

CHAPTER 12 – USER’S MANUAL

Using agentMom

1. Introduction – What is agentMom

agentMom is a framework upon which distributed multiagent systems can be developed. It is implemented in Java and provides the basic building blocks for building agents, conversations between agents, and the message that are passed in the conversations. agentMom is capable of using five different type of conversations.

1. Unicast conversation using TCP/IP. This is basically a one-to-one communication.
2. Secured unicast conversation using Secure Socket Layers (SSL) over TCP/IP.
3. Multicast conversation using multicast socket and datagram packet. With multicast conversation, an agent is capable of sending a message to a group of agents that subscribes to the same multicast group.
4. Secured multicast conversation using multicast socket and datagram packet with symmetric key algorithm.
5. Broadcast conversation using datagram socket and datagram packet. With broadcast conversation, an agent is capable of sending a message to all agents within the same local network.

An overview of how conversations in agentMom work is shown below. An agent allows itself to speak with other agents by starting a conversation handler that monitors a local port for messages. All agent communication is performed via conversation classes, which define valid sequences of messages that agents can use to communicate. When one agent wants to communicate with other agents, it starts one of its conversations as a separate Java thread. The conversation then establishes a socket connection with the other agent’s message handler and sends the initial message in the conversation. When the message handler receives a message, it passes the message to the agent’s receive message method that compares the message against its known list of allowable message types to see if it is the start of a valid conversation. If the conversation is valid, the agent starts its side of the appropriate conversation, also as a separate Java thread. From that point on, all communication is controlled by the conversation threads at of each agent. The conversations send/read messages to/from others using built in readMessage and sendMessage methods.

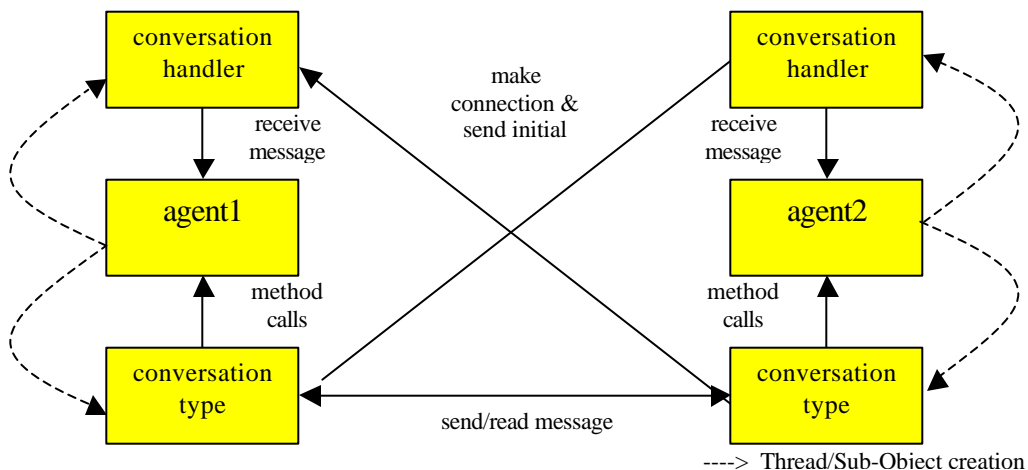


Figure 19 new agentMom

2. How to use agentMom

The ksu.cis.mom package, which makes up the basics of agentMom, is shown below. It consists of nine abstract classes, MomObject, Agent, Component, AgentConversation, Conversation, SecureUnicastConversation, MulticastConversation, SecureMulticastConversation and BroadcastConversation, and seven concrete classes MessageHandler, SecureUnicastHandler, MulticastHandler, SecureMulticastHandler, BroadcastHandler, Message and Sorry. The agentMom and example source codes are included in the CD with this document. Examples can also be found at the end of this chapter.

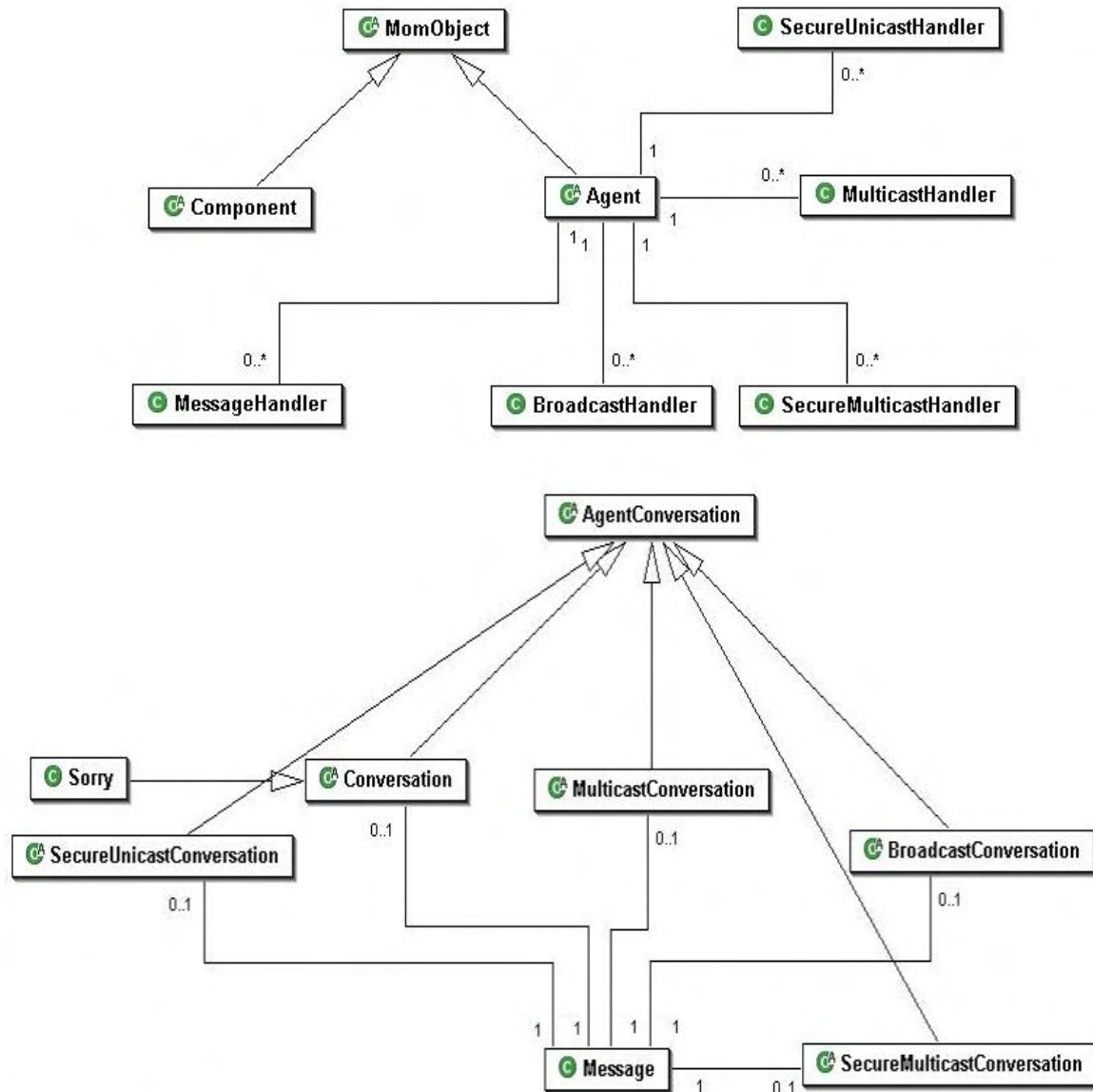
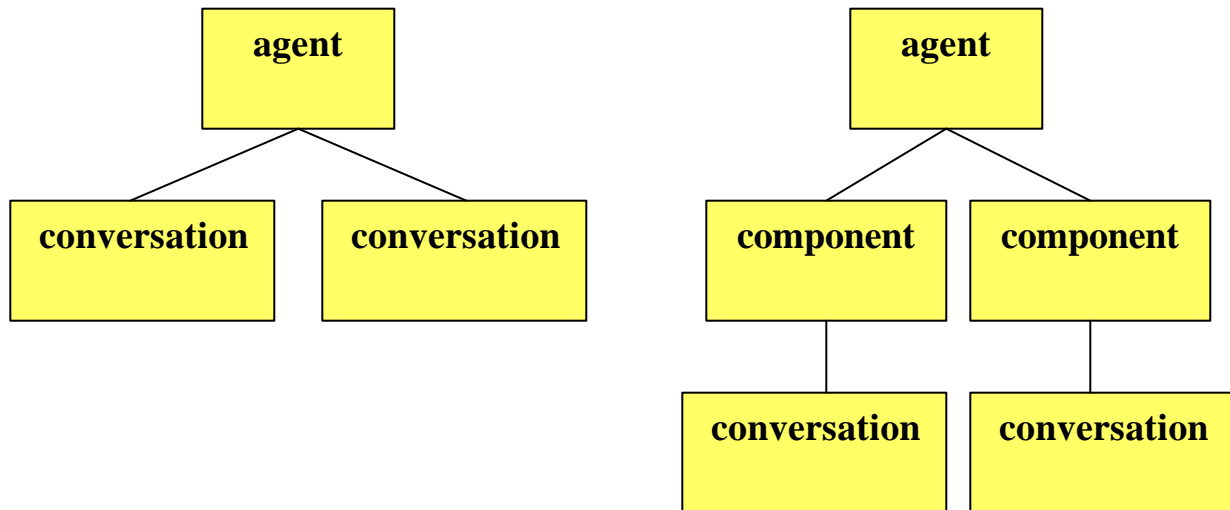


Figure 20 Class diagram: new agentMom

Note that the name Conversation and MessageHandler are not consistent with the other conversations because we want the agentMom to be compatible to the older version. The previous version of agentMom only has unicast conversation type (Conversation class), the multicast, broadcast and secured conversations were added later. Thus, the Conversation class can be thought of as the UnicastConversation class and MessageHandler as the UnicastHandler.



a. Agent directly controls conversations

b. Component controls conversations

Figure 21 new agentMom's architecture

Furthermore, there are two architectures that can be applied to agentMom. The first architecture is shown in Figure a. In the first architecture, agent directly controls the conversations. This architecture is very straightforward since conversations belong to agent. Basically, the conversations are originated from the run method in agent class. In the second architecture as shown in Figure b, an agent consists of one or more components, and the conversations belong to components, not directly to agents. Also, an agent can have multiple components and components can have multiple conversation. The difference from the first architecture is that component is responsible for making conversation with other agents. In the first architecture, agents are directly responsible for controlling the conversation. Having components separately from agent allows developers to map the agent role's tasks to the component. From now, we will refer to the first architecture as agent-based architecture and the second architecture as component-based architecture.

3. Requirements for using agentMom

3.1 agentMom package

3.2 java 1.4 is needed in order to use SecureUnicastHandler, SecureUnicastConversation, SecureMulticastHandler, SecureMulticastConversation class. java 1.2 or higher is required if the security features are not used.

3.3 truststore, keystore and certificate files are needed in order to use SecureUnicastHandler and SecureUnicastConversation class. (details on how to create these files are in section 4 of this chapter).

3.4 Multicast supported router is required in order to use multicast capability in agentMom.

4. agentMom Package

4.1 MomObject Class

MomObject is an abstract class that both Agents and Components inherit from. It allows conversations to work with either agents or components as their parents. Class that inherits from this class must implement the sendInternal method. The MomObject consists of 9 attributes as shown below:

MomObject parent – reference to other MomObject type. It is used by the Component class so that conversation classes can work with either agents or components as their parents.

