2006

Group data communication with M2MI

Kay Cheng

Rochester Institute of Technology

TECHNICAL REPORT

OF

GROUP DATA COMMUNICATION OF M2MI VIA LAN

-- M2MI-Based Collaborative Groupware

By Kai Cheng
kxc8217@cs.rit.edu

Version 6.0

Committee:

Chairman: Prof. Hans-Peter Bischof
Reader: Prof. Fereydoun Kazemian
Observer: Prof. Paul T. Tymann

Table of Contents

1. Abstract of Project Objectives

This graduate project will provide three different design patterns of M2MI-based collaborative systems, implement and simulate those designs in LAN environment, and compare the advantages and disadvantages of the M2MI-based solutions with RMI-based solutions of those three different problems, collaborative groupware, multiple participants chat system, and the distributed solution of shared resource allocation.

Groupware Description

This chapter descripts the design patterns of three different M2MI-based collaborative systems.

Multiple Participants Chat System

It has multiple participants chatting feature, for example: instant chatting between two users, chatting within a group, and recovering lost chat messages, etc.

This system has the following features:

- All the users have the ability to know which users are online and available to receive message and message log recovery request
- Instant message – instant message exchanging between two users
- Group chat – messages exchanging within multiple users chat session
- Chat log recovery – recovering lost chat messages

Design Pattern

It is designed for instant or group chat for multiple participants (synchronous, namely, multiple users may send out messages at the same time) that are nearby (like in a conference room). It applies to situations in which small computing devices, which run in ad hoc network, need to perform the following operations: user identification, instant data sending, instant data receiving, group data exchanging, and data log recovery, etc. My project will design this pattern and simulate it in LAN environment.

The core of this design pattern is there are two phases for this design.

First, User Recovery -- every user has to use an omnihandle to recover other users' unihandles for later use.

Second, Function Implementation -- when a user wants to send data to another user, it will call the appropriate method on the receiver's unihandle, which the sender gets at the first phase. The receiver will invoke the method on its own unihandle object and get the data. For **n** users chat session, repeat this for **n - 1** times.
Each user keeps the most recent message ID **i** and the sender and receivers of this message. He will check the incoming message ID **j**. If **j** >= **i** with the same participants, then requests the sender to recovery the lost data from same participants.

Collaborative Groupware

In my project, I use the *Appointment Making Application* as the example. It has calendar-scheduling feature, for example: appointment request, acceptance, rejection, modification, cancellation, and display, etc.

This system has the following features:

- All the users have the ability to know which users are online and available to receive an appointment request
- Create an appointment with different user and contents.
- Send an appointment the another instance of the program
- Accept or reject an appointment with reply specified by user
- Display all the valid appointments
- Cancel an existing valid appointment
- Update an existing appointment with another user by cancel an valid appointment and create an new one

Design Pattern

It is designed for appointment proposal, acceptance, rejection, and cancellation for multiple participants (synchronous, namely, multiple users send out appointments at the same time) that are nearby (like in a conference room). It applies to the situations in which small computing devices, running in ad hoc network, need to perform the following operations: event creation, event message sending, event message receiving, event confirmation, event rejection, event updating, and event display, etc. My project will design this pattern and simulate it in LAN environment

The core of this design pattern is there are two phases for this design.

First, User Recovery -- every user has to use an omnihandle to recover other users' unihandles for later use.

Second, Function Implementation -- when a user wants to send appointment to another user, it will call the appropriate method on the receiver's unihandle, which the sender gets at the first phase. The receiver will invoke the method on its own unihandle object and get the appointment. Then depends on the receiver's action, the receiver will send back accept or reject reply. Then both sender and receiver save the appointment and display it or discard the appointment up to the receiver's action. All the participants in an appointment can send out the cancel requests, after receivers confirm it, this existing appointment will be removed from every participants' active appointment lists. For **n** users chat session, repeat this for **n - 1** times.

Each user keeps the most recent appointment ID **i** and the participants. He will check the incoming appointment ID **j**. If **j** >= **i** with the same participants, then requests the sender to recovery the lost data of the same participants.

Distributed Solution of Shared Resource Allocation

In my project, I use *Distributed Solution of Dinning Philosopher* as the example. This application is designed for the distributed solution of the Dining Philosophers problem when using in ad hoc environment. It provides the distributed solution for shared resource allocation in network. The main objective of this solution is to find a reasonable design to allocate shared resource to separate users (running on different computing device, which means has different Java Virtual machine), make sure they have equal opportunity to use shared resource, and avoid deadlock.

Design Pattern

It is designed for solving the dining philosophers, which is a classical example to illustrate various problems (like deadlock and equal opportunity for all users) that can occur when many synchronized threads are competing for limited shared resources.
It applies to situations in which fined number **k** small computing devices, running in ad hoc network, need to compete for some shared resource. Every device should have equal opportunity to use the shared resource and avoid deadlock. My project will design this pattern and simulate it in LAN environment.

The core of this design pattern is there are two phases for this design.

First, User Recovery -- every philosopher has to use an omnihandle to recover all the chopsticks' unihandles for later use.
Second, Function Implementation – when a philosopher wants to eat, he always calls the **use()** method on his lower number chopstick's unihandle, which the philosopher gets at the first phase. Only if he gets his lower number chopstick then he can continue to call the **use()** method on his higher number chopstick otherwise he will hold the lower number chopstick and keep trying to request the higher number chopstick until he gets both of them and eat. When he is done eating, he will release all the chopsticks he holds. Chopstick is designed with exclusive access that means only one philosopher can use it at the same time and chopstick will block all the others philosopher till the first one is done eating. It will send back the cancel message to the blocked philosopher to notify them that they are blocked.

Research Objective Concepts

This project has the following research concepts:

- Investigate the design pattern and model design of collaborative groupware
- Investigate the JAVA design and implementation of the collaborative groupware
- Investigate M2MI mechanism using in the three different problems in ad hoc environment.
- Investigate the architecture, mechanism and performance of the designs of the three problems and compare them with RMI based solution. Test will be performed while using varieties of M2MP packets.

2. Architecture Overview

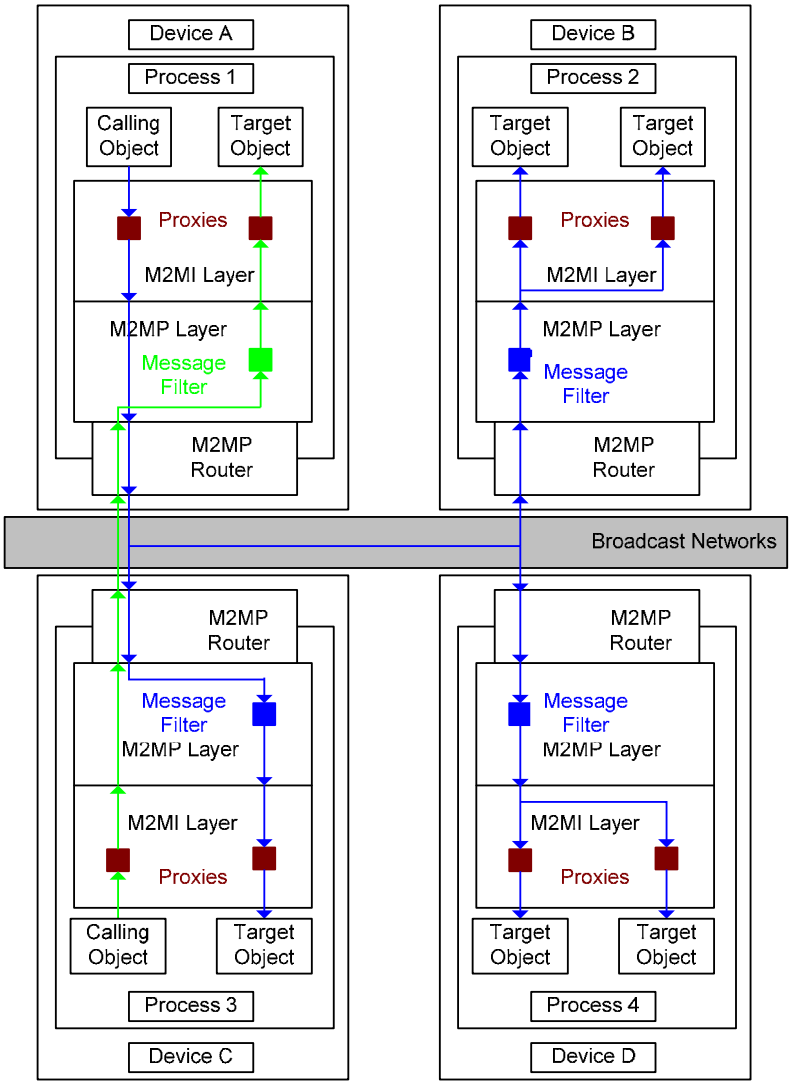


Figure 1: "Collaborative Groupware System Architecture"

Figure 1 shows the architecture of M2MI-based collaborative system. It is suitable for small computing device running in wireless ad hoc networks environment. When a method is invoked on an object's omnihandle for a particular interface, all the target object, which implements the same interface, will perform the same method. If a method is invoked is on a unihandle, only the object bounded to this unihandle performs the method. If a method is invoked on a multihandle, the set of objects that implement the same interface and attach to that multihandle will perform the method.

3. Functionality Specification

This chapter descripts the functionality specifications and design details for the three different problems of M2MI-based collaborative systems: multiple participants chat system, appointment making groupware system, and the distributed solution for dinning philosophers.

3.1 Multiple Participants Chat System

This chapter descripts the background and problem, design pattern based on the M2MI mechanism, and the design specification of the multiple participants chat system.

3.1.1 Background and Problem

This ***Multiple Participants Chat Application System*** is designed for instant or group chatting for multiple participants (synchronous, namely, multiple users send out messages at the same time) that are nearby (like in a conference room).

The chat system can perform the following functions and may encounter the following problems.

Instant Chat

A user has the ability to type in texts, and those texts can be displayed on the user's device that the sender specifies. In this section, we only deal with one-to-one chat. The one-to-many chat (multiple chat sessions) will be treated in Group Chat section.

To achieve that, we may have the following problems:

- Every user must have a local active user list so it can look up this list to find the user it wants to send message to
- The messages are only displayed on the two users' devices involved in the instant message chat session. This can avoid unnecessary data transmission and broadcast storm
- The messages are displayed on the users' devices involved in the chat session in the order that they are typed in
- Every active user should keep notifying all the other active users if it is still available to receive instant message

We will discuss the solution in the design specification section.

Group Chat

A user has the ability to create a multiple participants chat session and the users involved can add themselves respectively in the group chat session. Whenever a user in a chat group types in texts, those texts can be displayed on all the users' devices in that particular chat group.

To achieve that, we may have the following problems:

- Every user must have a local active user list so it can look up this list to find the users that it wants to invite to create a group chat session
- Every user can create a multiple participants chat session and send it to all the users involved
- A user who joins later has the ability to add himself in it
- All the users in a group chat session can type in texts and the texts can be displayed on all the users' devices in that group chat session
- The messages are only displayed on the devices involved in the group chat session. This can avoid unnecessary data transmission and broadcast storm
- The messages are displayed on the users' devices involved in the chat session in the order that they are typed in
- Every active user should keep notifying all the other active users if it is still available to receive group messages

We will discuss the solution in the design specification section.

Chat Log Recovery

When some users are out of the service area (since this collaborative chat system is designed based on M2MI mechanism and compatible with wireless proximal ad hoc networks), or messages are lost during broadcasting (since the M2MI using UDP), all users' devices should have the ability: detecting messages lost and recovering the lost messages by synchronizing with other users' devices.

To achieve that, we may have the following problems:

- Every user can detect if it has all the messages that it should have received
- When a user finds it does not have all the messages, it can synchronize with other devices to get the lost messages recovered
- To avoid unnecessary data transmission and broadcast storm, users exchange chat logs as less as possible

We will discuss the solution in the design specification section

3.1.2 Design Pattern based on M2MI mechanism

This chapter descripts the design pattern of *M2MI-based Multiple Participants Chat System.*

“Patterns are devices that allow programs to share knowledge about their design.”[7] Generally, same problem may occur again and again. So developer has to repeat solving almost the same problem many times. Pattern is a better way that we can make the design that we have developed to solve a specific program design problem reusable.

