2MI implementations to a PDA platform with Java capability and 802.11 or Bluetooth wireless connectivity. The porting effort may require redesigning and reimplementing the software to reduce the memory and CPU requirements to a level suitable for a small mobile wireless device. Once ported, we plan to test interoperation of M2MP and M2MI from PDA to desktop host and from PDA to PDA.

We also plan to develop tools to help develop and debug M2MI-based systems and to monitor and visualize M2MI-based systems during operation.

9.3 M2MI-Based Systems

Initial prototypes of several collaborative applications, including chat, IM, whiteboard, calendar, file sharing, and tuple space, have been constructed using M2MI. The prototypes run on desktop hosts.

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 Sections 4.2 and 4.4), service discovery (see Section 4.5), multiple simultaneous groups (see Section 4.2), random selection of respondents to avoid broadcast storms (see Sections 4.3 and 4.7), and timeouts to recover from missing responses.

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 [33]

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.

9.4 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 [28] 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 [10], 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 [11, 17, 41, 45] 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 [3], 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 [6].

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. Where the existing algorithms are not well

suited for that environment, we will define modified cryptographic algorithms that are better suited. To reduce memory and processing requirements in small devices, we will define elliptic curve based variants of the cryptographic algorithms where necessary. Finally, we will analyze how to extend the M2MI infrastructure to provide confidentiality and authentication.

10 Acknowledgments

Jim Waldo inspired the idea for M2MI when he said "Everyone that's out there, call this method" during a discussion about M2MP. Wendi Heinzelman suggested the digital photo sharing application in Section 4.7.

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, and the anonymous referees for their comments on earlier drafts of this paper.

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

References

[1] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[2] 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.

[3] J. Benaloh and M. 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.

[4] J. Binder, J. King, K. Mooney, and M. Wilkinson. Ad hoc wireless networks security system summary. Research seminar class project report, Rochester Institute of Technology, Rochester, NY, May 2002.

[5] H.-P. Bischof. Many-to-Many Invocation Compiler. `http://www.cs.rit.edu/~anhinga/downloads/historical.shtml`.

[6] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[7] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[8] D. R. Cheriton and D. 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.

[9] D. M. Chiu, M. Kadansky, J. Provino, J. Wesley, H.-P. Bischof, and H. 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`.

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

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

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

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

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

[15] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[16] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. 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`.

[17] 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.

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

[19] T. Hastings, R. Herriot, R. deBry, S. Isaacson, and P. Powell. Internet Printing Protocol/1.1: Model and Semantics. Internet Request for Comments (RFC) 2911, September 2000.

[20] K. Hegde. M2MP over Ethernet. Master's thesis, Rochester Institute of Technology, Rochester, NY, 2002. In progress.

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

[22] Jini Printing Working Group, A. Kaminsky, editor. Jini Print Service API Draft Standard 1.0. `http://print.jini.org/`, May 2000.

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

[24] Project JXTA. `http://www.jxta.org/`.

[25] A. Kaminsky. Many-to-Many Invocation Library. `http://www.cs.rit.edu/~anhinga/m2mi.shtml`.

[26] A. Kaminsky. Many-to-Many Protocol Library. `http://www.cs.rit.edu/~anhinga/m2mp.shtml`.

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

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

[29] 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`.

[30] 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`.

[31] 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.

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

[33] 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.

[34] National Institute of Standards and Technology. Digital signature standard. NIST FIPS PUB 186, May 1994.

[35] 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.

[36] Object Management Group. The common object request broker: Architecture and specification, revision 2.4.1, November 2000.

[37] E. Pagani and G. P. 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.

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

[39] C. E. Perkins and P. Bhagwat. DSDV routing over a multihop wireless network of mobile computers. In T. Imielinski and H. F. Korth, editors, *Mobile Computing*, pages 183–206. Kluwer Academic Publishers, 1996.

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

[41] 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.

[42] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, Fall 1996.

[43] R. Rivest. The MD5 message-digest algorithm. Internet Request for Comments (RFC) 1321, April 1992.

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

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

[46] Sun Microsystems. Java Object Serialization Specification. `http://java.sun.com/j2se/1.4/docs/guide/serialization/index.html`.

[47] Sun Microsystems. Java Print Service Specification. `http://java.sun.com/j2se/1.4/docs/guide/jps/index.html`.

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

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

[50] 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`.

A Chat Application Source Code

Figure 22 gives the Java source code for the M2MI-based chat application of Section 4.1 (the one with omnihandles and a single chat session).

Class `ChatDemo` is the main program. After initializing the M2MI layer,[3] the main program creates a chat UI object and a chat object. These objects do all the work.

The chat UI object, class `ChatFrame`, provides a simple graphical user interface to display the messages in the chat log and to let the user enter and send a new chat message. The chat UI object provides an operation to register a chat frame listener object (interface `ChatFrameListener`). The source code for class `ChatFrame` is omitted since it has nothing to do with M2MI.

Interface `ChatFrameListener` specifies the interface for a chat frame listener object. Whenever the user sends a new chat message, the chat UI object calls the `send` method on its registered chat frame listeners, passing in the chat message text.

Interface `Chat` is the target interface for M2MI invocations on the exported chat objects.

Class `ChatObject` is the exported chat object, which implements interfaces `ChatFrameListener` and `Chat`. When constructed, the chat object is given the chat UI object and the user name. The chat object registers itself with the chat UI object as a chat frame listener. The chat object also exports itself to the M2MI layer as target interface `Chat`. Finally, the chat object obtains an omnihandle for interface `Chat` from the M2MI layer.

When the user sends a new chat message, the chat UI object calls the chat object's — that is, the chat frame listener's — `send` method. (This is a normal method call, not an M2MI invocation.) The chat object prepends the user name to the message text and calls `putMessage` on the omnihandle for interface `Chat`.

The omnihandle invocation causes every chat object's `putMessage` method to be executed. Each chat object calls a method in its corresponding chat UI object to add the chat message to the chat log. (This is a normal method call, not an M2MI invocation.)

[3]The argument is a globally unique address for the M2MI layer that would typically be the host's network interface's MAC address.

```
public class ChatDemo
   {
   public static void main
      (String[] args)
      {
      M2MI.initialize (1234L);
      ChatFrame theChatFrame = new ChatFrame();
      ChatObject theChatObject =
         new ChatObject (theChatFrame, args[0]);
      }
   }

public interface ChatFrameListener
   {
   public void send
      (String line);
   }

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

public class ChatObject
   implements ChatFrameListener, Chat
   {
   private ChatFrame myChatFrame;
   private String myUserName;
   private Chat allChats;

   public ChatObject
      (ChatFrame theChatFrame,
       String theUserName)
      {
      myChatFrame = theChatFrame;
      myUserName = theUserName;
      myChatFrame.addListener (this);
      M2MI.export (this, Chat.class);
      allChats = (Chat) M2MI.getOmnihandle
         (Chat.class);
      }

   public void send
      (String line)
      {
      allChats.putMessage
         (myUserName + "> " + line);
      }

   public void putMessage
      (String line)
      {
      myChatFrame.addLineToLog (line);
      }
   }
```

Figure 22: Chat application source code