String name – agent name

int port – unicast port number

int multicast_port [] – array of multicast port number

int broadcast_port – broadcast port number

int secure_unicast_port – secured unicast port number

int secure_multicast_port [] – array of secured multicast port number

InetAddress group [] – array of multicast address

InetAddress broadcast_address – broadcast address

The user does not need to know the details of this class since this class does not provide any service, and it is used within only within the agentMom package. Please refer to the Component Design document for more detail on this class.

4.2 Agent Class

The Agent class is an abstract class that defines the minimum requirements for an agent to use agentMom package. This class inherits the MomObject class. It also implements Runnable interface to be runnable as a separate thread, which requires a run method. Notice that run method is an abstract method, so the sub class of this class must implement this method. If the agent-based architecture is used, the run method is where the agent normally initiates any conversations.

The agent class provides two main constructors. One is for using only unicast conversation. This constructor require two parameters, name of the agent and the port number for unicast communication on which its unicast message handler (MessageHandler class) will listen for incoming messages. The constructor is shown below.

```
public Agent(String name, int port)
```

Another constructor is for using any or all type of conversations. The constructor is shown below.

```
public Agent(String name,  
             int unicast_port,  
             int[] multicast_port,  
             int broadcast_port,  
             int secure_unicast_port,  
             int[] secure_multicast_port)
```

It takes six parameters, String of agent name, integer of port used in each conversation. If any port is assigned to be less than one, then it indicates that the conversation is not going to be used.

Note that the arguments `multicast_port` and `secure_multicast_port` are an array of integer type. It is because we allows agent to subscribe to multiple groups. Thus, one port is used for one group. Assigning the first element less than one indicates that the multicast will not be used.

For all receive methods (`receiveMessage`, `receiveMulticastConversation`, etc), they are used when the handlers (`MessageHandler`, `MulticastHandler`, etc) receive a start of a new conversation from other agents. The handler will call the receive method corresponding to itself. For example, the `MessageHandler` will call the `receiveMessage` method, and `MulticastHandler` will call `receiveMulticastConversation` method, when it receives a start of a new conversation from other agents.

Because the original source of `agentMom` defines `receiveMessage` method to be an abstract method, all sub class of the `Agent` class must implement this method. However, the other types of conversation are optional. Users only need to override the receive methods that they want to use. These receive methods are an empty method in `Agent` class. However, the parameters passing to the methods must be the same as defined in the `Agent` class.

Basically, users can implement by either reading the message from the connection stream (unicast) or reading the message directly (multicast and broadcast), and then determining if it is a valid conversation.

For example, the `receiveMessage` method for unicast conversation can be implement as shown below:

```
public void receiveMessage(
    Socket server,
    ObjectInputStream input,
    ObjectOutputStream output)
{
    int i;
    Message m;
    Server_Register s;
    Thread t;

    try
    {
        m = (Message) input.readObject();
        write("Received message " + m.performative + " from " + m.sender);
        if (m.performative.equals("register"))
        {
            t = new Thread(new Server_Register(server, input, output, this, m));
            t.start(); // start new thread
        }
        else if (m.performative.equals("unregister"))
        {
            t = new Thread(new Server_Unregister(server, input, output, this, m));
            t.start(); // start new thread
        }
        else
        {
            System.out.println(
                "** Invalid Attempt to start new conversation with performative "
                + m.performative
                + " from "
                + m.sender);
        }
    }
}
```

```

    t = new Thread(new Sorry(server, input, output, this, m));
    t.start(); // start new thread
}
}
catch (ClassNotFoundException cnfex)
{
    System.out.println(" ** ClassNotFoundException ");
}
catch (IOException cnfex)
{
    System.out.println(" ** IOException on port (ServerAgent.receiveMessage)");
}
}
}

```

In this case, after the `receiveMessage` method reads the message using the “`m = (Message) input.readObject();`” method call, the performative of the message is checked to see if it is either “register” or “unregister”. In this case, these are the only two performatives the agent can recognize that start conversations in which it can participate. If it is either of these performatives, it creates a new conversation object as a new thread, sends it the initial message, and starts it running using the conversation’s `run` method. If the message received does not start with a recognizable performative, the agent starts the default `Sorry` conversation, which simply sends a sorry message in reply to the performative.

Another example, the `receiveMulticastConversation` method for multicast conversation can be overridden as shown below:

```

public void receiveMulticastConversation(
    MulticastSocket mSocket,
    Message m,
    Vector multicast_queue)
{
    int i;
    Server_Register s;
    Thread t;
    write("Received message " + m.performative + " from " + m.sender);
    if (m.performative.equals("calculate"))
    {
        t = new Thread(new MulticastServer(mSocket, multicast_queue, this,
            m_port[0], m));
        t.start();
    }
}
}

```

In the case of multicast, the message is directly passed to the receive method. The `MulticastSocket mSocket` is used to send out multicast messages and `multicast_queue` is a message queue for the conversation to receive multicast message. Then, the performative of the message is checked to see if it is “calculate”. In this case, this is the only one performative that agent can participate. The other performatives are ignored. If the agent recognizes the performative, it creates a new multicast conversation thread (`MulticastServer`) and starts the thread.

Furthermore, agent may start a new component class instead of conversation class if component-based architecture is used, and the component class will be responsible for starting the conversation.

4.3 AgentConversation Class

This class is an abstract class that all types of conversation inherit from. It is a generalization of all conversations in agentMom package. It defines the minimum requirements for a conversation to be in agentMom package. It also allows user to easily implement a new type of conversation for agentMom package. Sub class of this class should override the sendMessage(), readMessage() and nonblockedReadMessage() method and provides at least two type of constructors, conversation initiator and conversation respondent. Users of agentMom do not need to know any thing about this class since it does not provide any service.

4.4 Component Class

The Component class is an abstract class that defines the minimum requirements for a component. This class inherits from MomObject class. It implements the Runnable interface to be able to run as a thread. It requires only one parameter, MomObject. MomObject is used to be able to refer to the agent that uses this component.

The idea of Component class is to support agent architecture that component performs different tasks. Each component is responsible for particular tasks. Thus, the agents' role's tasks can be mapped to component. Also, components are responsible for starting the conversation with other agents, instead of agent itself. Therefore, agent starts the components, and component starts the conversations.

There are two important attributes in this class, internalMessage and externalMessage. Both attributes are a message queue of type Vector used for internal and external communication. As the name imply, the internalMessage queue is used for communication between components of agent. The externalMessage is used for passing message between component class and conversation, so the message can be deliver to other agents by conversation class. Methods involve with these attributes are checkExternal, checkInternal, enqueueExternal, enqueueInternal and sendInternal. These methods are very straightforward. For example, checkInternal is a method for fetching a message from the internalMessage queue. The sendInternal method allows the component to communicate with other components within an agent. This method simply broadcast message to all active components within the agent.

4.5 MessageHandler Class

The MessageHandler class used to handle unicast connection from other agents. When an agent is created, it needs to create a new message handler thread as shown below:

```
MessageHandler h = new MessageHandler(this.port, this);  
h.start();
```

Two parameters are required to create this class. The two required parameters are the port number and a pointer to the parent agent object. When started, the message handler starts a socket server on the indicated port and waits for a connection from another agent. When a connection is received, the message handler calls the parent agent's receiveMessage method with the connection and the input and output streams. Thus, agent needs to implement the receiveMessage method and starts an appropriate conversation as described in the Agent class.

4.6 SecureUnicastHandler Class

The SecureUnicastHandler is used to handle secured unicast connection from other agents. Basically, it performs the same functionality as the MessageHandler class with security service. The difference is that this class use Secure Socket Layers to handle

secured communication over the TCP/IP connection. An agent can create this class as shown below:

```
SecureUnicastHandler suh = new SecureUnicastHandler(this.port, this); suh.start();
```

To create the SecureUnicastHandler, it requires two parameters, port number and a reference to parent agent object. When started, the message handler starts a SSL socket server on the indicated port and waits for a connection from another agent. When a connection is received, the message handler calls the parent agent's receiveSecureUnicastConversation method with the connection and the input and output streams. Thus, agent needs to implement the receiveSecureUnicastConversation method and starts an appropriate conversation as described in the Agent class. The agent then verifies the received message and starts an appropriate conversation.

SSL uses many cryptography technologies together such as public key, private key, session key, authentication, digital signature, etc. These are transparent to the user of SSL technology. Basically, SSLSocket and SSLServerSocket can be used almost the same way as Socket and ServerSocket class. However, the "keystore", "trustore" and "certificate" must be generated on both sides of communications. Also, each side of communication must have "certificate" of the other side installed. Please note that to be able to use this class, java version 1.4 is required. For example, the tool "keytool", provided in java version 1.4 packages, can be used to generate these requirements. An example on how to create is shown at the end of this section.

Certificates must be created for clients and servers that need to communicate securely using SSL. Java 1.4 uses certificates created using the Java "keytool" shipped with J2SE. I used the following command to create an RSA certificate for the server

```
D:\>keytool -genkey -v -keyalg RSA
Enter keystore password: xxxxxx
What is your first and last name?
 [Unknown]: chairaj mekprasertvit
What is the name of your organizational unit?
 [Unknown]: ksu
What is the name of your organization?
 [Unknown]: cis
What is the name of your City or Locality?
 [Unknown]: manhattan
What is the name of your State or Province?
 [Unknown]: ks
What is the two-letter country code for this unit?
 [Unknown]: us
Is CN=chairaj mekprasertvit, OU=ksu, O=cis, L=manhattan, ST=ks, C=us correct?
 [no]: y
Generating 1,024 bit RSA key pair and self-signed certificate (MD5WithRSA)
 for: CN=chairaj mekprasertvit, OU=ksu, O=cis, L=manhattan, ST=ks, C=us
Enter key password for <mykey>
 (RETURN if same as keystore password):
[Saving C:\j2sdk-1.4.2\keystore]
D:\>
```

Then, we need to export the self-signed certificate.

```
C:\j2sdk-1.4.2>keytool -export -keystore .keystore -file certificate
Enter keystore password: xxxxxx
Certificate stored in file <certificate>
```


Note that this is a self-signed certificate. Alternatively, we can generate Certificate Signing Request (CSR) with `-certreq` and send that to a Certificate Authority (CA) for signing, but this is only experimenting software.

Finally, we import the certificate into a new truststore.

```
C:\j2sdk-1.4.2>keytool -import -file certificate -keystore truststore
Enter keystore password: xxxxxx
Owner: CN=chairoj mekprasertvit, OU=ksu, O=cis, L=manhattan, ST=ks, C=us
Issuer: CN=chairoj mekprasertvit, OU=ksu, O=cis, L=manhattan, ST=ks, C=us
Serial number: 402cb4ae
Valid from: Fri Feb 03 05:27:42 CST 2004 until: Thu May 03 06:27:42 CDT 2004
Certificate fingerprints:
    MD5: E7:87:63:4E:2F:04:FA:3A:15:92:31:70:4F:B0:1F:C4
    SHA1: 94:17:E2:0D:00:DE:09:A7:DA:6A:3E:68:83:FC:39:68:D7:02:25:6E
Trust this certificate? [no]: yes
Certificate was added to keystore
```

Notice that the certificate is valid only a period of time (3 months). Now, we can then run the class using SSL as shown below:

```
java -Djavax.net.ssl.keyStore=.keystore -Djavax.net.ssl.keyStorePassword=xxxxxx -
Djavax.net.ssl.trustStore=truststore -Djavax.net.ssl.trustStorePassword=xxxxxx agentTest
```

4.7 MulticastHandler Class

MulticastHandler is responsible for initializing and starting multicast socket, including joining/leaving multicast group. In generally, it handles multicast connection with other agents. It uses the MulticastSocket class to subscribe to multicast group. When an agent is created, it needs to create a new multicast handler thread as shown below:

```
MessageHandler mh = new MulticastHandler(this, port, time-to-live, group); mh.start();
or
MessageHandler mh = new MulticastHandler(this, port, time-to-live, group, packetSize);
mh.start();
```

There are two constructors for this class. The difference is that the second constructor allows specifying the buffer size of received multicast message. The first constructor has a default of 1024 bytes of buffer size. Be aware that the buffer size of received message must be equal or greater to the sent message. Both constructors have the same first four parameters. The four parameters are a pointer to the parent agent object, port number, time-to-live of the multicast message and the multicast group address.