Pattern Name

M2MI Instant Message-exchange **(a documenting design pattern** [7]**)**

“Documenting patterns is one way that developer can reuse and possibly share the information that users have learned about how it is best to solve a specific program design problem. Documenting design patterns are usually done in a fairly well defined form. The general form for documenting patterns is to define items such as:

1. The motivation or context that this pattern applies to
2. Prerequisites that should be satisfied before deciding to use a pattern
3. A description of the program structure that the pattern will define
4. A list of the participants needed to complete a pattern
5. Consequences of using the pattern...both positive and negative
6. Examples”

Intent and Motivation

The ***M2MI Instant Message-exchange*** pattern applies to situations in which small computing devices, which run in ad hoc network, need to perform the following operations: user identification, instant message sending, instant message receiving, group chatting, and chat log recovery, etc. My project will design this pattern and simulate it in LAN environment.

Applicability (or Prerequisites)

The ***M2MI Instant Message-exchange*** pattern is used in classical ad hoc environment for multiple participants chatting, which means no central server, no network administration, no complicated routing protocol and no complicated system development. It is suitable for small set of users, small computing devices and limited working space (Noisy or Quite space)

Solution

The following class will be developed when implement the ***M2MI Instant Message-exchange*** pattern.

Each ***M2MI Instant Message-exchange*** application exports an object implementing these following interfaces for user identification and leaving notification:

```
public interface UserReporter {
        // developer can add more arguments and more method
        public void declare (Chat myChat, String userName);
        public void announce (Chat theChat, String theUserName);
        public void leave (String theUserName);
        public void goodby (String theUN);
        …}
```

Interface that is designed for instant message sending, group chatting, get the latest message ID and get the message with specified message ID.

```
public interface Chat {
        // developer can add more arguments and more method
        public void putMsg (int theID, String theUN, Chat sender, String msg, String
        theParticipants);
        public void selfPutMsg (String msg);
        public void request(String thePar, Chat requester, int startLost);
        public void getLostMsg (String theLostMsg, Vector theLostM);
        …}
```

Participants

- Initial class, which declares a new user joining in and leaving. The ***Instant Message-exchange*** application obtains an omnihandle from M2MI layer for this initial interface
- Main class, which implements instant message sending, group message sending, lost messages request and recovery, and latest message ID request and recovery methods, etc. The ***Instant Message-exchange*** application obtains an unihandle from M2MI layer for this interface

Code Example

See the Design Specification section.

3.1.3 Design Specification

This chapter descripts the design details; include the exported interfaces, the member methods, and the algorisms, of the *Multiple Participants Chat System*.

Active User Discovery

Each chat application exports an object implementing these three following interfaces:

The following interface is designed for every user declares itself when it comes into the service. Then all the devices run this method on the omnihandle ***UserReport*** and get the unihandle **Chat c** and **c**'s **username** and keep it in their local user list.

```
public interface UserReporter {
    // developer can add more arguments and methods
    public void declare (Chat myChat, String userName);
    public void announce (Chat theChat, String theUserName);
    public void leave (String theUserName);
    public void goodby (String theUN);
    }
```

This interface is designed for chatter's message sending, lost message requesting, lost message recovery, the latest message ID requesting, and the latest message ID recovery.

```
public interface Chat {
    // developer can add more arguments and methods
    public void putMsg (int theID, String theUN, Chat sender, String msg, String
    theParticipants);
    public void selfPutMsg (String msg);
    public void request(String thePar, Chat requester, int startLost);
    public void getLostMsg (String theLostMsg, Vector theLostM);
    }
```

This interface specifies the interface for a chat object is triggered when the user sends a line of text in its graphic user interface.

```
public interface ChatFrameListener {
    // developer can add more arguments and methods
    public void send(String line);
    }
```

Firgure 2 shows how new user declares itself. When the service starts, the existing user A exports a *ChatObject* object that implements the interface *Chat* and *ChatFrameListener* to M2MI layer and gets a unihandle **Chat_a** from M2MI layer for the *ChatObject*. Since **Chat_a** is a unihandles, only the destined device executes the methods on **Chat_a**. And then it exports a *UserReporterObject* Object that implements the interface *UserReporter*. Then the application gets an omnihandle **Reporter_a** from M2MI layer for interface *UserReporter* and passes in its own unihandle ***ChatObject*** **Chat_a** and its own ***username a*** as parameters. Finally, the user executes the **declare()** method every 3 seconds, and then the other existing user run the **declare ()** method on the same *UserReporterObject*

omnihandle **Reporter_a** and save the **username a** and unihandle **Chat Chat_a**. Every new user will do the same thing so an active user list has been generated at every active user's local device. Note here, since every existing user keep declaring itself every 3 seconds, any later join-in user will get all the active existing user even if they does not appear at the beginning. And also, it is a good way to track user's availability. (If user A does not receive an active user's declaration for a while, A will treat it as inactive and remove it from A's active user list)

For Example, the omnihandle **Reporter_a** is created with code:

```
M2MI.export (this, UserReporter.class) ;
UserReporterObject Reporter_a = (UserReporter) M2MI.getOmnihandle
(UserReporter.class);
```

The unihandle **Chat_a** is created with code:

```
M2MI.export (this, Chat.class) ;
Chat Chat_a = (Chat) M2MI.getUnihandle(this, Chat.class) ;
```

Passing in **Chat_a** and **A's username a** as parameters for the declare method on the omnihandle **UserReporterObejct**:

```
UserReporterObject theReporterObject = new UserReporterObject (Chat_a, a) ;
theReporterObject.announce(Chat_a, a) ;
```

Inactive User Removal

Figure 2 shows how an active user leaves and all the other users remove it from their active user list. When User B wants to leave, a **clearGone()** thread at B calls **leave(b's userName)** method on the omnihandle **theReporterObject** and so all the other user will know B's leaving and remove B from their active user list.

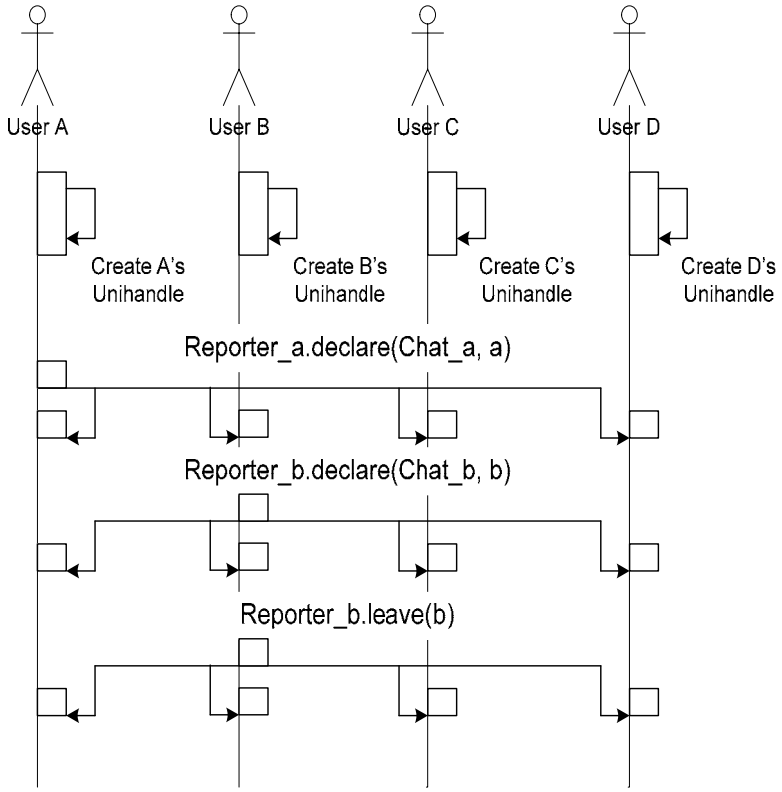


Figure 2 User Join in and leave for 4 users Chat system

Instant Chat

When user A wants to chat with another user B, it looks up the corresponding *Chat* unihandle in the active user list and calls the **putMsg ()** method on the receiver's *Chat* object's unihandle, which is chat_b, passing in data (**msgID, a**, **Chat_a**, **message, "a b"**). Only the destined receipt, the user B, rather than the whole group executes the **putMsg ()** method and then displays the chat message. All the other users in service that are not involved in this chat session do not execute any method about this chat at all so they can not see any chat message from the conversation between A and B.

Figure 3 shows a sequence of M2MI invocations that instant chat for 4 devices. User A wants to chat with user C. First, the application running on device A looks up device C's **Chat** unihandle, which is **chat_c**. Second, device A calls **putMsg ()** method on C's Chat unihandle **chat_c** and passes in the **msgID**, A's username **a**, A's **Chat** unihandle **chat_a,** the message **msg,** and the participants **"a c"**. (we can just simple using int value, when each chat message with c is generated, **msgID** increases by 1) Only C executes **putMsg ()** method, and display the message. When C wants to reply messages to A, C will do the exactly same thing but call **putMsg ()** method on A's Chat unihandle chat_a.

If a user wants to have multiple instant chat session, just select the user it wants to chat and chat. The Chat application will act like a new instant chat session and the messages are still only shown on the two users evolved in that instant chat session.

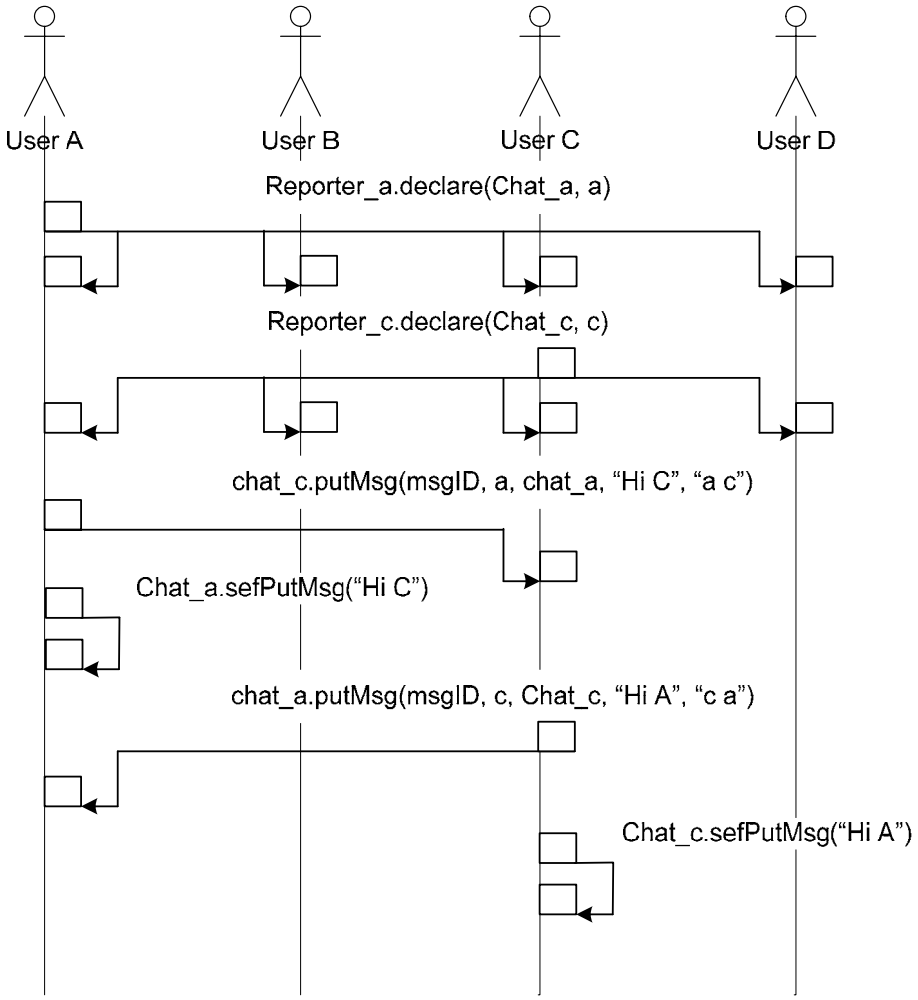


Figure 3 Instant Chat between 4 users Chat System

Group Chat

When user A wants to chat with the user B and C, it looks up the corresponding *Chat* unihandles in the active user list and calls the **putMsg ()** method on B’s and C’s *Chat* object’s unihandle respectively, which are chat_b and chat_c, passing in the data (**msgID**, **a, Chat_a**, **message, “a b c”**). Only the destined receipt, user B and user C, rather than the whole group executes the **putMsg ()** method and then displays the chat message. All the other users in service that are not involved in this chat session do not execute any method about this chat at all so they can not see any chat message from the conversation among A, B and C.