Note that the multicast address is actually a class D IP addresses that is in the range 224.0.0.0 to 239.255.255.255. And the time-to-live of the message is in the range of 0-255. The table below roughly shows the scope for a given range of time-to-live.

TTL	Scope
0-32	Institution
33-64	Region
65-128	Continent
129-255	Unrestricted (global)

However, this is not always true because the network may refuse to forward the message, and message may be dropped by the network router because the multicast socket use an unreliable protocol.

Basically, we use the multicast handler the same way as we use the MessageHandler class. When MulticastHandler class is created, it starts the multicast socket on the indicated port and joins to specified multicast group. Then, it automatically sends a multicast message to the group indicating that this agent has join the group. When a MulticastHandler receives the message indicating the join, it calls the receiveMulticastJoin method in agent class. Agent has to override this method to make use of it. For example, agent can keep track of the other agents who join the group after itself. The same thing applies to the receiveMulticastLeave method in the Agent class. When the sendLeave method in the MulticastHandler class is executed. It automatically sends message leave to the group and then unsubscribes from multicast group.

After this class is initialized, it waits for a message from other agents. When multicast handler receives a message, it will check whether the message is a start of new conversation/join/leave/conversation message. If it is a start of conversation, the MulticastHandler simply calls the parent agent's receiveMulticastcastConversation method with the multicast socket, received message and multicast message queue. The agent then verifies the received message and starts an appropriate conversation. If the received message is for any multicast conversation class, it adds the message to the multicast message queue. Then the multicast conversation class can get the message from this queue later. If the join or leave message is received, it calls the parent agent's receiveMulticastJoin or receiveMulticastLeave as described above. Moreover, the agent may leave the group by calling the method sendLeave in the MulticastHandler class.

Note that to properly using multicast protocol the network router does need to support it; otherwise, the message may be delivered as broadcast message or not delivered at all. Also, the operating systems must be configured to accept multicast message.

4.8 SecureMulticastHandler Class

The SecureMulticastHandler is used to handle secured multicast connection from other agents. Basically, it performs the same functionality as the MulticastHandler class with security service. The difference is that this class use symmetric key algorithm to perform message encryption and decryption. An agent can create this class as shown below:

```
SecureMulticastHandler smh = new SecureMessageHandler(this, port, time-to-live, group, key, algorithm);  
smh.start();
```

or

```
SecureMulticastHandler smh = new SecureMessageHandler(this, port, time-to-live, group, packetSize, key,  
algorithm);  
smh.start();
```

Detail of this class is almost the same as the MulticastHandler. However, there are two more parameters required in the constructors, key and algorithm. The key parameter is the Key class in the java.Security package. It stores the private key used for encrypting and decrypting message. Notice that we use the same key to encryption and encryption. The algorithm parameter is the name of algorithm used to generate the key and how to perform encryption and decryption. Thus, all agents in this multicast group need to have

the same key and algorithm. There are many ways to distribute the key. For example, agents can request the private key from a trust server using secured unicast conversation. An agent can generate a key as shown below:

```
algorithm = "DES";  
key = KeyGenerator.getInstance(algorithm).generateKey();
```

In this case, we use the “DES” algorithm, and then use the KeyGenerator class to generate a random key based on the algorithm used. There is no restriction on how an agent obtains the key and algorithm as long as it is the symmetric key algorithm. Message encryption and decryption are automatically performed by agentMom package; message is encrypted before sending and decrypted after receiving, so agent only provides the key and algorithm, and make sure that all agents in the group have the same key and algorithm.

4.9 BroadcastHandler

BroadcastHandler is responsible for initializing and starting datagram socket for broadcast conversation. It uses the DatagramSocket class to send and receive broadcast conversation in form of datagram packet.

When an agent is created, it needs to create a new broadcast handler thread to be able to receive a start of broadcast conversation from the other agents. When BroadcastHandler is created, it starts the DatagramSocket class for broadcast conversation. Below is how an agent can start the BroadcastHandler class.

```
BroadcastHandler bh = new BroadcastHandler(this, port, address);  
bh.start();
```

or

```
BroadcastHandler bh = new BroadcastHandler(this, port, address, packetSize);  
bh.start();
```

There are two constructors for this class. The difference is that the second constructor allows specifying the buffer size of received multicast message. The first constructor has a default of 1024 bytes of buffer size. Be aware that the buffer size of received message must be equal or greater to the sent message. Both constructors have the same first three parameters. The three parameters are a pointer to the parent agent object, port number and the broadcast address.

In general, broadcast address is in the form "xxx.xxx.xxx.255" for local broadcast. However, many networks do not allow the use of broadcast, and they may have a specific address for broadcasting. Users have to check the availability of this address.

After this class is initialized, it waits for a message from other agents. When broadcast handler receives a message, it will check whether the message is a start of new conversation. If it is a start of conversation, the BroadcastHandler simply calls the parent agent's receiveBroadcastConversation method with the datagram socket, received message and broadcast message queue. The agent then verifies the received message and starts an appropriate conversation. If the received message is for any broadcast conversation class, it adds the message to the broadcast message queue. Then the broadcast conversation class can get the message from this queue later.

4.10 Conversation

The Conversation class is an abstract class that actually carries out the message passing between agents using unicast communication. There are three methods in the Conversation class, `readMessage`, `nonblockedReadMessage` and `sendMessage` that actually pass the messages back and forth over the socket connection. There are really two types of conversation classes that can be derived from the Conversation class, one for the conversation initiator and one for the conversation respondent. The basic difference lies in which constructor is used and the details in the abstract run method, which must be implemented in the concrete class derived from the Conversation class.

An example of an initiator conversation class is the `Client_Register` class in Register-Deregister Example. To initiate this conversation, the `ClientAgent` creates a new `Client_Register` object (as a separate thread) using the `Client_Register` constructor. This constructor does not need to send a socket, input stream, or output stream (see second Conversation constructor in Register-Deregister Example) since, as an initiator, the conversation creates a new socket and opens an input and output stream with a second agent's message handler. When the `ClientAgent` starts the `Client_Register` conversation class, the `Client_Register`'s run method is started. This method controls the conversation. It creates a new connection using the following commands.

```
connection = new Socket(connectionHost, connectionPort);
output = new ObjectOutputStream(connection.getOutputStream());
output.flush();
input = new ObjectInputStream(connection.getInputStream());
```

After the connection is made, the method enters a while loop that iterates until the conversation is completed. Inside the while loop is a simple switch statement that has a case for each possible state of the conversation. Actually the state in the run method may or may not correspond one-to-one with the states of the conversation as defined in a MaSE conversation diagram. Actually, it is possible to have one state for each state in the diagram plus a state for each transition out of a state. In a simple conversation such as the `Client_Register` conversation, this could be modeled as a simple sequence of statements; however, in the general case, conversations may have loops and many branches out of a single state, thus the switch within a loop provides the most general mechanism for modeling conversation states. The loop and switch statement are shown below.

```
while (notDone)
{
    switch (state)
    {
        case 0 :
            m.performative = "register";
            m.content = service;
            sendMessage(m, output);
            state = 1;
            break;
        case 1 :
            m = readMessage(input);
            if (m.performative.equals("reply"))
                notDone = false;
            else
                parent.write("** ERROR - did not get reply back **");
            break;
```

```
}  
}
```

In the code above, the state variable starts at state zero. In state 0, the message performative is set to register and the message content is set to a string sent to the conversation by the ClientAgent when it was initialized. Actually the content of a message can take any Java object type, but it must implement the interface Serializable. After sending the message, the state variable is set to 1 and the break statement takes us out of the switch statement. Since notDone is still true, we stay in the loop, this time entering the case 1 option of the switch statement. At this point, we wait at the readMessage call until a message comes in from the other agent. In this case, we use the readMessage method that is a blocking read (wait until message arrives). There is also a nonblockedReadMessage that allows the read message to timeout thus allowing the conversation to check to see if it has a message without waiting forever. The default value of timeout is 100 milliseconds.

Then, if the message is what we expect (a reply performative), we process it; otherwise we print an error message. In this case, we do nothing with the reply and simply set the notDone variable to false so that we will exit the while loop.

After exiting the conversation, we close the connection with the other agent using the sequence of close statements shown below.

```
input.close();  
output.close();  
connection.close();
```

4.11 MulticastConversation

The MulticastConversation class is an abstract class that actually carries out the message passing between agents in the group using multicast communication. There are two methods in the MulticastConversation class, readMessage, nonblockedReadMessage and sendMessage that actually pass the messages back and forth over the socket connection. There are really two types of conversation classes that can be derived from the MulticastConversation class, one for the conversation initiator and one for the conversation respondent. The basic difference lies in which constructor is used and the details in the abstract run method, which must be implemented in the concrete class derived from the MulticastConversation class.

Detail on how to create a multicast conversation initiator and respondent is the same as unicast conversation such as using the while loop and switch statement in the run method. However, the connection of multicast conversation is performed differently. There is no need to create a connection socket for sending a message since the MulticastConversation takes care of this. MulticastSocket uses connectionless protocol, so there is no input and out stream as in the unicast conversation.

In order to initiate multicast conversation, an agent need to start the multicast handler first because the handler is responsible for joining the multicast group, and it is also responsible for receiving all multicast messages and place in the message queue. The conversation can fetch the message through the message queue by using its conversation name as explained in the next paragraph. When the multicast conversation is created, the constructor needs to call the super class constructor of the initiator side as shown below:

```
super(agent, group, port, messageQueue);
```

The agent parameter is a pointer to the parent agent. The group, port and messageQueue parameter are the multicast address, port for multicast and multicast message queue, respectively. Note that the multicast message queue must be a pointer to the same queue as the one the agent passes to the MulticastHandler's constructor. An example of multicast conversation initiator's run method can be created as shown below:

```
public void run()
{
    Message m = new Message();
    int state = 0;
    boolean notDone = true;
    try
    {
        while (notDone)
        {
            switch (state)
            {
                case 0 :
                    m.performative = "calculate";
                    m.content = command1;
                    startConversation(m, conversation_name, "MulticastServer");
                    state = 1;
                    break;
                case 1 :
                    m = readMessage("multicast");
                    if (m.performative.equals("reply"))
                    {
                        parent.write((String)m.content + " from " + m.sender + "\n");
                        m.performative = "good bye";
                        m.content = command2;
                        sendMessage(m);
                        notDone = false;
                    }
                    else
                        parent.write("*** ERROR - did not get reply back ***");
            }
        }
    } catch (IOException e)
    {
        parent.write("*** IO Exception in multicast conversation.");
    }
}
```

Notice that the startConversation method is used on case 0. This is how multicast conversation can be started. Any subsequence will use sendMessage instead as shown in case 1. Also, there are three required parameters for sending a message. The first one is the Message object. The second one is the name of this conversation (the name of the conversation must be unique from the other multicast conversation because the destination conversation can then reply to the correct conversation. The last parameter is the name of the destination conversation. The reason for using the name of conversation is that an agent may have multiple concurrent threads of multicast conversation that use the same multicast group, hence sharing the same message queue. The originating and destinating conversation are needed. Moreover, multicast conversation must also pass the name to the readMessage method to fetch the message destined for this conversation. In

this case, the multicast is the name of this conversation and MulticastServer is the name of the destination conversation.

As same as any conversation class in agentMom, there're two methods for fetching message from the queue. One is a blocked read (readMessage), and another one is nonblocked read (nonblockedReadMessage). However, the nonblocked read in multicast allows specifying the timeout in milliseconds.

Be aware that the conversation needs to know how many times it has to perform read message because this is one-to-many conversation. The conversation may receive more than one reply message, so the read message also need to perform more than one times. Thus, it may get an unexpected message if the read message does not perform properly.

4.12 SecureMulticastConversation

The SecureMulticastConversation class is an abstract class that actually carries out the message passing between agents in the group using secured multicast communication. SecureMulticastConversation has the same detail as the MulticastConversation class. The agent has to start the SecureMulticastHandler first before starting a secured multicast conversation. Also, the private key and the name of the must be the same as the SecureMulticastHandler class uses. Below is how the conversation can call the super class constructor of the initiator side:

```
super(agent, group, port, messageQueue, key, algorithm);
```

The details on how to send and receive message can be performed the same way as in the multicast conversation. Message encryption and decryption is done automatically.

4.13 SecureUnicastConversation

The SecureUnicastConversation class is an abstract class that actually carries out the message passing to agents in the same group using secured multicast communication. Because this class relies on the SSL technology as same as the SecureUnicastHandler class, it has the same requirements as the SecureUnicastHandler class. There is only one different between using this class and the Conversation class, the socket. In the Conversation class, the Socket class is used to make connection to the other agents. In this class, the SSLSocket is used instead. The code below shows how to create the SSLSocket class to make a connection with other agents:

```
SSLConnectionFactory sslFact =  
(SSLConnectionFactory) SSLConnectionFactory.getDefault();  
connection = (SSLSocket) sslFact.createSocket(connectionHost, connectionPort);  
output = new ObjectOutputStream(connection.getOutputStream());  
input = new ObjectInputStream(connection.getInputStream());
```

After we initialize the connection, output, input, we can then use these variables as same as we do in the Conversation class. To exit the conversation, we also do the same way as in the Conversation class as shown below.

```
input.close();  
output.close();  
connection.close();
```

4.14 BroadcastConversation

The BroadcastConversation class is an abstract class that actually carries out the message passing to all agents under the same local network. The BroadcastConversation class uses the DatagramSocket to send and receive message. In fact, the DatagramSocket

class is a super class of the MulticastSocket class. Detail on using broadcast conversation is the same as in multicast conversation. As same as multicast conversation, the BroadcastHandler is needed to start first. Below is how the conversation can call the super class constructor of the initiator side:

```
super(agent, broadcastAddress, port, messageQueue);
```

An example of the while loop in broadcast conversation initiator's run method can be created as shown below:

```
while (notDone)
{
    switch (state)
    {
        case 0 :
            m.performative = "calculate";
            m.content = command1;
            startConversation(m, conversation_name, "BroadcastServer");
            state = 1;
            break;
        case 1 :
            m = readMessage("broadcast");
            if (m.performative.equals("reply"))
            {
                parent.write((String) m.content + " from " + m.sender + "\n");
                m.performative = "good bye";
                m.content = command2;
                sendMessage(m);
                notDone = false;
            }
            else
                parent.write("** ERROR - did not get reply back **");
    }
}
```

Please be aware that sending many broadcast messages can easily flood the network. Also, message can be lost or undelivered easily using this type of conversation. Below are some of the possible causes:

- 1) The network do not allow broadcast message.
- 2) The broadcast address is incorrect.
- 3) The router drops message, especially during busy traffic.
- 4) The packet

4.15 Message Class

Message class defines the field used in the message passed back and forth between agents. Note that these fields are derived from the fields in a KQML message, and some of them are automatically filled by the sendMessage method in each type of conversation classes. In agentMom, there is no restriction in using these fields. For more information about KQML please refer to <http://www.fipa.org>. It is fairly straightforward and consists of the following attributes.

Object content = null

Support for complex object that encapsulates a number of attribute types. These complex objects can be used to pass multiple parameters in a single message. Note that in order to pass an object across a socket connection, it must implement the interface Serializable.

String force = null

Specify whether the sender will never deny the meaning of the performative.

String host = null

Host name that this message is sent to.

String inreplyto = null

The expected label in a reply.

String language = null

Name of representation language of the content.

String ontology = null

Name of the ontology used in the content

String performative = null

Describe the action that the message intends. The user can define any performative they feel are necessary.

int port = 0

Port number used for the message.

String receiver = null

Name of the receiver

String replywith = null

Whether the sender expects a reply, and if so, a label for the reply.

String sender = null

Name of the sender (agent's name).

When a conversation calls the `sendMessage` method, it automatically fill the sender, host, and port fields using the parent agent's name and port attributes and automatically gets the host name from the system. The `replywith` and `inreplyto` fields are also automatically fill if the `sendMessage` is called from `MulticastConversation`, `SecureMulticastConversation` and `BroadcastConversation`. The other fields of interest in an `agentMom` message are the performative and content fields. The performative field describes the action that the message intends and is used in the agent and conversation classes to

- (1) Determine the type of conversation being requested and
- (2) To control the execution of a conversation in the run method.

Because `agentMom` does not have any specific performative types, users can define any performative they feel are necessary. The content of an `agentMom` message is also very general. Basically, the message passes any valid Java object type. This can be as simple as a string, or a more complex object that encapsulates a number of attribute types. These complex objects can be used to pass multiple parameters in a single message as shown in the class below.

```
public class ComplexObject implements Serializable
{
    String agent;
    String host;
    int port;
    String service;
    public ComplexObject(String a, String h, int p, String ser)
    {
        agent = a;
        host = h;
        port = p;
        service = ser;
    }
}
```

This class encapsulates four parameters (three strings and an integer) that can be assigned to message content field. Note that in order to pass an object across a socket connection, it must implement the interface `Serializable`.

Note that in order to pass an object across a socket connection, it must implement the interface Serializable.

4.16 Sorry Class

The **Sorry** class defines a general-purpose conversation to reply "Sorry" to any unknown/unexpected type of unicast conversation. It is a simply concrete class of Conversation class, so there is no implementation required in this class. Automatically, performative field is set to "sorry" and content field is set to "unknown conversation request" when using this class. The example on how to use this class is shown above when we described the Agent class.

5. Step By Step Construction

To build an application using the agentMom framework, you need to perform the following:

- 1) Get a copy of agentMom classes (included in the CD with this document).
- 2) Define your agent classes and conversations according the MaSE (Multiagent Systems Engineering) methodology. An environment, agentTool, is available to help you with this. Please note that the current agentTool only supports code generation for unicast conversation.
- 3) For each agent class in your system, extend the agentMom Agent class.
 - a) Define any necessary receive message methods for each type of conversation to handle all conversations for which the agent is a respondent.
 - b) For each action defined in the set of conversations in which this agent may participate, define a method in the agent class. This will be your interface to the conversation.
 - c) Implement the run method as the main procedure of the method. If your agent initiates any conversations, this could be where they will originate.
 - d) If you want to run your agent as a stand-alone application, create a main method to initialize the agent running.
- 4) For each conversation in your system design, create two conversation classes, in initiator and a respondent class.
 - a) For each initiator class, define a constructor that includes, as parameters, the first message sent.
 - b) For each respondent class, define a constructor that includes, as a parameter, the message read by the parent receive conversation method before the conversation thread was started.
 - c) Implement the run method
 - i) Define a state variable initialized to state 0.
 - ii) If it is an initiator conversation, create a connection with the agent of interest.
 - iii) Create a switch statement within a while loop where each case in the switch statement corresponds to a state or a transition. Ensure at least one of the states exits the while loop.
 - iv) Close the connection.
- 5) Create any supporting classes for things such as
 - a) Objects that combine multiple parameters into a single object.
 - b) System setup/testing routines.
 - c) Components of intelligent agents.

6. Examples

Register-Deregister Example ClientAgent

```
package ksu.cis.momtest;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import afit.mom.*;
/**
 * This type was created in VisualAge.
 */
public class ClientAgent extends Agent implements ActionListener {
    private TextArea theText;
    private Panel textPanel;
    private TextField Tname;
    public String serverHost;
    public int serverPort;

    /**
     * ClientAgent constructor comment.
     */
    public ClientAgent(String agentName, int agentPort,
        String sHost, int sPort) {
        super(agentName, agentPort);

        this.serverHost = sHost; // the Host to connect to
        this.serverPort = sPort; // the Port to connect to

        /* server initialization */
        MessageHandler h = new MessageHandler(this.port, this);
        h.start();

        /* window initialization */
        setSize(400, 300); // set default size & height of new window

        addWindowListener(new WindowDestroyer());
        setTitle(this.name); // name the window
        setBackground(Color.red);
        setLayout(new BorderLayout());

        // text panel
        textPanel = new Panel();
        theText = new TextArea(12, 50);
        theText.setBackground(Color.white);
        theText.setForeground(Color.black);
        textPanel.add(theText);
        add(textPanel, "Center");

        // top label
        Panel namePanel = new Panel();
        namePanel.setForeground(Color.white);
        Label nameLabel = new Label("Enter text of your choice: ");
        namePanel.add(nameLabel, "Center");
        add(namePanel, "North");
    }

    /**
     * This method was created in VisualAge.
     */
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Exit")) {

```

```

        System.exit(0);
    } else
        theText.setText("Error in memo interface!");
    textPanel.repaint();
}

/**
 * This method was created in VisualAge.
 */
public String get_service() {
    if (this.name.equals("Client-1"))
        return "Laundry";
    if (this.name.equals("Client-2"))
        return "Dry Cleaning";
    if (this.name.equals("Client-3"))
        return "Drill Press Operator";
    else
        return "Unknown";
}

/**
 * This method was created in VisualAge.
 */
public static void main(String[] args) {
    String serverHost;
    int serverPort = 5000;

    try {
        serverHost = InetAddress.getLocalHost().getHostName();
        ClientAgent s =
            new ClientAgent("Client", 3000, serverHost, serverPort);
        s.run();
    } catch (UnknownHostException e) {
        System.out.println(" ** Host Exception when starting ClientAgent");
    }
}

/**
 * receiveMessage method comment.
 */
public void receiveMessage(Socket server, ObjectInputStream input,
                           ObjectOutputStream output) {

    int i;
    Message m;
    Server_Register s;
    Thread t;

    /* define all possible conversations receivable here */
    try {
        m = (Message) input.readObject();
        System.out.println(
            " ** Invalid Attempt to start new conversation with performative "
            + m.performative + " from " + m.sender);
        t = new Thread(new Sorry(server, input, output, this, m));
        t.start(); // start new thread
    } catch (ClassNotFoundException cnfex) {
        System.out.println(" ** ClassNotFoundException ");
    } catch (IOException cnfex) {
        System.out.println(" ** IOException on port
(ServerAgent.receiveMessage)");
    }
}
}

```

```

/**
 * This method was created in VisualAge.
 */
public void run() {
    Thread init;
    theText.append("Starting Client \n");
    setVisible(true);

    init = new Thread(new Client_Register(this, serverHost, serverPort,
                                         get_service())); // start conversation
    init.start();

    int j;
    for (int i = 0; i < 10000000; i++)
        j = i - 1;
    init = new Thread(new Client_Unregister(this, serverHost, serverPort,
                                             get_service())); // start conversation
    init.start();
}

/**
 * This method was created in VisualAge.
 */
protected void write(String s) {
    this.theText.append(s + "\n");
}
}