Chat Log Recovery

Every time when a user generates a message with some other user/users, its own **msgID** will increase by 1 automatically. Every time when a user gets a message, it will detect if the message's **msgID** is equal its own **msgID** + 1 and the participants is equal its own username concatenate with message sender's username. If yes, it will save the message and display it otherwise it will request the sender, got from the coming message, to resend the message for the same participants and synchronized the current **msgID**.

Since my project runs at LAN environment to simulate the wireless ad hoc environment, even M2MI is designed based on UDP, the package lost and user out of service area does rarely happen. So I write a test case, which is that every user discards the very first three messages from another user. Then when the same user sends out the 4th message, the user will detect the message lost and request for message lost recovery and **msgID** synchronization.

Figure 4 shows a sequence of M2MI invocation for message lost detection and recovery for 4 devices while message sending between 2 users. First, when every user joins the group, it declares itself with calling **declare ()** method on the omnihandle **theUserReporter** and pass in its own **Chat** object, which is attach to a unihandle. Then every other users call the **declare ()** methods, get and save the unihandle. When A wants to talk with C only, A searches its user list and get C's **Chat** unihandle **chat_c**, and calls the **putMsg ()** method on **chat_c**, passes in the message, the **MsgID**, and A's **Chat** object **chat_a**. According to my test case, C will discard the first three messages from A. When user A sends out the 4th message, C compare the coming **msgID,** which is 4, with his own **msgID** associated with the participants "**a c**"**,** which is 0, and found out itself lost the first three messages. Then C calls **request ()** on **chat_a**, passing in **thePar "a c", chat_c** and the **msgID** that C expected. Then A picks the message associated with **thePar "a c"** from his sending out message buffer with the **msgID** ranges from 1 to the latest one, then calls **getLostMsg ()** on **chat_c** to sends out all C 's need. Now only C gets the lost messages and displays that on C. and has C's **msgID** synchronized with A.

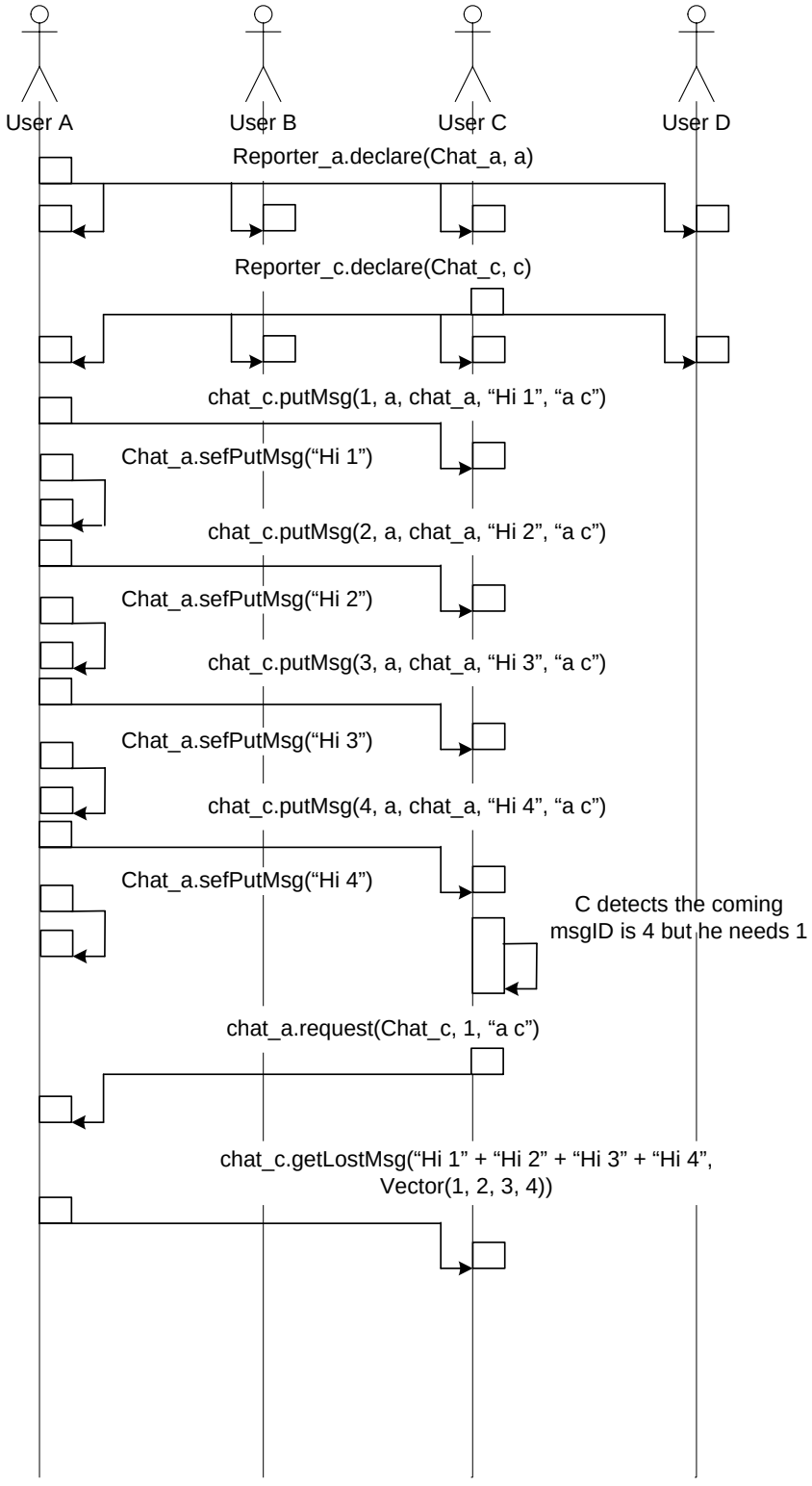


Figure 4 Message Lost Detect and Recovery for 4 Devices

3.2 Appointment Making Groupware System

This chapter descripts the background and problem, design pattern based on the M2MI mechanism, and the design specification of the *Appointment Making Groupware System*.

3.2.1 Background and Problem

The ***Appointment Making Groupware System*** is designed for appointment proposal, acceptance, rejection, and cancellation for multiple participants (synchronous, namely, multiple users send out appointments at the same time) that are nearby (like in a conference room).

The appointment making groupware system can perform the following functions and may encounter the following problems.

Make an Appointment and Accept (or Reject) It

A user can create an appointment and send it to another user. The information of the appointment can be displayed on both users' devices. After both users accept the appointment, the data will be saved in both users' devices otherwise it will be discarded

To achieve that, we may have the following problems:

- Every user must have a local active user list so it can look up this list to find the user it wants to send appointment to
- Every user can receive appointment request and then send back the reply to accept it or reject it
- Once an appointment is accepted by two users, it will be saved in the two users' local devices otherwise it will be discarded
- The information of appointments is only displayed on the two users' devices involved in the appointment making process. This can avoid unnecessary data transmission and broadcast storm
- Every active user should keep notifying all the other active users if it is still available to receive appointment

We will discuss the solution in the design specification section.

Update (or Cancel) an Existing Appointment and Accept (or Reject) It

A user can send the update or cancel request of an existing appointment to another user involved in that appointment, and the receiver can accept or reject it. The data will be saved in both users' devices if both user accept it otherwise it will be discarded.

To achieve that, we may have the following problems:

- Every user must have a local appointment list so it can find the existing appointment and the other user involved in it, and then send the updating (include canceling) request to the user
- Every user can receive updating appointment request and then send back the accepting reply or rejecting reply
- Once the update of an appointment is accepted by two users, the updated appointment will be saved in the two users' devices otherwise it will be discarded. If an appointment is canceled, it will be removed from both users' devices
- The information of appointments is only displayed on the two users' devices involved in the appointment making process
- Every active user should keep notifying all the other active users if it is still available to receive appointment

We will discuss the solution in the design specification section.

Exceptions

We need to handle the exception in the application level since the data transmission of M2MI-based collaborative groupware system is not 100% reliable. Data may be lost during broadcasting.

The following problems may occur:

- While new devices come in, all users need to get an updated user list
- If the receipt is out of the broadcasting area or does not reply, the sender will wait a certain time for the reply and then try additional two times, if the receipt still not reply, then the process fails with an appropriate message
- When a sender sends an appointment with another device, if any interruption, like data transferred error, broadcasting problem, or just the receipt walks out of broadcasting area, this error occurs. The application will check the date reliability and integrity to make sure all the users get the correct data

We will discuss the solution in the design specification section.

3.2.2 Design Pattern Based on M2MI Mechanism

This chapter descripts the M2MI-based design pattern to give a solution of *M2MI Collaborative Groupware System.*

Pattern Name

***M2MI Collaborative Groupware* (a documenting design pattern [7])**

Intent and Motivation

The ***M2MI Collaborative Groupware*** pattern applies to the situations in which small computing devices, running in ad hoc network, need to perform the following operations: event creation, event message sending, event message receiving, event confirmation, event rejection, event updating, and event display, etc.

Applicability (or Prerequisites)

The ***M2MI Collaborative Groupware*** pattern is used in classical ad hoc environment for multiple participants' event making, which means no central server, no network administration, no complicated routing protocol and no complicated system development. It is suitable for small set of users, small computing devices, and limited working space. My project will design this pattern and simulate it in LAN environment.

Solution

The following class will be developed when implement the ***M2MI Collaborative Groupware*** pattern.

Each ***M2MI Collaborative Groupware*** application exports an object implementing these following interfaces for user identification and leaving notification. The application obtains an omnihandle for this interface:

```
public interface UserReporter {
        // developer can add more arguments and more method
        public void declare (Amker theAmker, String theUserName);
        public void announce (AMker theAmker, String theUserName);
        public void leave (String theUserName);
        public void goodby (String theUN);
        …}
```

Interface that is designed for sending appointment, accepting appointment, saving valid appointment, rejecting appointment, canceling existing appointment, requesting lost appointment package and latest appointment ID, and retrieve lost appointment package with specified appointment ID.

```
public interface AMker {
        // developer can add more arguments and more method
        public void putApp (AMker sender, Appointment myApp);
        public void selfPutApp (Appointment myApp);
        public void accept(String theID, String thePtcpt, String theReason);
        public void saveIt(Appointment theApp, String theReason);
        public void reject(String theID, String thePtcpt, String theReason);
        public void selfRejectIt(Appointment theApp, String theReason);
```

```
public void rejectIt(Appointment theApp, String theReason);
public void cancel(String theID, String thePtcpt, String theReason);
public void cancelIt(Appointment theApp, String theReason);
public void request(AMker requester, int startLost, String theParticipants);
public void getLostApp (Vector theLostApp);
…}
```

Participants

- ***M2MI Collaborative Groupware*** initial class, which declares an Amker process and have the member methods for user joining and leaving. The ***M2MI Collaborative Groupware*** application obtains an omnihandle from M2MI layer for this initial interface
- ***M2MI Collaborative Groupware*** appointment main class, which implements appointment sending, receiving, saving, rejecting and canceling. The ***M2MI Collaborative Groupware*** application obtains an unihandle from M2MI layer for this interface

Code Example

See the Design Specification section.

3.2.3 Design Specification

This chapter descripts the design details; include the exported interfaces, the member methods, and the algorisms, of the *Appointment Making Groupware System*.

Make an Appointment and Accept (or Reject) It

Each appointment maker application exports an object implementing these three following interfaces:

The following interface is designed for every user declares itself when it comes into the service. Then all the devices run this method on the omnihandle ***UserReport*** and get the unihandle **AMker theAmker** and **theAmker**'s **username** and keep it in their local user list.

```
public interface UserReporter {
    // developer can add more arguments and more method
    public void declare (Amker theAmker, String theUserName);
    public void announce (AMker theAmker, String theUserName);
    public void leave (String theUserName);
    public void goodby (String theUN);
    }
```

This interface is designed for every appointment maker to send appointment, accept appointment, save appointment, reject appointment, cancel an existing appointment, request lost appointment, recover lost appointment, request the latest appointment ID, and recover the latest appointment ID.

```
public interface AMker {
        // developer can add more arguments and more method
        public void putApp (AMker sender, Appointment myApp);
        public void selfPutApp (Appointment myApp);
        public void accept(String theID, String thePtcpt, String theReason);
        public void saveIt(Appointment theApp, String theReason);
        public void reject(String theID, String thePtcpt, String theReason);
        public void selfRejectIt(Appointment theApp, String theReason);
        public void rejectIt(Appointment theApp, String theReason);
        public void cancel(String theID, String thePtcpt, String theReason);
        public void cancelIt(Appointment theApp, String theReason);
        public void request(AMker requester, int startLost, String theParticipants);
        public void getLostApp (Vector theLostApp);
        }
public interface AMframeListener {
        public void send(Appointment myApp);
        public void accept(String theID, String thePtcpt, String theReason);
        public void reject(String theID, String thePtcpt, String theReason);
        public void cancel(String theID, String thePtcpt, String theReason);
        }
```

The Firgure 5 show how new user declares itself, sends out an appointment, and accepts an appointment. When the service starts, the existing user A exports a *AmkerObject* object that implements the interface *Amker* and *AmkerFrameListener* to M2MI layer and gets a unihandle **Amker_a** from M2MI layer for the *AmkerObject*. And then it exports a *UserReporterObject* Object that implements the interface *UserReporter*. Then the application gets an omnihandle **Reporter_a** from M2MI layer for interface *UserReporter* and passes in its own unihandle ***AmkerObject* Amker_a** and its own ***username a*** as parameters. Finally, the user executes the **declare ()** method every 3 seconds, and then the other existing user run the **declare ()** method on the same *UserReporterObject* omnihandle **Reporter_a** and save the **username a** and unihandle **Amker Amker_a**. Every new user will do the same thing so an active user list has been generated at every active user's local device.

When user A wants to make an appointment with user C, first, the application running on device A looks up device C's ***AmkerObject*** unihandle, which is **Amker_c**. Second, device A calls the **send ()** method on C's unihandle **Amker_c** and passes in the appointment (data class include both appointment requester's and receiver's unihandle, which is **Amker_a** and **Amker_c,** content and the appID). Only C executes the **send ()** method, and gets the appointment. If C wants to accept the appointment then C calls **accept ()** method on the requester's unihandle, which is **Amker_a**, to reply to device A. When A get the **accept ()** method invocation, A stores this appointment to its local disk. If C decides to reject this appointment, it will call the **reject ()** method on **Amker_a** instead and discard the coming appointment. When A gets the **reject ()** method invocation, A will discard the appointment too.

Inactive User Removal

Figure 5 also shows how an active user leaves and all the other users remove it from their active user list. When User B wants to leave, a **clearGone()** thread at B calls **leave(b's userName)** method on the omnihandle **theReporterObject** and so all the other user will know B's leaving and remove B from their active user list. And a **backApp ()** thread at B will save all the existing valid appointment to a local file ***AppointmentList***.

Update (or Cancel) an Existing Appointment and Accept (or Reject) It

Figure 6 shows the how user cancels an existing appointment (Updating an appointment is implemented as two phase, first, cancel the existing appointment, then send out a new one)

When user A wants to cancel an existing appointment with user C, first, the application running on device A looks up device C's ***AmkerObject*** unihandle, which is **Amker_c**. Second, device A calls the **cancel ()** method on C's unihandle **Amker_c** and passes in the appointment ID, the requester and the receiver, and the reason. Only C executes the **cancel ()** method. So both of them will remove the existing appointment and return the message to the user.

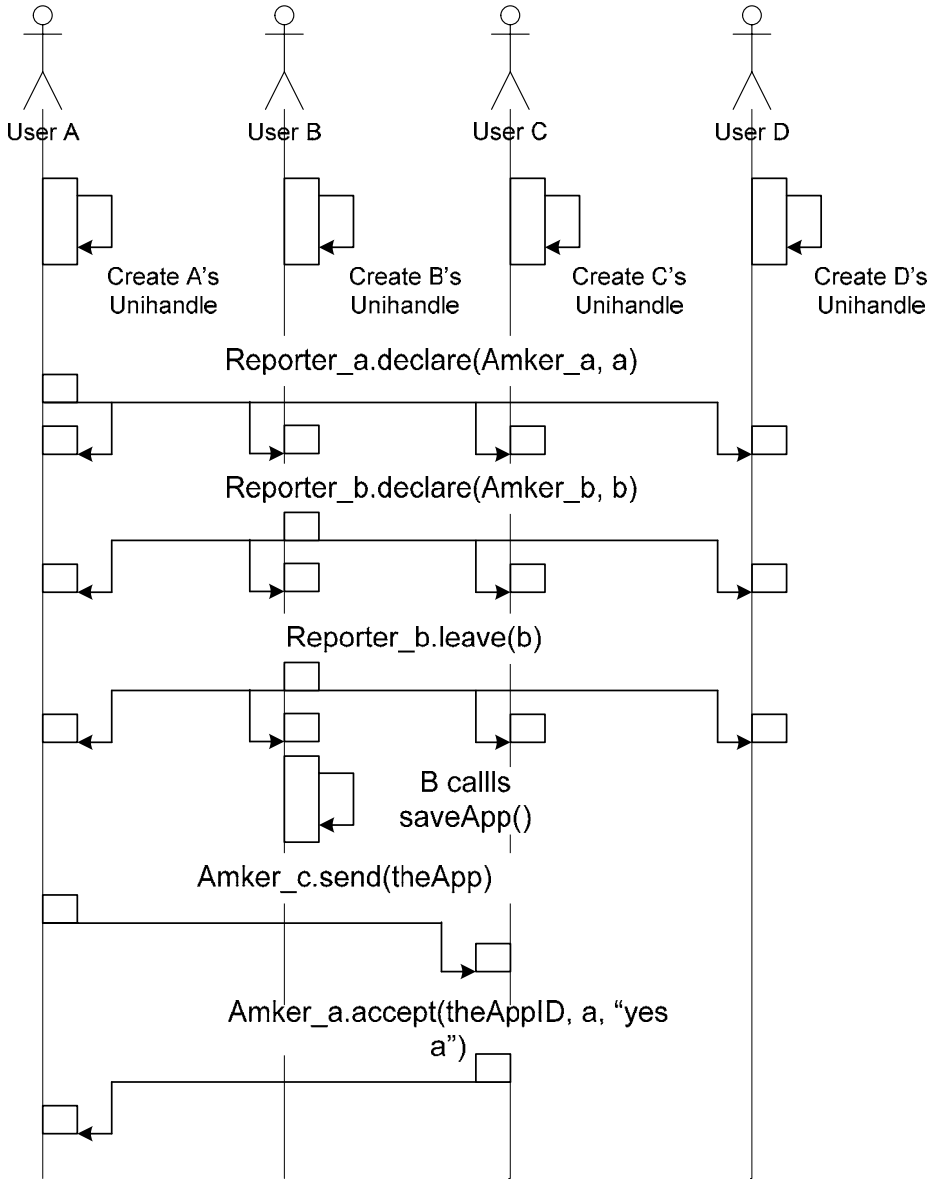


Figure 5 User Declaration, Appointment Sending and Accepting an Appointment between 4 user System

Exceptions

First, since we require every AmkerObject declares itself every 3 seconds and everything inactive user give the notification at the application level, so any new later join in user will get the up-to-date user list.

Every time when a user generates an appointment, its own **appID** will increase by 1 automatically associated with the participants (Those two factors can identify an appointment). Every time when a user gets an appointment, it will detect if the

appointment's **appID** is equal its own **appID** + 1. If yes, it will save the appointment and display it otherwise it will request the sender, got from the coming appointment, to resend the message associated with the participants and synchronized the current **appID**.

Since my project runs at LAN environment to simulate the wireless ad hoc environment, even M2MI is designed based on UDP, the package lost and user out of service area does rarely happen. So I write a test case, which is that every user discards the very first three appointments from another user. Then when the same user sends out the 4th message, the user will detect the package lost and request for appointment lost recovery and **appID** synchronization.

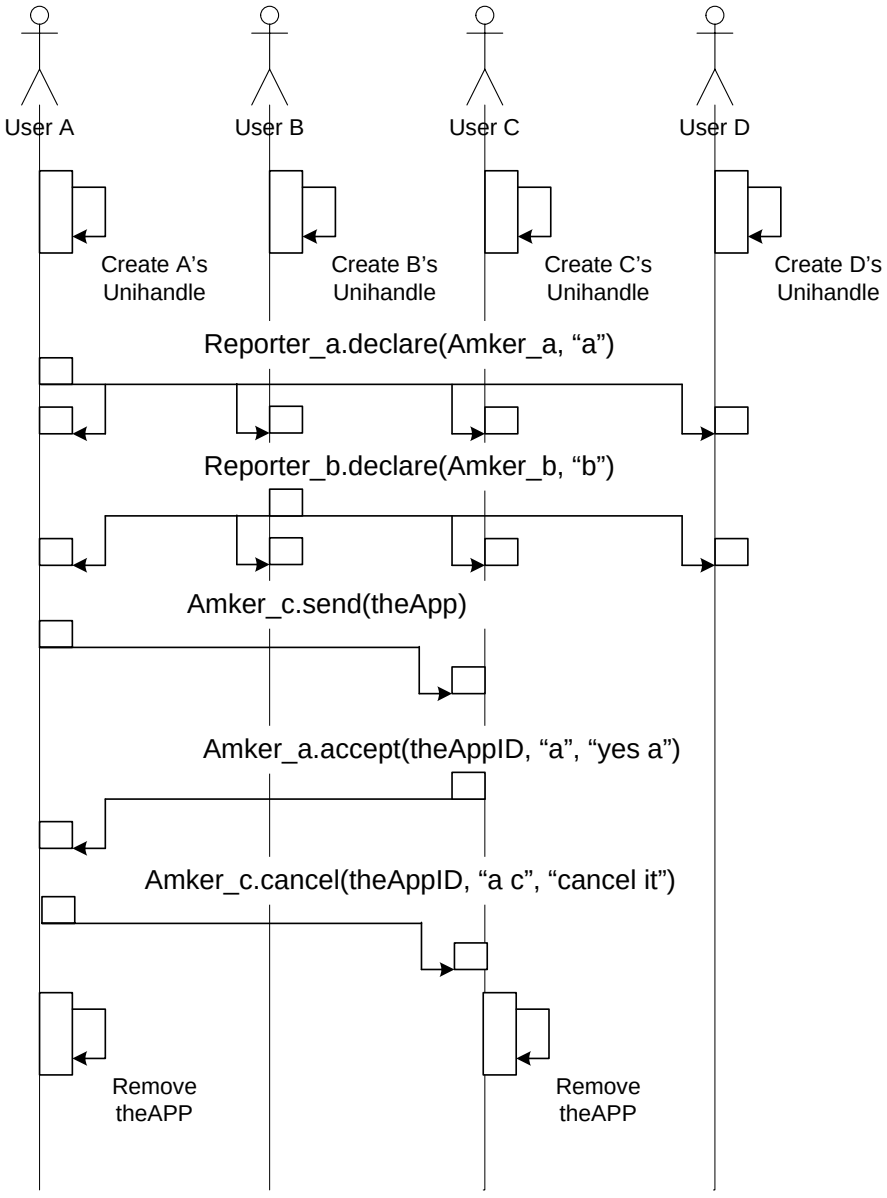


Figure 6 Cancellation of an Existing Appointment between 4 users System

Figure 7 shows a sequence of M2MI invocation for appointment lost detection and recovery for 4 devices. First, when every user joins the group, it declares itself with calling **declare ()** method on the omnihandle **theUserReporter** and pass in its own **Amker** object, which is attach to a unihandle. Then every other users call the **declare ()** methods, get and save the unihandle. When A wants to talk with C only, A searches its user list and get C's **Amker** unihandle **Amker_c**, and calls the **send ()** method on **Amker_c**, passes in the appointment. According to my test case, C will discard the first three appointment requests from A even if C already receives it. When user A sends out the 4th appointment, C compare the coming **appID,** which is 4, and found out itself lost the first three messages. Then C calls **request ()** on **Amker_a**, passing in **Amker_c**, the participants "**a c**", and the **appID** that C expected. Then A picks the appointments associated with the participants "**a c**" from his sending out appointment buffer with the **appID** ranges from 1 to the latest one, then calls **getLostApp ()** on **Amker_c** to sends out all C 's need. Now only C gets the lost messages and displays that on C. and has C's **appID** synchronized with A.