```

ServerAgent

```
package ksu.cis.momtest;

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import afit.mom.*;

/**
 * This type was created in VisualAge.
 */
public class ServerAgent extends Agent implements ActionListener {
    private TextArea theText;
    private Panel textPanel;
    private TextField Tname;

    private Vector registrants = new Vector(25);

    /**
     * ServerAgent constructor comment.
     */
    public ServerAgent(String n, int p) {
        super(n, p);
        MessageHandler handler;          /* server initialization */
        handler = new MessageHandler(port, this);
        handler.start();

        /* window initialization */
        setSize(500, 300); // set default size & height of new window

        addWindowListener(new WindowDestroyer()); // create listener
        setTitle("Server"); // name the window
        setBackground(Color.blue);
        setLayout(new BorderLayout());

        // text panel
        textPanel = new Panel();
        theText = new TextArea(12, 60);
        theText.setBackground(Color.white);
        theText.setForeground(Color.black);
        textPanel.add(theText);
        add(textPanel, "Center");

        // top label
        Panel namePanel = new Panel();
        namePanel.setForeground(Color.white);
        Label nameLabel = new Label("Enter text of your choice: ");
        namePanel.add(nameLabel, "Center");
        add(namePanel, "North");

        setVisible(true);
    }

    /**
     * This method was created in VisualAge.
     */
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Exit")) {
            System.exit(0);
        } else
            theText.setText("Error in memo interface!");
    }
}
```

```

        textPanel.repaint();
    }

    /**
     * This method was created in VisualAge.
     */
    protected void add_service(String agent, String service, String host, int port)
    {
        this.registrants.addElement(new Registration(agent, host, port, service));
    }

    /**
     * This method was created in VisualAge.
     */
    public static void main(String[] args) {
        ServerAgent s = new ServerAgent("Server", 5000);
        s.run();
    }

    /**
     * This method was created in VisualAge.
     */
    protected void printRegistrants() {
        java.util.Enumeration enum = this.registrants.elements();
        Registration v;
        enum = this.registrants.elements();
        while (enum.hasMoreElements()) {
            v = (Registration) enum.nextElement();
            this.theText.append("< " + v.agent + ", " + v.service + ", " + v.host +
", "
                                + v.port + "> \n");
        }
    }

    /**
     * receiveMessage method comment.
     */
    public void receiveMessage(Socket server, ObjectInputStream input,
                                ObjectOutputStream output) {

        int i;
        Message m;
        Server_Register s;
        Thread t;

        /* define all possible conversations receivable here */
        try {
            m = (Message) input.readObject();
            this.theText.append("Received message " + m.performative
                                + " from "+m.sender+"\n");
            if (m.performative.equals("register")) {
                t = new Thread(new Server_Register(server, input, output, this, m));
                t.start(); // start new thread
            } else
                if (m.performative.equals("unregister")) {
                    t = new Thread(new Server_Unregister(server, input, output,
                                                            this, m));

                    t.start(); // start new thread
                } else {
                    System.out.println(
                        " ** Invalid Attempt to start new conversation with
performative "
                        + m.performative + " from " + m.sender);
                }
        }
    }

```

```

        t = new Thread(new Sorry(server, input, output, this, m));
        t.start(); // start new thread
    }
} catch (ClassNotFoundException cnfex) {
    System.out.println(" ** ClassNotFoundException ");
} catch (IOException cnfex) {
    System.out.println(" ** IOException on port
(ServerAgent.receiveMessage)");
}
}

/**
 * This method was created in VisualAge.
 */
protected void remove_service(String agent, String service,
                               String host, int port) {
    java.util.Enumeration enum = this.registrants.elements();
    Registration v;
    while (enum.hasMoreElements()) {
        v = (Registration) enum.nextElement();
        if (v.agent.equals(agent) && v.service.equals(service)
            && v.host.equals(host) && v.port == port)
            this.registrants.removeElement(v);
    }
}

/**
 * This method was created in VisualAge.
 */
public void run() {
    theText.append("Starting Server \n");
    setVisible(true);
}

/**
 * This method was created in VisualAge.
 */
protected void write(String s) {
    this.theText.append(s + "\n");
}
}

```


Client_Register

```
package ksu.cis.momtest;

/**
 * This type was created in VisualAge.
 */
import java.net.*;
import java.io.*;
import afit.mom.*;
public class Client_Register extends Conversation {
    ClientAgent parent; // override parent
    String service;

public Client_Register(ClientAgent a, String hostName,
    int portNum, String theService) {
    super(a, hostName, portNum);
    parent = a;
    service = theService;
}

public void run() {
    Message m = new Message();
    int state = 0;
    boolean notDone = true;
    parent.write("Starting Client_Register conversation.");

    //set up conversation
    try {
        connection = new Socket(connectionHost, connectionPort);
        output = new ObjectOutputStream(connection.getOutputStream());
        output.flush();
        input = new ObjectInputStream(connection.getInputStream());
        while (notDone) {
            switch (state) {
                case 0 :
                    m.performative = "register";
                    m.content = service;
                    sendMessage(m, output);
                    state = 1;
                    break;
                case 1 :
                    m = readMessage(input);
                    if (m.performative.equals("reply"))
                        notDone = false;
                    else
                        parent.write("*** ERROR - did not get reply back ***");
                    break;
            }
        }
        input.close();
        output.close();
        connection.close();
    } catch (UnknownHostException e) {
        parent.write("*** Unknown Host exception in Client_Register.");
    } catch (EOFException oef) {
        parent.write("*** Server terminated connection in Client_Register.");
    } catch (IOException e) {
        parent.write("*** IO Exception in Client_Register.");
    }
}
}
```

Client_Unregister

```
package ksu.cis.momtest;

/**
 * This type was created in VisualAge.
 */
import java.net.*;
import java.io.*;
import afit.mom.*;
public class Client_Unregister extends Conversation {
    ClientAgent parent; // override parent
    String service;

public Client_Unregister(ClientAgent a, String hostName,
    int portNum, String theService) {
    super(a, hostName, portNum);
    parent = a;
    service = theService;
}

public void run() {
    Message m = new Message();
    int state = 0;
    boolean notDone = true;
    parent.write("Starting Client_Unregister conversation.");
    //set up conversation
    try {
        connection = new Socket(connectionHost, connectionPort);
        output = new ObjectOutputStream(connection.getOutputStream());
        output.flush();
        input = new ObjectInputStream(connection.getInputStream());
        while (notDone) {
            switch (state) {
                case 0 :
                    m.setPerformative("unregister");
                    m.content = service;
                    sendMessage(m, output);
                    state = 1;
                    break;
                case 1 :
                    m = readMessage(input);
                    if (m.performative.equals("reply"))
                        notDone = false;
                    else
                        parent.write("*** ERROR - did not get reply back ** \n");
                    break;
            }
        }
        input.close();
        output.close();
        connection.close();
    } catch (UnknownHostException e) {
        parent.write("*** Unknown Host exception in Client_Unregister.");
    } catch (EOFException oef) {
        parent.write("*** Server terminated connection in Client_Unregister.");
    } catch (IOException e) {
        parent.write("*** IO Exception in Client_Unregister.");
        e.printStackTrace();
    }
}
}
```

Server_Register

```
package ksu.cis.momtest;

/**
 * This type was created in VisualAge.
 */
import java.net.*;
import java.io.*;
import afit.mom.*;
public class Server_Register extends Conversation {
    ServerAgent parent; // override parent

public Server_Register(Socket s, ObjectInputStream i, ObjectOutputStream o,
ServerAgent a, Message m1) {
    super(s, i, o, a, m1);
    parent = a;
}

public void run() {
    int state = 0;
    boolean notDone = true;
    parent.write("Got >>" + m.getPerformative() + " - " + m.getContent()
        + " from " + m.getSender());
    //set up conversation
    try {
        while (notDone) {
            switch (state) {
                case 0 :
                    parent.add_service(m.sender, (String) m.content, m.host,
m.port);

                    parent.printRegistrants();
                    state = 1;
                    break;
                case 1 :
                    m.setPerformative("reply");
                    m.setContent("OK");
                    sendMessage(m, output);
                    notDone = false;
            }
        }
        input.close();
        output.close();
        connection.close();
    } catch (UnknownHostException e) {
        parent.write("*** Unknown Host exception.");
    } catch (EOFException oef) {
        parent.write("*** Server terminated connection.");
    } catch (IOException e) {
        parent.write("*** IO Exception. (Server_Register)");
        e.printStackTrace();
    }
}
}
```

Server_Unregister

```
package ksu.cis.momtest;

/**
 * This type was created in VisualAge.
 */
import java.net.*;
import java.io.*;
import afit.mom.*;
public class Server_Unregister extends Conversation {
    ServerAgent parent; // override parent

public Server_Unregister(Socket s, ObjectInputStream i, ObjectOutputStream o,
ServerAgent a, Message m1) {
    super(s, i, o, a, m1);
    parent = a;
}

public void run() {
    int state = 0;
    boolean notDone = true;
    parent.write("Got >>" + m.getPerformative() + " - " + m.getContent()
        + " from " + m.getSender());

    //set up conversation
    try {
        while (notDone) {
            switch (state) {
                case 0 :
                    state = 1;
                    break;
                case 1 :
                    parent.remove_service(m.sender, (String) m.content, m.host,
m.port);

                    parent.printRegistrants();
                    state = 2;
                    break;
                case 2 :
                    m.setPerformative("reply");
                    m.setContent("OK");
                    sendMessage(m, output);
                    notDone = false;
                    break;
            }
        }
        input.close();
        output.close();
        connection.close();
    } catch (UnknownHostException e) {
        parent.write("*** Unknown Host exception.");
    } catch (EOFException oef) {
        parent.write("*** Server terminated connection.");
    } catch (IOException e) {
        parent.write("*** IO Exception.");
        e.printStackTrace();
    }
}
}
```

Registration

```
package ksu.cis.momtest;

/**
 * This type was created in VisualAge.
 */
public class Registration {
    String agent;
    String host;
    int port;
    String service;

    /**
     * Registration constructor comment.
     */
    public Registration(String a, String h, int p, String ser) {
        agent = a;
        host = h;
        port = p;
        service = ser;
    }
}
```

Tester

```
package ksu.cis.momtest;

/**
 * This type was created in VisualAge.
 */
import java.net.*;
public class Tester {
/**
 * Tester constructor comment.
 */
public Tester() {
    super();
}
/**
 * Starts the application.
 */
public static void main(java.lang.String[] args) {
    int NUMCLIENTS = 3;
    int cPort = 3000; // starting port address
    int sPort = 5000;
    String host;
    Thread server;
    Thread clients[] = new Thread[NUMCLIENTS];
    //

    try {
        host = InetAddress.getLocalHost().getHostName();

        System.out.println("Starting Server Agent on host " + host);
        server = new Thread(new ServerAgent("Server", sPort), "Server");
        server.start();

        for (int i = 0; i < clients.length; i++) {
            System.out.println("Starting Client Agent" + (i + 1));
            clients[i] = new Thread(new ClientAgent("Client-" + (i + 1),
                cPort + i, host, sPort), "Client-" + (i + 1));
            clients[i].start();
        }
        while (true);
    } catch (UnknownHostException e) {
        System.out.println(" ** Unknown Host Exception when starting
ClientAgent");
    }
}
}
```

WindowDestroyer

```
package ksu.cis.momtest;

import java.awt.*;
import java.awt.event.*;
/**
 * This type was created in VisualAge.
 */
public class WindowDestroyer extends WindowAdapter {
/**
 * This method was created in VisualAge.
 */
public void windowClosing(WindowEvent e) {
    System.exit(0);
}
}
```

File Sharing Example

The File Sharing example consists of client agents who want to request files from the server and the server agents who are responsible for sending files. The scenario starts by each start joining the same multicast group. The client agents can request a file by sending a unicast request to any server agent in multicast group. If that server agent does not have the file, it then sends multicast to the group to send the file instead. In this scenario, we assume that each server contains files different from the others, so there is only one who actually sends the file. Also, the client agent always request existing files because the client agent do not get any answer if the file is not found in the group.