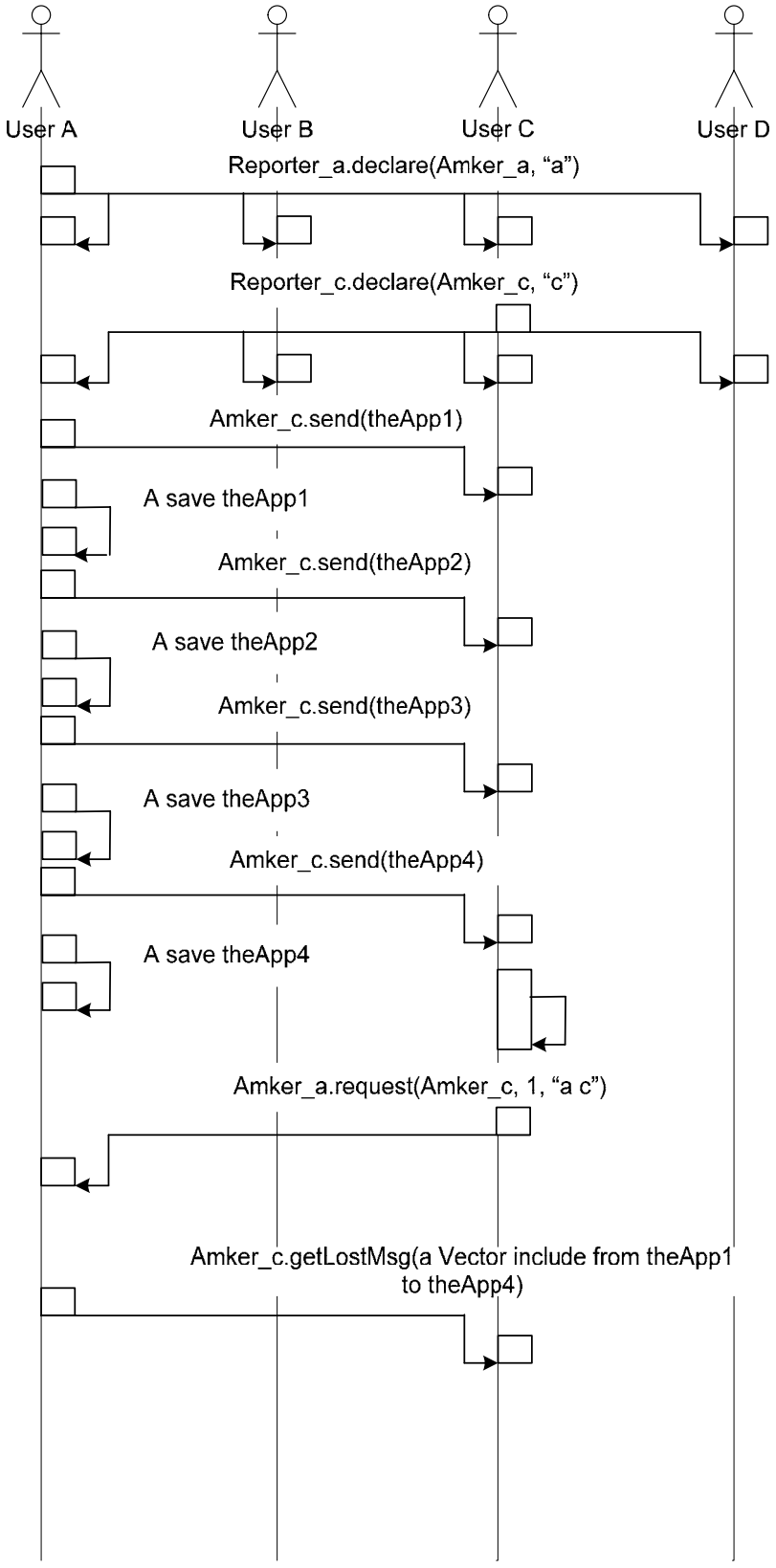


Figure 7 Appointment Lost Detect and Recovery for 4 Users System

3.3 Distributed Solution of Dining Philosophers

This chapter descripts the background and problem, design pattern based on the M2MI mechanism, and the design specification of the *Distributed Solution of Dinning Philosophers.*

3.3.1 Background and Problem

The dining philosophers is a classical example to illustrate various problems(like deadlock and equal opportunity for all users) that can occur when many synchronized threads are competing for limited shared resources.

"Fixed number **k** philosophers are sitting at a circular table. Each philosopher spends his life alternately thinking and eating. In the center of the table there is a bowl of rice. Between each pair of philosophers is one chopstick. When hungry, each philosopher must get his left and right chopstick before he can start eating. After he is done, he puts down the chopsticks and let other philosophers eat. The philosophers must find some way to share chopsticks such that they all eat with a reasonable frequency and avoid deadlock." [6] (Figure 8) (use **k** = 5 as example)

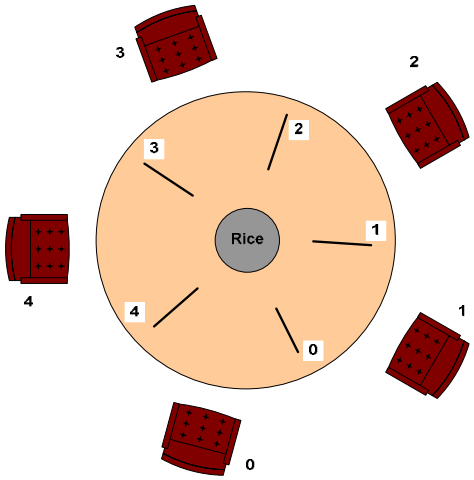


Figure 8: Dining Philosophers problem

To provide a distributed solution of Dining Philosophers, we may encounter the following problems:

- We assume the **k** philosophers (consumers) are already sitting around a circular table and the **k** chopsticks (shared resources) are already put on the table. That means we don't handle later join-in consumers and resources.
- Every philosopher and chopstick is in separated Java virtual machine or different device

- Every philosopher knows which chopsticks he needs
- Every chopstick can only be occupied by one philosopher each time, that means, when a shared resource is occupied, it will block all the occupy requests from other consumers
- Every Philosopher must have equal opportunity to eat rice
- Deadlock must be avoided, that means, we should avoid the situation that every philosopher is waiting for another one (circular deadlock).

We will discuss the solution in the design pattern section.

3.3.2 Design Pattern Based on M2MI mechanism

This chapter descripts the M2MI-based design pattern to give a solution of *Distributed Dinning Philosophers.*

Pattern Name

M2MI Shared Resource Allocation **(a documenting design pattern [7])**

Intent and Motivation

The ***M2MI Shared Resource Allocation*** pattern applies to situations in which small computing devices, running in ad hoc network, need to compete for some shared resource. Every device should have equal opportunity to use the shared resource and avoid deadlock. My project will design this pattern and simulate it in LAN environment.

Applicability (or Prerequisites)

The ***M2MI Shared Resource Allocation*** pattern is used in classical ad hoc environment for multiple devices competing for shared resource, which means no central server, no network administration, no complicated routing protocol and no complicated system development. It is suitable for fix-number users and resource, small computing devices, and limited working space.

Solution

The following class will be developed when implement the ***M2MI Shared Resource Allocation*** pattern.

Each resource in the ***M2MI Shared Resource Allocation*** pattern exports an object implementing the following interface to identify the new resource joining in. The application obtains an omnihandle for this interface.

```
public interface ResourceReporter {
        //developer can add more arguments and more methods
```

```
public void announce (Resource theResource, int theID);
public void addResource (Resource theRecource, int theID);
…}
```

Each resource implements the following interface to implement the **use ()** and **release ()** method. The application obtains a unihandle for this interface.

```
public interface Resource {
    //developer can add more arguments and more methods
    public void use (int consumerID, Consumer theConsumer);
    public void put (int consumerID, Consumer theConsumer);
    …}
```

Each consumer implements the following interface. The application obtains a unihandle for this interface.

```
public interface Consumer {
    //developer can add more arguments and more methods
    public void getIfPickLow(int theChopID, boolean ifPickLow);
    public void getIfPickHigh(int theChopID, boolean ifPickHigh);
    public void cancelLow(int theChopID, boolean ifPickLow);
    public void cancelHigh(int theChopID, boolean ifPickHigh);
    …}
```

When every resource sits on the table (namely, comes into service), it declares itself with **announce ()** method and say it is available at the beginning. Assume all consumers have already sit around the table (namely, ready for receive the resource declaration), every consumer has the fully available resource list now (Initially, all resources are available). Then every consumer **i** can search the available resource list for its left **(i + k - 1) % k** and right **(i + k) % k** resources' unihandles. When a consumer wants to use the resources, it will call the **use()** method on its lower number resource's unihandle first. If the lower number resource is already occupied by another consumer, the resource will call the **cancelLow()** on the requesting consumer's unihandle, which it gets from the **use()** invocation. Only if the consumer get its lower number resource, it will continue to call the **use()** method on its higher number resource's unihandle. If the higher number resource is already occupied by another consumer, the requesting consumer will hold its lower number resource, requesting the higher number resource after a timeout until it gets all the resources it needs and continues to use them then release them.

Resource is designed with exclusive access. A resource can be only occupied by just one consumer at the same time. An occupied resource will block all the other consumers and

send back a cancel message to notify the blocked consumers it was blocked until it is released.

Deadlock is successfully prevented.

3.3.3 Design Specification

The following interface is designed for every chopstick declares itself when it comes into the service. Then all the devices run this method on the omnihandle ***ChopReport*** and get the unihandle **Chop c** and **c**'s **username** and keep it in their local user list.

```
public interface UserReporter {
    //developer can add more arguments and more methods
    public void announce (Chop theChop, int theID);
    public void addChop (Chop theChop, int theID);
    }
```

The following interface is designed for chopstick's use and release.

```
public interface Chop {
    //developer can add more arguments and more methods
    public void use (int philID, Phil thePhil);
    public void put (int philID, Phil thePhil);
    }
```

The following interface is designed for every philosopher.

```
public interface Phil {
    //developer can add more arguments and more methods
    public void getIfPickLow(int theChopID, boolean ifPickLow);
    public void getIfPickHigh(int theChopID, boolean ifPickHigh);
    public void cancelLow(int theChopID, boolean ifPickLow);
    public void cancelHigh(int theChopID, boolean ifPickHigh);
    }
```

Firgure 9 shows how this distributed solution of dinning philosophers work. When the simulation starts, all the philosophers and chopsticks are ready. Fist, all the philosophers exports a *PhilObj* object that implements the interface *Phil* to M2MI layer and gets a unihandle **Phil_i** (i is the philID) from M2MI layer for the *PhilObj*. The existing chop **0** exports a *ChopObj* object that implements the interface *Chop* to M2MI layer and gets a unihandle **Chop_0** from M2MI layer for the *ChopObj*. Since **Chop_0** is a unihandles, only the destined device executes the methods on **Chop_0**. And then it exports a

ChopReporterObj Object that implements the interface *ChopReporter*. Then the application gets an omnihandle **Reporter_0** from M2MI layer for interface *ChopReporter* and passes in its own unihandle ***ChopObjt* Chop_0** and its own ***chopID 0*** as parameters. Finally, all the chopsticks and philosophers execute the **announce ()** method and then save the **chopID 0** and the unihandle **Chop Chop_0**. All the other chopsticks will do the same thing so an active chopstick list has been generated at every active philosopher's local device. All the chopsticks are initially available to use. Since only philosophers need to know chopsticks but chopsticks don't need to know all the philosophers so only chopsticks have to declare themselves and keep an fully chopstick list.

For Example, the omnihandle **Reporter_1** is created with code:

```
M2MI.export (this, ChopReporter.class) ;
ChopReporterObject Reporter_1 = (ChopReporter) M2MI.getOmnihandle
(ChopReporter.class);
```

The unihandle **Chop_1** is created with code:

```
M2MI.export (this, Chop.class) ;
Chop Chop_a = (Chop) M2MI.getUnihandle(this, Chop.class) ;
```

Passing in **Chop_1** and **1's chopID 1** as parameters for the declare method on the omnihandle **ChopReporterObejct**:

```
ChopReporterObj theReporterObj = new ChopReporterObject () ;
theReporterObj.announce(Chop_1, 1) ;
```

When philosopher 0 wants to use chopstick 0 and 4 to eat, first it will call the **use()** method on its lower number chopstick 0's unihandle first. Let suppose philosopher 0 gets the chopstick 0. So philosopher 1 will be blocked by chopstick 0 and send the request after a timeout until philosopher 0 is done using chopstick 0 and release chopstick 0. Then philosopher 0 will continue to call **use()** method on its higher number chopstick 4. If chopstick 4 is already occupied by philosopher 4, then philosopher 4 must have the chopstick 3 before that since every philosopher has to get its lower number chopstick first. Then philosopher 4 can continue to use chopstick 3 and 4 then release both of them. So philosopher 0 can get chopstick 4, use chopstick 0 and 4, and then release them. If chopstick 4 is free, then philosopher 0 can start to eat and then release chopstick 0 and 4 so philosopher 4 can finish eating at the second order.