ClientAgent

```
package ksu.cis.example.FileSharing;

import java.io.*;
import java.util.*;
import ksu.cis.mom.*;
import java.net.*;

/**
 * Client agent class. This agent request a file from server agent.
 * @author Chairroj Mekprasertvit
 *
 */
public class ClientAgent extends Agent
{
    private int state = 0;
    private boolean alive = true;
    public static final int unicastPort = 3000;
    private ClientAgentGUI gui;

    public ClientAgent(ClientAgentGUI gui)
    {
        super("ClientAgent", unicastPort);
        this.gui = gui;
        this.port = unicastPort;
        MessageHandler h = new MessageHandler(unicastPort, this);
        h.start();
    }
    /**
     * The performative that this agent can participate is "found file".
     * Other performatives will be ignored.
     *
     */
    public void receiveMessage(
        Socket server,
        ObjectInputStream input,
        ObjectOutputStream output)
    {
        Thread t;
        Message m;
        try
        {
            {
                m = (Message) input.readObject();
                if(m.performative.equals("found file"))
                {
                    t = new Thread(new ClientReceiveFile(server, input, output, this, m,
gui));
                    t.start();
                }
            }
        } catch(ClassNotFoundException cnfex)
        {
            System.out.println("*** Class not found exception in " + this.name);
        }
        catch(IOException ioex)
        {
            System.out.println("*** IOException in " + this.name);
        }
    }

    public void sendRequest(String req)
```

```

    {
        StringTokenizer token = new StringTokenizer(req);
        Thread init;
        init = new Thread(
            new ClientRequestFile(this, token.nextToken(), token.nextToken(),
gui));
        init.start();
    }

/**
 * run method for client. It put agent to sleep for 1000 millisc.
 * @see java.lang.Runnable#run()
 */
public void run()
{
    while(alive)
    {
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException iex)
        {
            System.out.println("*** Thread Interrupted in " + this.name);
        }
    }
}
}
}

```

ClientAgentGUI

```
package ksu.cis.example.FileSharing;
import java.awt.*;
import java.awt.event.*;

/**
 * GUI for Client agent. It has a text field to input what file to request,
 * and from whom eg. "192.150.121.19 readme.txt"
 * @author Chairroj Mekprasertvit
 *
 */
public class ClientAgentGUI extends Frame
{
    private TextField field;
    private TextArea display;
    private ClientAgent agent;

    /**
     * Constrcutor for agent_interface.
     * @param name
     * @param broadcastAddress
     */
    public ClientAgentGUI()
    {
        super("Please Wait...");
        setLayout (new BorderLayout());
        field = new TextField();
        display = new TextArea();
        add(display, "Center");
        display.setBackground(Color.white);
        display.setForeground(Color.black);
        add(field, "South");

        setSize(300, 200);
        setVisible(true);
        show();

        agent = new ClientAgent(this);
        Thread t = new Thread(agent);
        t.start();

        //create field for outgoing secured multicast message.
        field.setEnabled(false);
        field.addActionListener
        (
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    setDisplay(e.getActionCommand());
                    agent.sendRequest(e.getActionCommand());
                }
            }
        );
    }

    public void startUI()
    {
        field.setEnabled(true);
        setTitle("Demo III: " + agent.name);
        setDisplay("*** Client started.");
    }
}
```

```

/**
 * <p>Enable the TextField. Allows user to type secured multicast message.
 */
public void setField()
{
    field.setEnabled(true);
}

public void setDisplay(String m)
{
    display.append(m + "\n");
}

/**
 * main method to start the client.
 * @param args
 */
public static void main(String[] args)
{
    ClientAgentGUI gui = new ClientAgentGUI();
    gui.addWindowListener
    (
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }
    );
    gui.startUI();
} // end of main method
}

```

ClientReceiveFile

```
package ksu.cis.example.FileSharing;

import ksu.cis.mom.*;
import java.net.*;
import java.io.*;

/**
 * Unicast conversation respondent. It send an acknowledgement message when the
 * file is received.
 * @author Chairroj Mekprasertvit
 */
public class ClientReceiveFile extends Conversation
{
    private ClientAgent parent;
    private ClientAgentGUI gui;
    private File file;
    public ClientReceiveFile(
        Socket s,
        ObjectInputStream i,
        ObjectOutputStream o,
        ClientAgent a,
        Message m,
        ClientAgentGUI gui)
    {
        super(s, i, o, a, m);
        parent = a;
        this.gui = gui;
    }

    public void run()
    {
        int state = 0;
        boolean notDone = true;
        try
        {
            while(notDone)
            {
                switch(state)
                {
                    case 0:
                        InfoObject info = (InfoObject) m.content;
                        try
                        {
                            FileOutputStream out = new FileOutputStream(info.getFileName());
                            out.write(info.getFile());
                            out.close();
                        }
                        catch(IOException ioex)
                        {
                            parent.write("*** Error writing file." + ioex.toString());
                        }
                        state = 1;
                    case 1:
                        Message reply = new Message();
                        reply.setPerformative("ack");
                        sendMessage(reply, output);
                        gui.setDisplay("*** Agent successfully get the file");
                        notDone = false;
                        break;
                }
            }
        }
    }
}
```

```
    input.close();
    output.close();
    connection.close();
}
catch (UnknownHostException e)
{
    parent.write("** Unknown Host exception in Client_Register.");
}
catch (EOFException oef)
{
    parent.write("** Server terminated connection in Client_Register.");
}
catch (IOException e)
{
    parent.write("** IO Exception in Client_Register.");
}
} // end run method
} // end class
```

ClientRequestFile

```
package ksu.cis.example.FileSharing;

import ksu.cis.mom.*;
import java.net.*;
import java.io.*;

/**
 * Unicast conversation initiator. It basically send a request for file and
 * waiting for the accept message.
 * @author Chairroj Mekprasertvit
 */
public class ClientRequestFile extends Conversation
{
    private ClientAgent parent;
    private ClientAgentGUI gui;
    private String fileName;
    private String serverName;
    public ClientRequestFile(
        ClientAgent agent,
        String server,
        String req,
        ClientAgentGUI gui)
    {
        super(agent, server, agent.port);
        parent = agent;
        fileName = req;
        serverName = server;
        this.gui = gui;
    }

    /**
     *
     * @see java.lang Runnable#run()
     */
    public void run()
    {
        Message m = new Message();
        int state = 0;
        boolean notDone = true;
        try
        {
            connection = new Socket(connectionHost, connectionPort);
            output = new ObjectOutputStream(connection.getOutputStream());
            output.flush();
            input = new ObjectInputStream(connection.getInputStream());
            while (notDone)
            {
                switch (state)
                {
                    case 0 :
                        m.performative = "find file";
                        m.content = fileName;
                        this.sendMessage(m, output);
                        state = 1;
                        break;
                    case 1 :
                        m = this.readMessage(input);
                        if(m.performative.equals("reply"))
                        {
                            gui.setDisplay("The request is " + m.content.toString());
                        }
                        else

```

```

        parent.write("*** ERROR - did not get reply back ***");
        notDone = false;
        break;
    }
}
input.close();
output.close();
connection.close();
}
catch (UnknownHostException e)
{
    parent.write("*** Unknown Host exception in unicast. " + e.toString());
}
catch (EOFException oef)
{
    parent.write(
        "*** Server terminated connection in unicast. " + oef.toString());
}
catch (IOException e)
{
    parent.write("*** IO Exception in unicast. " + e.toString());
}
}
}
}

```


InfoObject

```
package ksu.cis.example.FileSharing;
import java.io.*;
/**
 * InfoObject type for the content field of Message class. It contains the
 * requesting file's name, the file and the requesting agent IP address.
 * @author Chairroj Mekprasertvit
 */
public class InfoObject implements Serializable
{
    private String fileName = null;
    private byte file[];
    private String client = null;
    public InfoObject(String f, String c)
    {
        fileName = f;
        client = c;
    }
    public InfoObject(String f, byte b[])
    {
        fileName = f;
        file = b;
    }
    public String getFileName()
    {
        return fileName;
    }
    public String getClient()
    {
        return client;
    }
    public byte[] getFile()
    {
        return file;
    }
}
```

ServerAgent

```
package ksu.cis.example.FileSharing;
import java.io.*;
import java.net.*;
import java.util.*;
import ksu.cis.mom.*;

/**
 * Server agent that has two services: get the file request and send file.
 * @author Chairroj Mekprasertvit
 */
public class ServerAgent extends Agent
{
    public static final int multicastPort[] = {3002, 3003};
    public static final int secureMulticastPort[] = {0,0};
    public static final int unicastPort = 3000;

    public ServerAgent()
    {
        super("ServerAgent", unicastPort, multicastPort, 0, 0,
secureMulticastPort);
        try
        {
            super.group = new InetAddress[1];
            super.group[0] = InetAddress.getByName("239.192.100.100");
        }
        catch (Throwable e)
        {
            write("Invalid Internet Address");
        }

        MessageHandler h = new MessageHandler(unicastPort, this);
        h.start();
        MulticastHandler mh = new MulticastHandler(this, multicastPort[0], 1,
group[0]);
        mh.start();
    }
    /**
     * receiveMessage method. When receive a unicast connection from other agent,
     * it find the file in itself. If the file not found, it send multicast
     * message to the group to send the file instead.
     */
    public void receiveMessage(
        Socket server,
        ObjectInputStream input,
        ObjectOutputStream output)
    {
        Thread t;
        Message m;
        try
        {
            m = (Message) input.readObject();
            // send accept request
            if(m.performative.equals("find file"))
            {
                ServerRespondRequest srr = new ServerRespondRequest(
                    server, input, output, this, m);
                t = new Thread(srr);
                t.start();

                //wait until conversation is done
                while(srr.getDone())
                {

```

```

        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException iex)
        {
            write("** ServerRespondRequest InterruptedException.");
        }
    }

    // check if this agent has the file, then send back
    this.write("The file is " + m.content);
    boolean found = this.findFile((String) m.content);
    if(found)
    {
        ServerReturnFile srf = new ServerReturnFile(
            this, server.getInetAddress().getHostName(), (String)m.content);
        t = new Thread(srf);
        t.start();
    }
    // if not found, then ask the group who has it
    else
    {
        String client = server.getInetAddress().getHostName();
        ServerMulticast sm = new ServerMulticast(
            this, multicastPort[0], group[0], (String)m.getContent(), client);
        t = new Thread(sm);
        t.start();

        while(sm.getDone())
        {
            try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException iex)
            {
                write("** ServerRespondRequest InterruptedException.");
            }
        }
    }
} // end if
else
    write("** Unrecognized Performative - IGNORED ");
}
catch(ClassNotFoundException cnfex)
{
    System.out.println("** Class not found exception in " + this.name);
}
catch(IOException ioex)
{
    System.out.println("** IOException in " + this.name);
}
}
/**
 * find the file when receive a multicast conversation
 */
public void receiveMulticastConversation(
    MulticastSocket mSocket,
    Message m,
    Vector multicast_queue)
{
    Thread t;

```

```

if (m.performative.equals("find file"))
{
    // check if this agent has the file, then send back
    InfoObject info = (InfoObject) m.content;
    boolean found = this.findFile(info.getFileName());
    if (found)
    {
        ServerReturnFile srf =
            new ServerReturnFile(this, info.getClient(), info.getFileName());
        t = new Thread(srf);
        t.start();
    }
    else
        write("** File not found on this agent.");
}
else
    write("** Unrecognized Performative - IGNORED ");
}

public boolean findFile(String fileName)
{
    String user_dir = System.getProperty ("user.dir");

    File file = new File(user_dir);
    String all[] = file.list();
    boolean found = false;

    for(int i=0; i<all.length; i++)
    {
        if(fileName.equals(all[i]))
        {
            found = true;
            write("** The file is found.");
            break;
        }
        else
            found = false;
    }
    return found;
}

public void run()
{
    boolean alive = true;
    while(alive)
    {
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException iex)
        {
            System.out.println("** Thread Interrupted in " + this.name);
        }
    }
}

public static void main(String[] args)
{
    ServerAgent server = new ServerAgent();
    Thread t = new Thread(server);
    t.start();
}
}

```

ServerMulticast

```
package ksu.cis.example.FileSharing;
import ksu.cis.mom.*;
import java.net.*;
import java.io.*;
/**
 * Multicast conversation initiator. It basically sends the multicast request
 * to the group asking to send the file to the client.
 */
public class ServerMulticast extends MulticastConversation
{
    private String fileName;
    private String conversation_name;
    private ServerAgent parent;
    private String client;
    private boolean isDone = false;
    public ServerMulticast(
        ServerAgent a,
        int port,
        InetAddress group,
        String req,
        String client)
    {
        super(a, group, port, a.multicast_queue);
        parent = a;
        this.fileName = req;
        this.conversation_name = "ServerMulticast";
        this.client = client;
    }
    public boolean getDone()
    {
        return isDone;
    }
    public void run()
    {
        Message m = new Message();
        int state = 0;
        boolean notDone = true;
        try
        {
            while (notDone)
            {
                switch (state)
                {
                    case 0 :
                        m.performative = "find file";
                        InfoObject info = new InfoObject(fileName, client);
                        m.content = info;
                        startConversation(m, conversation_name, "Server");
                        state = 1;
                        break;
                    case 1 :
                        notDone = true;
                        isDone = true;
                }
            }
        }
        catch (IOException e)
        {
            parent.write("*** IO Exception in multicast conversation.");
        }
    }
}
```

ServerRespondRequest

```
package ksu.cis.example.FileSharing;
import java.net.*;
import ksu.cis.mom.*;
import java.io.*;

/**
 * Unicast conversation respondent. It basically accepts the request,
 * and send the accept request message
 * @author Chairoj Mekprasertvit
 */
public class ServerRespondRequest extends Conversation
{
    private ServerAgent parent;
    private boolean isDone = false;

    public ServerRespondRequest(
        Socket s,
        ObjectInputStream i,
        ObjectOutputStream o,
        ServerAgent a,
        Message m)
    {
        super(s, i, o, a, m);
        parent = a;
    }

    public boolean getDone()
    {
        return isDone;
    }

    public void run()
    {
        int state = 0;
        boolean notDone = true;
        int result = 0;
        try
        {
            while (notDone)
            {
                switch (state)
                {
                    case 0 :
                        Message reply = new Message();
                        reply.setPerformative("reply");
                        reply.setContent("accepted");
                        this.sendMessage(reply, output);
                        state = 1;
                        break;
                    case 1 :
                        notDone = false;
                        break;
                }
            }
        }
        input.close();
    }
}
```

```
        output.close();
        connection.close();
        isDone = true;
    }
    catch (UnknownHostException e)
    {
        parent.write("** Unknown Host exception in Client_Register.");
    }
    catch (EOFException oef)
    {
        parent.write("** Server terminated connection in
Client_Register.");
    }
    catch (IOException e)
    {
        parent.write("** IO Exception in Client_Register.");
    }
}
}
```

ServerReturnFile

```
package ksu.cis.example.FileSharing;

import ksu.cis.mom.*;
import java.net.*;
import java.io.*;

/**
 * Unicast conversation initiator. It basically sends the file back
 * to the requesting client using unicast conversation.
 * @author Chairroj Mekprasertvit
 */
public class ServerReturnFile extends Conversation
{
    private ServerAgent parent;
    private String fileName;
    private String client;

    public ServerReturnFile(
        ServerAgent agent,
        String client,
        String file)
    {
        super(agent, client, agent.port);
        parent = agent;
        fileName = file;
        this.client = client;
        parent.write("client is " + client);
    }

    public void run()
    {
        Message m = new Message();
        int state = 0;
        boolean notDone = true;
        try
        {
            parent.write("ConnectionHost is " + connectionHost);
            connection = new Socket(connectionHost, connectionPort);
            output = new ObjectOutputStream(connection.getOutputStream());
            output.flush();
            input = new ObjectInputStream(connection.getInputStream());
            while (notDone)
            {
                switch (state)
                {
                    case 0 :
                        m.performative = "found file";
                        ByteArrayOutputStream bs = new ByteArrayOutputStream();
                        FileInputStream in = new FileInputStream(fileName);
                        int c;
                        while ((c = in.read()) != -1)
                            bs.write(c);
                        byte b[] = bs.toByteArray();
                        InfoObject info = new InfoObject(fileName, b);
                        m.content = info;
                }
            }
        }
    }
}
```



```

        this.sendMessage(m, output);
        state = 1;
        break;
    case 1 :
        m = this.readMessage(input);
        if (m.performative.equals("ack"))
        {
            parent.write("*** Client has received the file.");
        }
        else
            parent.write("*** ERROR - did not get reply back ***");
        notDone = false;
        break;
    }
}
input.close();
output.close();
connection.close();
}
catch (UnknownHostException e)
{
    parent.write("*** Unknown Host exception in unicast. " +
e.toString());
}
catch (EOFException oef)
{
    parent.write(
        "*** Server terminated connection in unicast. " +
oef.toString());
}
catch (IOException e)
{
    parent.write("*** IO Exception in unicast. " + e.toString());
}
}
}
}

```

Key Distribution Example

The Key Distribution example consists of distributor agents who want to distribute the private key used for encrypting and decrypting multicast message and the client agents who are waiting to form the multicast group using secured multicast conversation. The scenario starts by distributor agent sends broadcast message asking for IP address. The client agents then send back their IP using broadcast conversation. Distributor agent waits for a specific time period before starts sending the private key using secured unicast conversation to the agents that reply back within the time period. After the key is received, the client agents form the multicast group and communicate using secured multicast conversation based on the received private key.

ClientAgent

```
package ksu.cis.example.KeyDistribution;
import java.io.*;
import java.net.*;
import java.security.*;
import javax.net.ssl.*;
import ksu.cis.mom.*;
import java.util.*;
public class ClientAgent extends Agent
{
    public static final int unicastPort = 3000;
    public static final int secureUnicastPort = 3001;
    public static final int multicastPort[] = {3002, 3003};
    public static final int broadcastPort = 3004;
    public static final int secureMulticastPort[] = {3005, 3006};
    private ClientAgentGUI gui;
    private Key key;
    private String algorithm;
    private SecureMulticastHandler smh;
    public ClientAgent(String name, String broadcastAddress, ClientAgentGUI gui)
    {
        super(
            name,
            unicastPort,
            multicastPort,
            broadcastPort,
            secureUnicastPort,
            secureMulticastPort);
        this.gui = gui;
        try
        {
            super.group = new InetAddress[5];
            super.group[0] = InetAddress.getByName("239.255.10.11");
            super.group[1] = InetAddress.getByName("239.255.10.12");
            super.broadcast_address = InetAddress.getByName(broadcastAddress);
        }
        catch (Throwable e)
        {
            write("Invalid Internet Address");
        }
        BroadcastHandler bh =
            new BroadcastHandler(this, broadcastPort, broadcast_address);
        bh.start();

        SecureUnicastHandler suh =
            new SecureUnicastHandler(secureUnicastPort, this);
        suh.start();
    }
    public void receiveMessage(
        Socket server,
        ObjectInputStream input,
        ObjectOutputStream output)
    {
    }
    public void receiveBroadcastConversation(
        DatagramSocket bSocket,
        Message m,
        Vector broadcast_queue)
    {
        Thread t;
        if (m.performative.equals("register"))
        {

```

```

        t = new Thread(new ClientReturnRegister(bSocket, broadcast_queue, this,
broadcastPort, m, gui));
        t.start();
    }
}
public void receiveSecureMulticastConversation(
    MulticastSocket mSocket,
    Message m,
    Vector multicast_queue,
    Key k,
    String algorithm)
{
    System.out.println("receive multicast " + (String)m.content);
    gui.setDisplay((String)m.content);
}
public void receiveSecureUnicastConversation(
    SSLSocket server,
    ObjectInputStream input,
    ObjectOutputStream output)
{
    Message m;
    Thread t;
    try
    {
        m = (Message) input.readObject();
        if (m.performative.equals("GetKey"))
        {
            this.key = (Key) m.content;
            this.algorithm = key.getAlgorithm();
            gui.setDisplay("*** Agent has received the secret key.");
            gui.setDisplay("*** Key using: " + key.getAlgorithm());
            t = new Thread(new ClientGetKey(server, input, output, this, m));
            t.start();
            smh = new SecureMulticastHandler(
                this, secureMulticastPort[1], 1, group[1], key, "DES");
            smh.start();
        }
    }
    catch (ClassNotFoundException cnfex)
    {
        write("*** Error class not found in " + this.name);
    }
    catch (IOException ex)
    {
        write("*** Error read unicast input in "+ this.name);
    }
}
public void sendSecuredMulticast(String m)
{
    Thread init;
    init = new Thread(new SecureMulticastSend(
        this, secureMulticastPort[1], group[1], m, gui, key, algorithm));
    init.start();
}
public void run()
{
    boolean alive = true;
    gui.setDisplay("*** Agent Run.");
    while(alive)
    {}
}
}
}

```

ClientAgentGUI

```
package ksu.cis.example.KeyDistribution;
import java.awt.*;
import java.awt.event.*;

public class ClientAgentGUI extends Frame
{
    private TextField field;
    private TextArea display;
    private ClientAgent agent;
    public ClientAgentGUI(String name, String broadcastAddress)
    {
        super("Please Wait...");
        setLayout (new BorderLayout());
        field = new TextField();
        display = new TextArea();
        add(display, "Center");
        display.setBackground(Color.white);
        display.setForeground(Color.black);
        add(field, "South");
        setSize(300, 200);
        setVisible(true);
        show();
        agent = new ClientAgent(name, broadcastAddress, this);
        Thread t = new Thread(agent);
        t.start();
        //create field for outgoing secured multicast message.
        field.setEnabled(false);
        field.addActionListener
        (
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    setDisplay(e.getActionCommand());
                    agent.sendSecuredMulticast(e.getActionCommand());
                }
            }
        );
    }
    public void startUI()
    {
        field.setEnabled(true);
        setTitle("Demo III: " + agent.name);
        setDisplay("** Client started.");
    }
    public void setField()
    {
        field.setEnabled(true);
    }

    public void setDisplay(String m)
    {
        display.append(m + "\n");
    }

    public static void main(String[] args)
    {
        if(args.length != 2)
        {
            System.out.println("Usage: java ClientAgentGUI <name> <broadcast
address>");
            System.exit(1);
        }
    }
}
```

```
}
String name = args[0];
String broadcastAddress = args[1];
ClientAgentGUI gui = new ClientAgentGUI(name, broadcastAddress);
gui.addWindowListener
(
    new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }
);
gui.startUI();
} // end of main method
} //end of class agent_interface
```

ClientGetKey

```
package ksu.cis.example.KeyDistribution;
import ksu.cis.mom.*;
import java.io.*;
import javax.net.ssl.*;

public class ClientGetKey extends SecureUicastConversation
{
    public ClientAgent parent;
    public ClientGetKey(
        SSLSocket s,
        ObjectInputStream i,
        ObjectOutputStream o,
        ClientAgent a,
        Message m)
    {
        super(s, i, o, a, m);
        parent = a;
    }

    public void run()
    {
        int state = 0;
        boolean notDone = true;
        try
        {
            while (notDone)
            {
                switch (state)
                {
                    case 0 :
                        Message reply = new Message();
                        reply.setPerformative("ack");
                        sendMessage(reply, output);

                        notDone = false;
                        state = 1;
                        break;
                }
            }
            input.close();
            output.close();
            connection.close();
        }
        catch (EOFException eof)
        {
            parent.write("*** Server terminated connection in SSL.");
        }
        catch (IOException e)
        {
            parent.write("*** IO Exception in SSL.");
        }
    }
}
```