Chopstick is designed with exclusive access. A chopstick can be only occupied by just one philosopher at the same time. A free chopstick will call the **getIfPickHigh()** or **getIfPickHigh()** method(depends on this chopstick is the lower number or higher number chopstick of the requesting philosopher)on the requesting philosopher to notify him it already occupy the chopstick then chopstick can proceed. An occupied chopstick will

block all the other philosophers and call the **cancelHigh()** or **cancelLow()** (depends on this chopstick is the lower number or higher number chopstick of the blocked philosopher)on the blocked philosopher to notify him it was blocked until the chopstick is released.

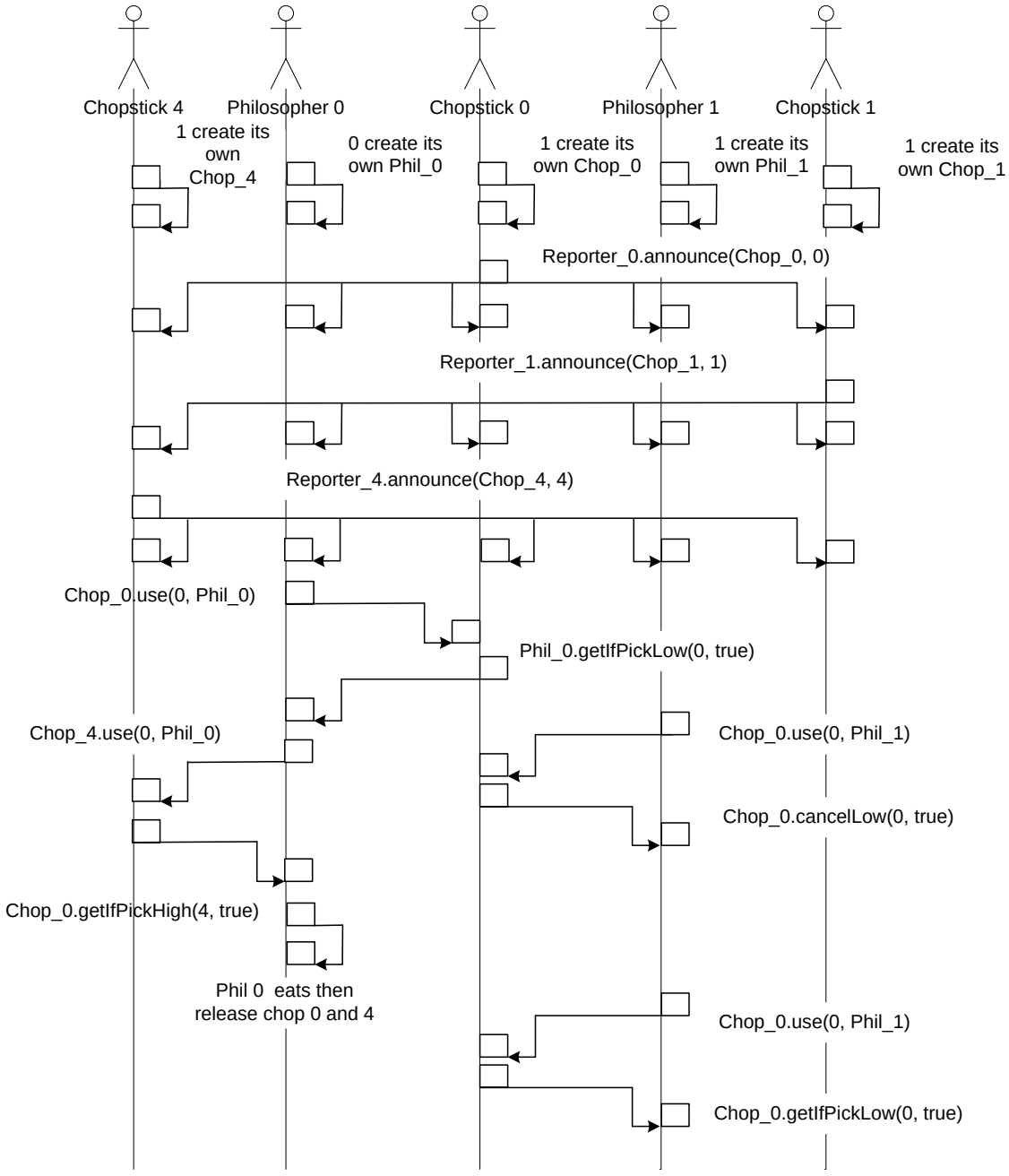


Figure 9 the simulation of Distributed Dinning Philosophers
(Only show 3/5 philosophers and 2/5 chopsticks)

4. User Manual

This chapter provides the user manuals of these three systems: *Multiple Participants Chat System*, *Appointment Making Groupware System*, and the *Distributed Solution of Dinning Philosophers.* The manual is include how to compile the systems, how to run M2MP Router, and how to use the collaborative systems

4.1 Multiple Participants Chat System

This chapter descripts the usage of M2MP Router, the compilation and the usage of multiple participants chat system.

4.1.1 How to compile

Unpack kxc.jar first or under kxc/m2miapi20020702/lib directory, run

javac edu/rit/m2mi/test/chat/chatFinal/*.java

It is the final version of multiple participants chat system

javac edu/rit/m2mi/test/chat/chatTest/*.java

It is the test version of multiple participants chat system with a test case. Every chat system will automatically discard the very 3 first messages. When the sender sends the 4th message, the receiver can detect the lost 3 messages and request from the sender to get the lost messages recovered.

4.1.2 How to run M2MP Router

Before you can run the chat system, you must run the M2MP Router (class edu.rit.m2mp.ip.M2MPRouter) in a separate process but in the same device with its corresponding chat system.

Under kxc/m2miapi20020702/lib, run

java edu.rit.m2mp.ip.M2MPRouter

The following command provides the details of data transfer and routing of M2MP Router.

java edu.rit.m2mp.ip.M2MPRouter –v

Running on Two Hosts

You can run different instance of Chat system on separate device. In this case, it is necessary to configure the M2MP Router to broadcast M2MP package between the hosts. If you just want to run chat system on two hosts, h1 and h2 (h can be the actual host name, like holly, New York, sauron, treebeard, legolas, etc. or the IP address. I prefer use host name at CS lab environment), the run in a separate process on h1:

java edu.rit.m2mp.ip.M2MPRouter –s h2 –r h1

And run in a separate process on h2:

java edu.rit.m2mp.ip.M2MPRouter –s h1 –r h2

Command with more data transmission details

java edu.rit.m2mp.ip.M2MPRouter –s h1 –r h2 -v

Now the M2MP package can be transferred between h1 and h2.

Running on Two or More Hosts

If you want to run chat system on two or more host, you will use IP multicasting to broadcast the M2MP package. Run in a separate process on each host involved:

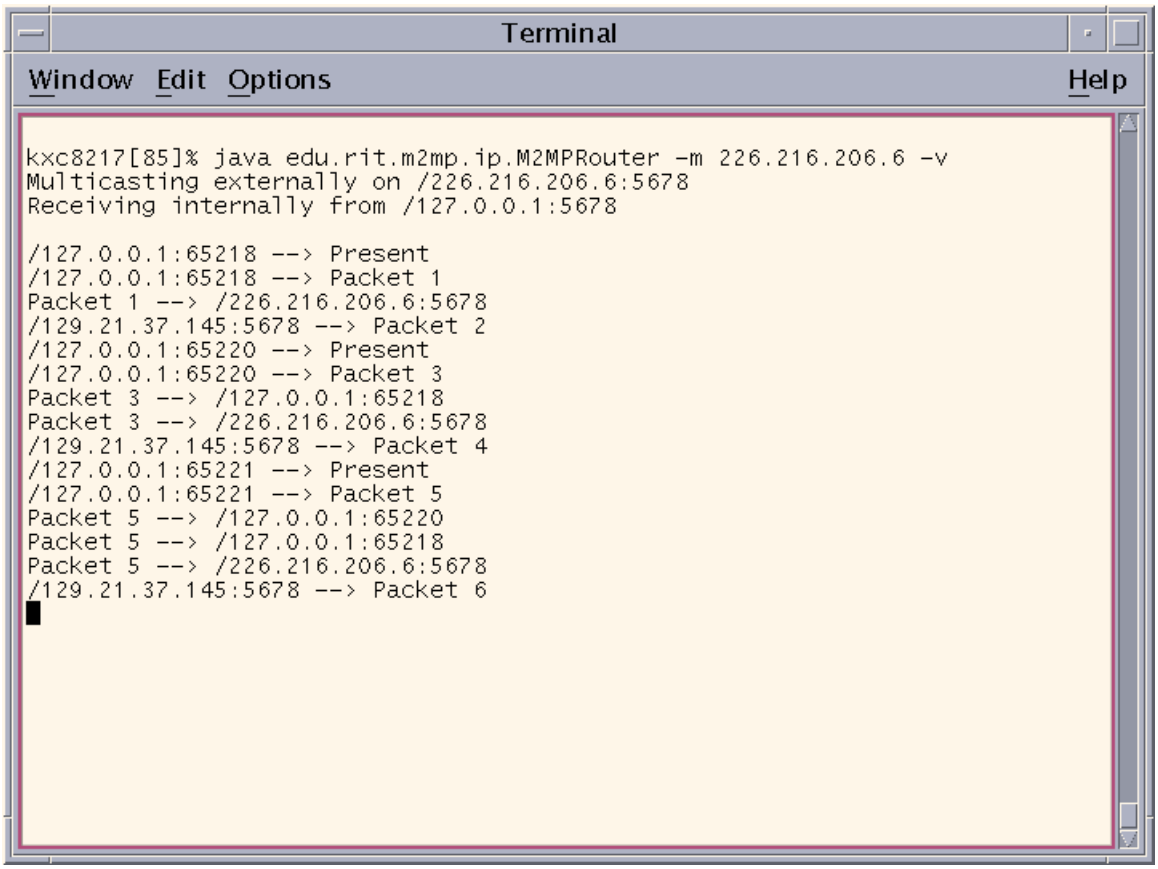
java edu.rit.m2mp.ip.M2MpRouter –m 226.216.208.1

The command line argument is a multicast IP address to send message to and receive message from. The multicast IP address must be in the range 224.0.0.0 to 239.255.255.255. Now message can be exchanged within a set of host.

Command with more data transmission details

java edu.rit.m2mp.ip.M2MPRouter –-m 226.216.208.1 -v

It will run from the command line like this:



4.1.3 How to use M2MI-based chat system

To run an instance of the chat system, you should run the following command under kxc/m2miapi20020702/lib

java edu.rit.m2mi.test.chat.chatFinal.ChatExc <username>

<Note>The username must be unique since I use username as the user identification.

Then user will see the “My Chat log”, “Online User”, “message field” with “Send” button, and the “selected user field” with “Select” button.

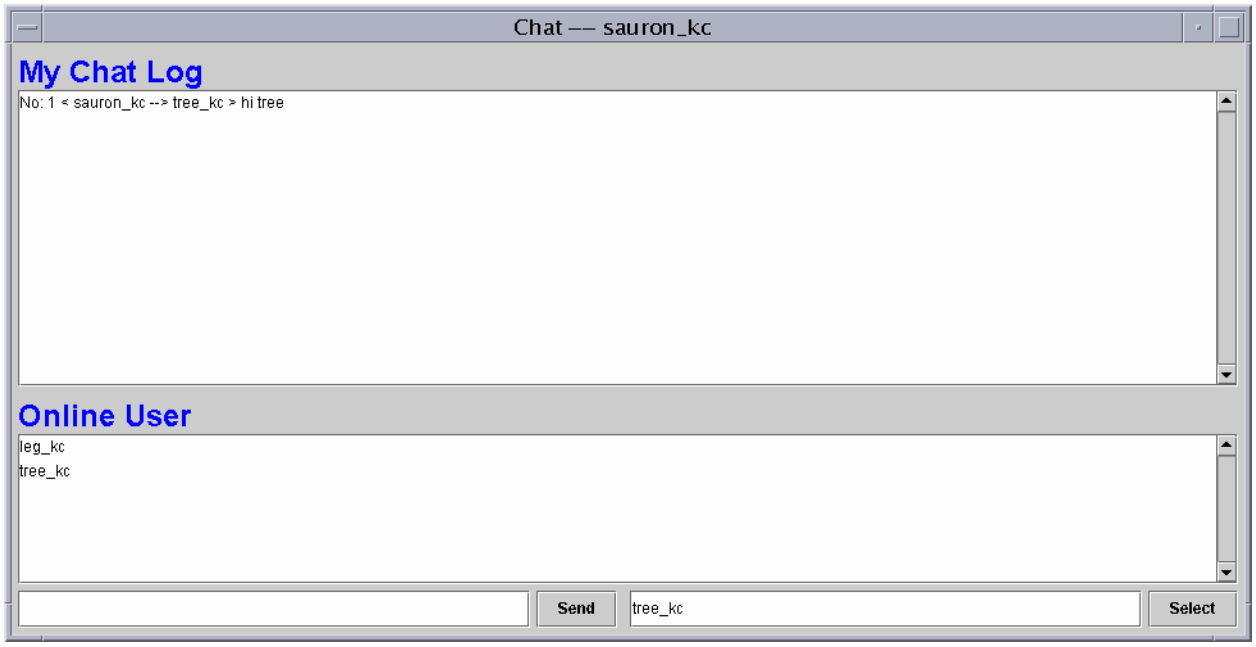
A user can only select one or a set of the users (from their usernames) from the “Online User” list. When a user exits or steps out of the service area for a certain timeout, it will be removed from the “Online User” list and not available to receiver message anymore until it comes back to service.

Two User Chat

For example, when user **sauron_kc** wants to send a message to user **tree_kc**, he types in “tree_kc” (case sensetive) in the “Selected user field” and then press “Select” or Enter

key. Then **sauron_kc** can type in any message it wants to send to **tree_kc** in the "message field" and then press "Send" or Enter key.

It will display a UI like this:

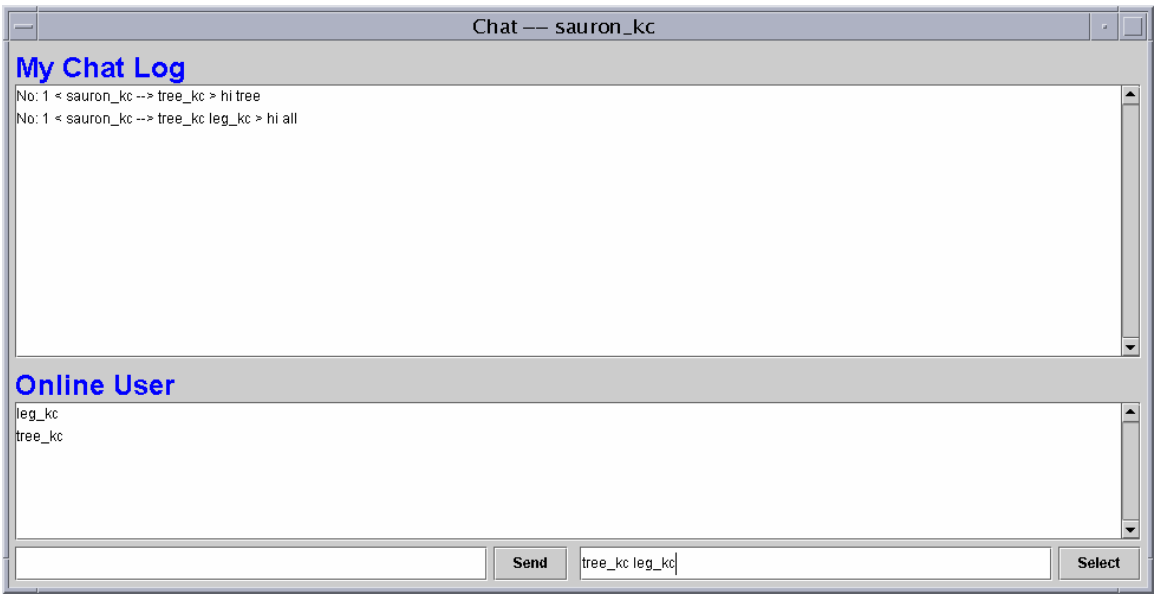


If **sauron_kc** wants to chat with other user/users, for example, **leg_kc**, he can type in "leg_kc" (case sensetive) in the "Selected user field" and then press "Select" or Enter key. Then **sauron_kc** can type in any message it wants to send to **leg_kc** in the "message field" and then press "Send" or Enter key.

Multiple Users Chat

If **sauron_kc** wants to chat with multiple users, just types in all of their usernames one by one in the "Selected user field", separating by space and press "Select" or Enter key. Then every message **sauron_kc** types in "Message field" will display on all the devices of the selected users.

It will display a UI like this:

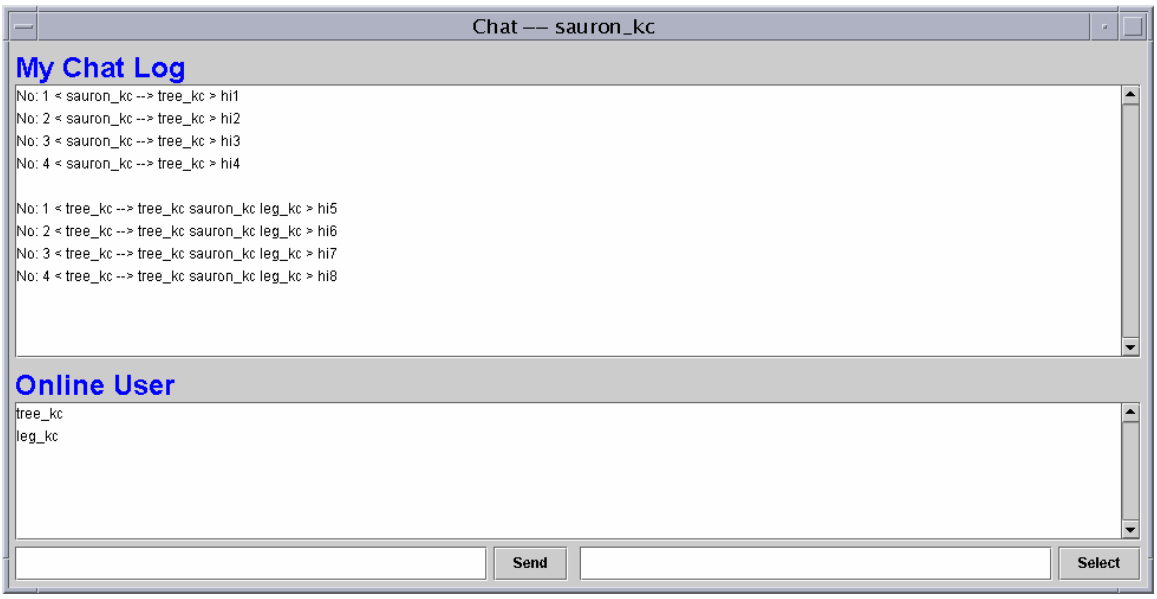


If **sauron_kc** wants to chat with other user/users, he can retype in the name/names of the user/users in the "Selected user field" and then press "Select" or Enter key. Then **sauron_kc** can type in any message it wants to send in the "message field" and then press "Send" or Enter key. Those messages will display on all the specified devices of the selected user.

Message log recovery

For message lost recovery, I wrote the test case which every user will discard the very first three messages from other users. Run the same command with the above but change the directory "chatFinal" to "chatTest". For example, **sauron_kc** sends "msg1", "msg2", and "msg3" to **tree_kc**. Although **tree_kc** actually gets them, it discards them like **tree_kc** never see them to simulate package lost for separate small computing devices in wireless ad hoc environment. When **sauron_kc** sends the 4^{th} message, **tree_kc** 's message log recovery mechanism will be triggered, detects what **tree_kc** needs and request from **sauron_kc** to get the all the lost message, "msg1", "msg2", and "msg3" and get its own **msgID** synchronized with the latest message ID.

It will display a UI like this:



4.2 Appointment Making Groupware System

This chapter descripts the usage of M2MP Router, the compilation and the usage of appointment making groupware system.

4.2.1 How to compile

Unpack kxc.jar first or under kxc/m2miapi20020702/lib directory, run

javac edu/rit/m2mi/test/app/appFinal/*.java

It is the final version of appointment making groupware system

javac edu/rit/m2mi/test/app/appTest/*.java

It is the test version of appointment making groupware system with a test case. Every appointment making groupware system will automatically discard the very 3 first appointments. When the sender sends the 4th message, the receiver can detect the lost 3 appointments and request from the sender to get the lost appointments recovered.

4.2.2 How to run M2MP Router

Before you can run the appointment making groupware system, you must run the M2MP Router (class edu.rit.m2mp.ip.M2MPRouter) in a separate process but in the same device with its corresponding appointment making groupware system.

Under kxc/m2miapi20020702/lib, run

java edu.rit.m2mp.ip.M2MPRouter

The following command provides the details of data transfer and routing of M2MP Router.

java edu.rit.m2mp.ip.M2MPRouter –v

Running on Two Hosts

You can run different instance of appointment making groupware system on separate device. In this case, it is necessary to configure the M2MP Router to broadcast M2MP package between the hosts. If you just want to run appointment making groupware system on two hosts, h1 and h2 (h can be the actual host name, like holly, New York, sauron, treebeard, legolas, etc. or the IP address. I prefer use host name at CS lab environment), the run in a separate process on h1:

java edu.rit.m2mp.ip.M2MPRouter –s h2 –r h1

And run in a separate process on h2:

java edu.rit.m2mp.ip.M2MPRouter –s h1 –r h2

Command with more data transmission details

java edu.rit.m2mp.ip.M2MPRouter –s h1 –r h2 -v

Now the M2MP package can be transferred between h1 and h2.

Running on Two or More Hosts

If you want to run appointment making groupware system on two or more hosts, you will use IP multicasting to broadcast the M2MP package. Run in a separate process on each host involved:

java edu.rit.m2mp.ip.M2MpRouter –m 226.216.208.1

The command line argument is a multicast IP address to send message to and receive message from. The multicast IP address must be in the range 224.0.0.0 to 239.255.255.255. Now message can be exchanged within a set of host.

Command with more data transmission details

java edu.rit.m2mp.ip.M2MPRouter –-m 226.216.208.1 -v

4.2.3 How to use M2MI-based Appointment Making system

Under kxc/m2miapi20020702/lib, run

java edu.rit.m2mi.test.app.appFinal.AppExc <username>

<Note> The username must be unique since I use username as a user ID.

Then user will see the “My Appointment log”, “Online User”, and “My Action log”.

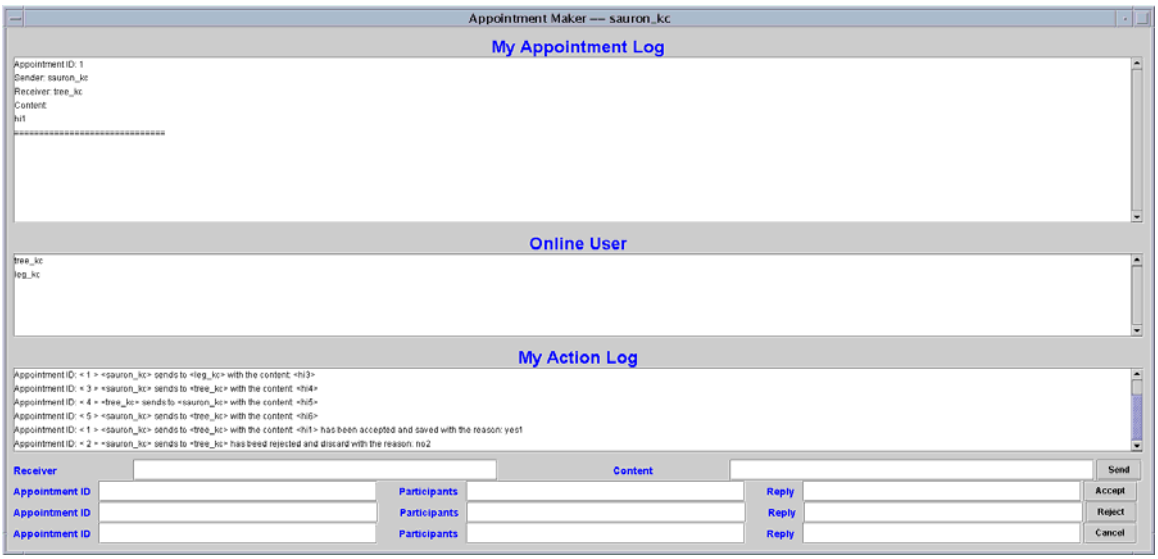
In this GUI, we also have “Receiver field”, and “Content” with “Send” button, the “Appointment ID”, “Participants”, “Reply” with “Accept” button, the “Appointment ID”, “Participants”, “Reply” with “Reject” button, and the “Appointment ID”, “Participants”, “Reply” with “Cancel” button,.