ClientReturnRegister

```
package ksu.cis.example.KeyDistribution;

import ksu.cis.mom.*;
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * @author Chairoj Mekprasertvit
 *
 */
public class ClientReturnRegister extends BroadcastConversation
{
    private ClientAgent parent;
    private ClientAgentGUI gui;
    private String conversation_name = "ClientReturnRegister";
    public ClientReturnRegister(
        DatagramSocket dSocket,
        Vector queue,
        ClientAgent a,
        int b_port,
        Message m,
        ClientAgentGUI gui)
    {
        super(a, a.broadcast_address, b_port, queue, m);
        this.gui = gui;
        this.dSocket = dSocket;
        parent = a;
    }

    /**
     * <p>Run method. Broadcast IP address of itself when received broadcast
     request
     * to register.
     * @see java.lang.Runnable#run()
     */
    public void run()
    {
        boolean notDone = true;
        InetAddress ip = null;
        int state = 0;
        int result = 0;

        while(notDone)
        {
            switch(state)
            {
                case 0:
                    try
                    {
                        ip = InetAddress.getLocalHost();
                        System.out.println(ip.getHostAddress());
                        state = 1;
                    }
                    catch(UnknownHostException e)
                    {
                        System.out.println("** Unknowhost Exception in
ClientReturnRegister.");
                    }
                    break;
                case 1:
                    Message register = new Message();

```



```

register.setPerformative("register");
register.setContent((Object)ip);
try
{
    sendMessage(register, conversation_name,
"DistributorRequestRegister");
    gui.setDisplay("*** Agent sends register IP: " +
ip.getHostAddress());
    System.out.println("*** Agent sends register IP: " + ip.toString());
}
catch (IOException ioex)
{
    parent.write(ioex.toString());
}
notDone = false;
break;
} // end switch
} // end while
} // end run method
} // end class

```

DistributorAgent

```
package ksu.cis.example.KeyDistribution;

import java.io.*;
import java.net.*;
import java.security.*;
import javax.net.ssl.*;
import javax.crypto.*;
import ksu.cis.mom.*;
import java.util.*;

public class DistributorAgent extends Agent
{
    public static final int unicastPort = 3000;
    public static final int secureUnicastPort = 3001;
    public static final int multicastPort[] = {3002, 3003};
    public static final int broadcastPort = 3004;
    public static final int secureMulticastPort[] = {3005, 3006};
    private Vector clientList;
    private DistributorAgentGUI gui;
    private Key key;
    private String algorithm;
    private boolean alive;
    private boolean finishRegister;

    public DistributorAgent(
        String name,
        String broadcastAddress,
        DistributorAgentGUI gui)
    {
        super(
            name,
            unicastPort,
            multicastPort,
            broadcastPort,
            secureUnicastPort,
            secureMulticastPort);
        this.gui = gui;
        this.alive = true;
        this.clientList = new Vector(10);
        this.finishRegister = false;

        try
        {
            super.group = new InetAddress[5];
            super.group[0] = InetAddress.getByName("239.255.10.11");
            super.broadcast_address = InetAddress.getByName(broadcastAddress);
            algorithm = "DES";
            key = KeyGenerator.getInstance(algorithm).generateKey();
        }
        catch (Throwable e)
        {
            write("Invalid Internet Address");
        }

        MessageHandler h = new MessageHandler(unicastPort, this);
        h.start();

        MulticastHandler mh =
            new MulticastHandler(this, multicastPort[0], 1, group[0]);
        mh.start();
    }
}
```

```

BroadcastHandler bh =
    new BroadcastHandler(this, broadcastPort, broadcast_address);
bh.start();
}

public void receiveMessage(
    Socket server,
    ObjectInputStream input,
    ObjectOutputStream output)
{
}

public void addToList(InetAddress ip)
{
    this.clientList.addElement(ip);
}

public void setFinishRegister(boolean bool)
{
    this.finishRegister = bool;
}
/**
 *
 * @see java.lang.Runnable#run()
 */
public void run()
{
    int state = 0;
    Thread init = new Thread();
    while(alive)
    {
        switch(state)
        {
            case 0:
                gui.setDisplay("*** Distributor sends Broadcast");
                init = new Thread(new DistributorRequestRegister(this, broadcastPort,
broadcast_address, gui));
                init.start();
                state = 1;
                break;
            case 1:
                try
                {
                    Thread.sleep(10000);
                    if(finishRegister == true)
                        state = 2;
                    else
                        state = 1;
                }
                catch(InterruptedException e)
                {
                    System.out.println(e.toString());
                }
                break;
            case 2:
                for(int i=0; i<clientList.size(); i++)
                {
                    gui.setDisplay("*** Distributor sends secured unicast to all agents
in the list");
                    init = new Thread(new DistributorSendKey(
this, (InetAddress)clientList.elementAt(i), secure_unicast_port,
key, gui));

```


DistributorAgentGUI

```
package ksu.cis.example.KeyDistribution;
import java.awt.*;
import java.awt.event.*;

public class DistributorAgentGUI extends Frame
{
    private TextArea display;
    private DistributorAgent agent;
    public DistributorAgentGUI(String name, String broadcastAddress)
    {
        super("Please Wait...");
        setLayout (new BorderLayout());
        display = new TextArea();
        add(display, "Center");
        display.setBackground(Color.white);
        display.setForeground(Color.black);

        setSize(300, 200);

        setVisible(true);
        show();

        agent = new DistributorAgent(name, broadcastAddress, this);
        Thread t = new Thread(agent);
        t.start();
    }

    public void startUI()
    {
        setTitle("Demo III: " + agent.name);
    }

    public void setDisplay(String m)
    {
        display.append(m + "\n");
        System.out.println(m);
    }

    public static void main(String[] args)
    {
        if(args.length != 2)
        {
            System.out.println("Usage: java DistributorAgentGUI <name> <broadcast
address>");
            System.exit(1);
        }
        String name = args[0];
        String broadcastAddress = args[1];
        DistributorAgentGUI gui = new DistributorAgentGUI(name, broadcastAddress);
        gui.addWindowListener
        (
            new WindowAdapter()
            {
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            }
        );
        gui.startUI();
    } // end of main method
}
```

DistributorRequestRegister

```
package ksu.cis.example.KeyDistribution;
import ksu.cis.mom.*;
import java.net.*;
import java.io.*;
public class DistributorRequestRegister extends BroadcastConversation
{
    private DistributorAgent parent;
    private String conversation_name;
    private DistributorAgentGUI gui;

    public DistributorRequestRegister(
        DistributorAgent agent,
        int b_port,
        InetAddress broadcast_address,
        DistributorAgentGUI gui)
    {
        super(agent, broadcast_address, b_port, agent.broadcast_queue);
        this.parent = agent;
        this.conversation_name = "DistributorRequestRegister";
        this.gui = gui;
    }
    public void run()
    {
        Message m = new Message();
        int state = 0;
        boolean notDone = true;
        int count = 0;
        try
        {
            while(notDone)
            {
                switch(state)
                {
                    case 0:
                        m.performative = "register";
                        startConversation(m, conversation_name, "ClientReturnRegister");
                        state = 1;
                        break;
                    case 1:
                        try
                        {
                            gui.setDisplay("** Agent will wait for reply for 5000
millisec.");
                            Thread.sleep(5000);
                            gui.setDisplay("** 5000 milisec is over.");
                        }
                        catch(InterruptedException e)
                        {
                            System.out.println(e.toString());
                        }
                        state = 3;
                        break;
                    case 3:

                        do
                        {
                            m = nonblockedReadMessage("DistributorRequestRegister", 1000);
                            if(m.content.equals("timeout"))
                            {
                                parent.write("TIMEOUT BROADCAST");
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

        if(m.performative.equals("register"))
        {
            parent.addToList((InetAddress) m.content);
            this.gui.setDisplay("IP: " + m.content.toString() + " is being
registered.");
        }
    }
    while(!m.content.equals("timeout"));
    notDone = false;
    parent.setFinishRegister(true);
    break;
}
}
}
}
catch(IOException e)
{
    parent.write("** IO Exception in DistributorRegisterRequest
conversation.");
}
}
}
}
}

```

DistributorSendKey

```
package ksu.cis.example.KeyDistribution;

import ksu.cis.mom.*;
import javax.net.ssl.*;
import java.net.*;
import java.security.*;
import java.io.*;

public class DistributorSendKey extends SecureUnicastConversation
{
    private DistributorAgent parent;
    private Key key;
    private DistributorAgentGUI gui;
    public DistributorSendKey(
        DistributorAgent agent,
        InetAddress ip,
        int su_port,
        Key key,
        DistributorAgentGUI gui)
    {
        super(agent, ip.getHostAddress(), su_port);
        parent = agent;
        this.key = key;
        this.gui = gui;
    }

    public void run()
    {
        Message m = new Message();
        int state = 0;
        boolean notDone = true;
        try
        {
            SSLSocketFactory sslFact =
                (SSLSocketFactory) SSLSocketFactory.getDefault();
            connection = (SSLSocket) sslFact.createSocket(connectionHost,
connectionPort);
            output = new ObjectOutputStream(connection.getOutputStream());
            input = new ObjectInputStream(connection.getInputStream());
            output.flush();
            while(notDone)
            {
                switch(state)
                {
                    case 0:
                        m.performative = "GetKey";
                        m.content = key;
                        sendMessage(m, output);
                        gui.setDisplay(
                            "Distributor sent key to " +
                                connection.getInetAddress().getHostName());
                        state = 1;
                        break;
                    case 1:
                        m = readMessage(input);
                        if(m.performative.equals("ack"))
                        {
                            gui.setDisplay(m.sender + " has received the key.");
                            notDone = false;
                        }
                        else
                            parent.write("** Error - did not get reply back.");
                }
            }
        }
    }
}
```



```
        break;
    }
}
input.close();
output.close();
connection.close();
}
catch (EOFException oef)
{
    parent.write("** Server terminated connection in DistributorSendKey.");
}
catch (IOException e)
{
    parent.write("** IO Exception in DistributorSendKey.");
}
} // end run method
} // end class DistributorSendKey
```

SecureMulticastSend

```
package ksu.cis.example.KeyDistribution;

import java.net.*;
import java.io.*;
import java.security.*;
import ksu.cis.mom.*;

public class SecureMulticastSend extends SecureMulticastConversation
{
    private ClientAgent parent;
    private ClientAgentGUI gui;
    private String mesg;
    private String conversation_name = "SecureMulticastSend";
    public SecureMulticastSend(
        ClientAgent a,
        int port,
        InetAddress group,
        String m,
        ClientAgentGUI gui,
        Key k,
        String algorithm)
    {
        super(a, group, port, a.secure_multicast_queue, k, algorithm);
        parent = a;
        this.gui = gui;
        this.mesg = m;
    }

    public void run()
    {
        Message m = new Message();
        int state = 0;
        boolean notDone = true;
        try
        {
            while(notDone)
            {
                switch(state)
                {
                    case 0:
                        m.performative = "display";
                        m.content = this.mesg;
                        startConversation(m, conversation_name, "SecureMulticastReceive");
                        state = 1;
                        break;
                    case 1:
                        notDone = false;
                        break;
                }
            }
        }
        catch (IOException e)
        {
            parent.write("*** IO Exception in secured multicast conversation.");
        }
    }
}
```