A user can only select one of the users (from its username) from the “Online User” list. When a user exits or steps out of the service area for a certain timeout, it will be removed from the “Online User” list and not available to receiver appointment anymore until it comes back to service.

Two User Appointments Making, Accepting, and Rejecting

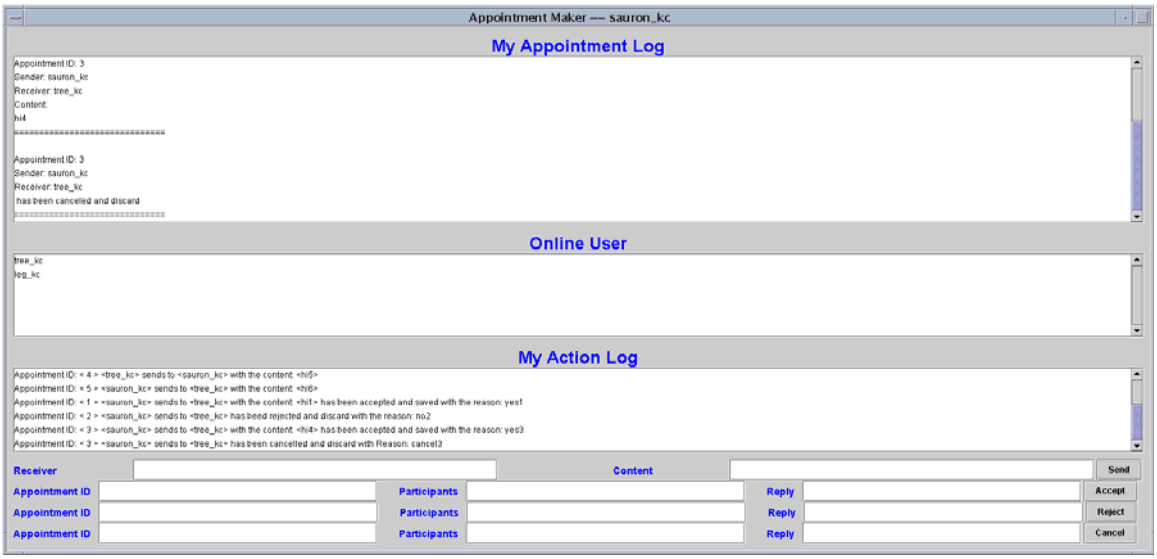
For example, when user **sauron_kc** wants to send an appointment to user **tree_kc**, it types in “tree_kc” (case sensetive) in the “Receiver field”, and put the content in “Content” and then press “Send”. Then **tree_kc** can see the coming appointment request at his “My Action Log”. If **tree_kc** wants to accept that, just type in the coming appointment ID in “Appointment ID”, “**sauron_kc tree_kc**” in “Participants”, type in any message it wants to send to **sauron_kc** in the “Reply” and press “Accept” then A will see the appointment has been accepted in A’s “My Action Log” and this appointment is showed on both **sauron_kc**’s and **tree_kc**’s “My Appointment log”. If **tree_kc** wants to reject this appointment, it will type in the same arguments but press “Reject” this time. **sauron_kc** will see it in his “My Action Log” and both side discard this appointment and show nothing in “My Appointment Log”.

It will display a GUI like this:



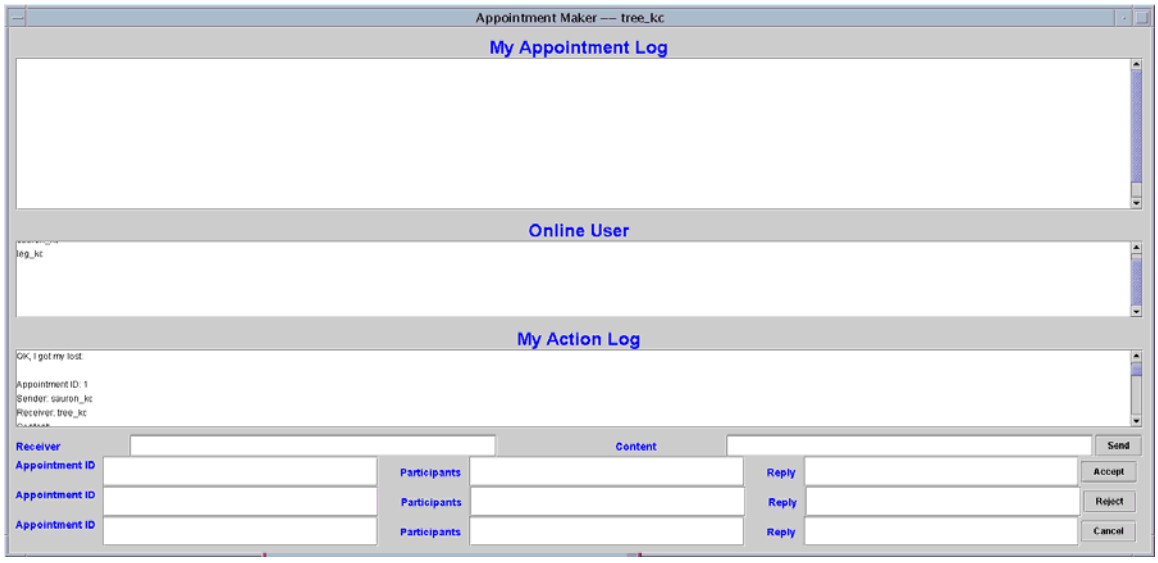
Two User Appointments Making Canceling and Updating

For example, when user **sauron_kc** (if he is the requester of this appointment) wants to cancel an existing appointment to user **tree_kc**, it types in this appointment ID in the "Appointment ID", "sauron_kc tree_kc" (case sensetive) in the "Participants", and put the reply message in "Reply" and then press "Cancel". Then **tree_kc** can see the coming appointment canceling request at his "My Action Log". Then this appointment cancellation is showed on both **sauron_kc**'s and **tree_kc**'s "My Appointment log" and both side discard this appointment.



Appointment log recovery

For appointment lost recovery, I wrote the test case which every user will discard the very first three appointments from other users. Run the same command with the above but change the directory “appFinal” to “appTest”. For example, **sauron_kc** sends “appointment 1”, “appointment 2”, “appointment 3” to **tree_kc**. Although **tree_kc** gets them, it discards them like **tree_kc** never see them to simulate the wireless ad hoc environment package lost. When **sauron_kc** sends the 4th appointment, **tree_kc** ’s appointment log recovery mechanism will be triggered, detects what **tree_kc** needs and request from **sauron_kc** to get the all the lost appointment package and get its own **AppID** synchronized with 4.



4.3 Dining Philosophers Distributed Solution

This chapter descripts the usage of M2MP Router, the compilation and the usage of dinning philosophers distributed solution.

4.3.1 How to compile

Unpack kxc.jar first or under kxc/m2miapi20020702/lib directory, run

javac edu/rit/m2mi/test/dp/*.java

It is the final version of dinning philosophers distributed solution since we don’t need to consider the message lost in this solution.

4.3.2 How to run M2MP Router

Before you can run the dinning philosophers system, you must run the M2MP Router (class edu.rit.m2mp.ip.M2MPRouter) in a separate process but in the same device with its corresponding dinning philosophers system.

Under kxc/m2miapi20020702/lib, run

java edu.rit.m2mp.ip.M2MPRouter

The following command provides the details of data transfer and routing of M2MP Router.

java edu.rit.m2mp.ip.M2MPRouter –v

Running on Two Hosts

You can run different instance of dinning philosophers system on separate device. In this case, it is necessary to configure the M2MP Router to broadcast M2MP package between the hosts. If you just want to run chat system on two hosts, h1 and h2 (h can be the actual host name, like holly, New York, sauron, treebeard, legolas, etc. or the IP address. I prefer use host name at CS lab environment), the run in a separate process on h1:

java edu.rit.m2mp.ip.M2MPRouter –s h2 –r h1

And run in a separate process on h2:

java edu.rit.m2mp.ip.M2MPRouter –s h1 –r h2

Command with more data transmission details

java edu.rit.m2mp.ip.M2MPRouter –s h1 –r h2 -v

Now the M2MP package can be transferred between h1 and h2.

Running on Two or More Hosts

If you want to run dinning philosophers system on two or more hosts, you will use IP multicasting to broadcast the M2MP package. Run in a separate process on each host involved:

java edu.rit.m2mp.ip.M2MpRouter –m 226.216.208.1

The command line argument is a multicast IP address to send message to and receive message from. The multicast IP address must be in the range 224.0.0.0 to 239.255.255.255. Now message can be exchanged within a set of host.

Command with more data transmission details

java edu.rit.m2mp.ip.M2MPRouter –-m 226.216.208.1 -v

4.3.3 How to run Distributed Solution of Dinning Philosophers

First, run the philosophers in separated JVM.

Under kxc/m2miapi20020702/lib, run

java edu.rit.m2mi.test.dp.PhilExc <philID>

<Note>The philID must be from 0 to 4 since I already assume only 5 philosophers out there. So you should run 5 instances of PhilExc in 5 separate Java Virtual Machines

Then you will see the “Philosopher i has already sat at the table”. And the philosophers are waiting for his chopsticks to join in and then eat.

Second, run the chopsticks in separated JVM.

Under kxc/m2miapi20020702/lib, run

java edu.rit.m2mi.test.dp.ChopExc <chopID>

<Note>The chopID must be from 0 to 4 since I already assume only 5 chopsiticks out there. So you should run 5 instances of ChopExc in 5 separate Java Virtual Machines

Then user will see the “Chopstick i has already been put on the table”.

The philosophers start to think, pick the chopsticks, eat, and put the chopstick until it is done eating and exit.

5. Software/Hardware Resources

The Software resources used in the project are given below:

Software Tool	Description
M2MP API	Many to Many Protocol library developed in Java by Prof. Bischof and Prof. Kaminsky at RIT
M2MI API	Many to Many Invocation library developed in Java by Prof. Bischof and Prof. Kaminsky at RIT
JDK 1.4	Java Development Kit from Sun for UNIX (Standard Edition)
Putty	SSH tool for the development at home
netwatch, netperf, etc..	Network monitoring tools on Linux for conducting various tests
MS Office 2000	Microsoft office tools on Windows
Star Office	Office tool in UNIX

The hardware requirements for the project are given below:

Hardware Platform	Operating System	Comments
Solaris Ultra 5 at Mobile computing lab at RIT (>= 2)	UNIX	Using in CS lab and final demonstration
Dell Dimension with Intel Pentium-III	Windows 2000, Linux,	Home PC. Development platform

6. References

[1] Hans-Peter Bischof and Alan Kaminsky. “Many-to-Many Invocation: A new framework for building collaborative applications in ad hoc networks.”
CSCW 2002 Workshop on Ad Hoc Communication and Collaboration in Ubiquitous

Computing Environments, New Orleans, Louisiana, USA, November 2002.

[2] Alan Kaminsky and Hans-Peter Bischof. “Many-to-Many Invocation: A new object oriented paradigm for ad hoc collaborative systems.”
17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002), Onward! Track, Seattle, Washington, USA, November 2002

[3] Alan Kaminsky and Hans-Peter Bischof. “Many-to-Many Invocation: A new paradigm for ad hoc collaborative systems.”
Technical Report TR-2002-01, Rochester Institute of Technology, IT Lab, February 6, 2002.

[4] Alan Kaminsky. “Infrastructure for distributed applications in ad hoc networks of small mobile wireless devices.”
Technical Report, Rochester Institute of Technology, IT Lab, May 22, 2001.

[5] R. Stevens, Unix Network Programming, Volume 1, 2nd Edition, Prentice Hall PTR, 1998

[6] C.A.R Hoare, *Communicating Sequential Processes*, Oxford University, 1990

[7] http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/