Applying Broadcasting/Multicasting/Secured Communication to agentMom in Multi-Agent Systems

User Manual

Version 1.0

This document is submitted in partial fulfillment of the requirements for the degree MSE.
This document is an updated of the previous agentMom's user manual.

Chairoj Mekprasertvit
CIS 895 – MSE Project
Kansas State University
Spring 2004

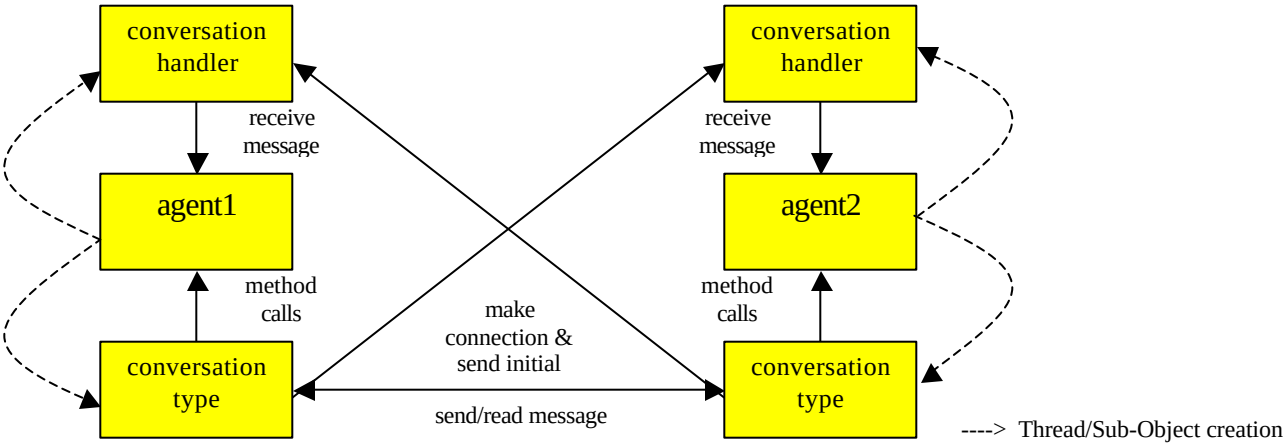
Using agentMom

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.

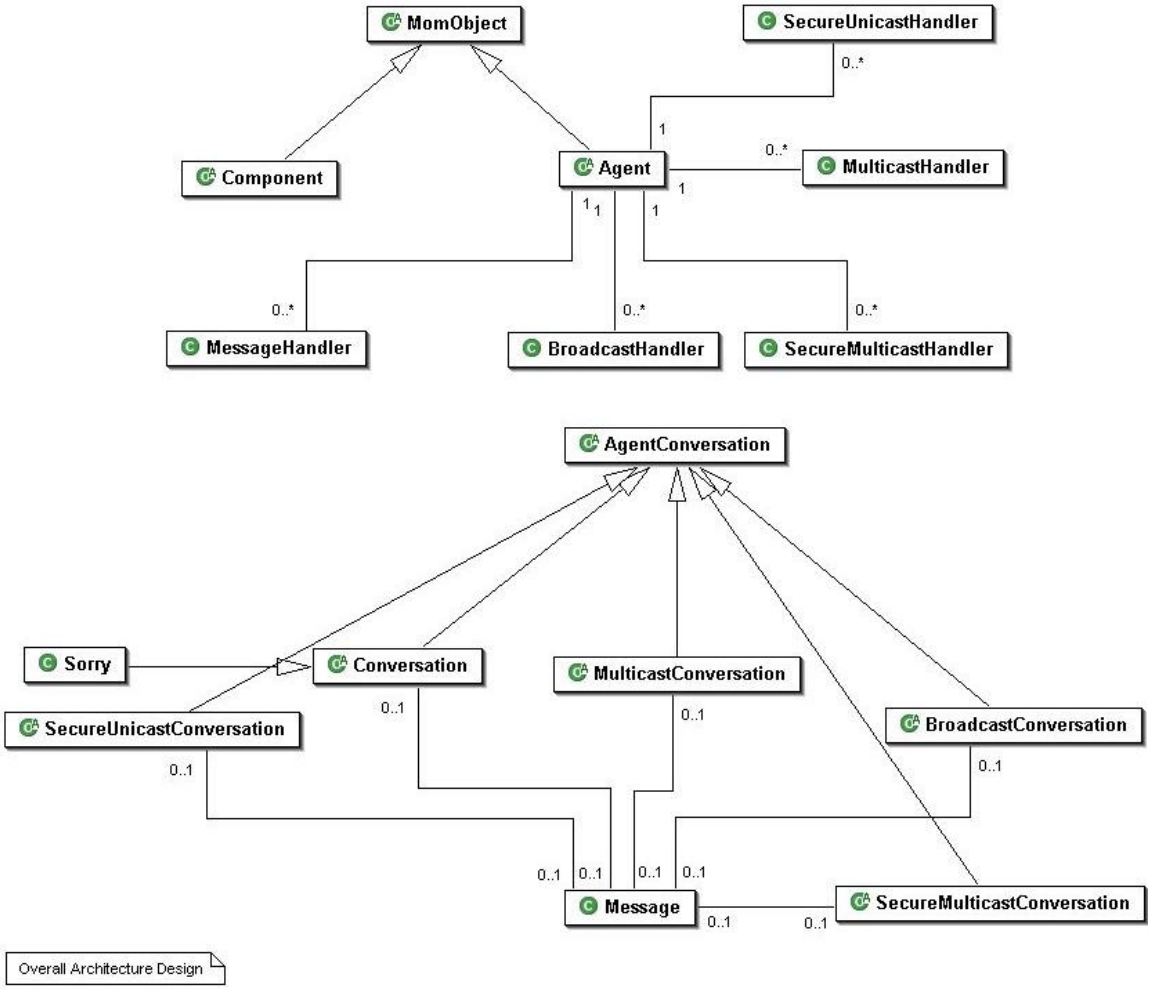
1. Unicast conversation using TCP/IP. This is basically a one-to-one communication.
2. Secured unicast conversation using Secure Socket Layers (SSL) over TCP/IP.
3. Multicast conversation using multicast socket and datagram packet. With multicast conversation, an agent is capable of sending a message 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.

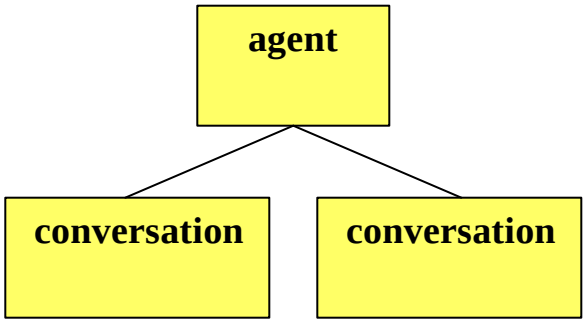


2. How to use agentMom

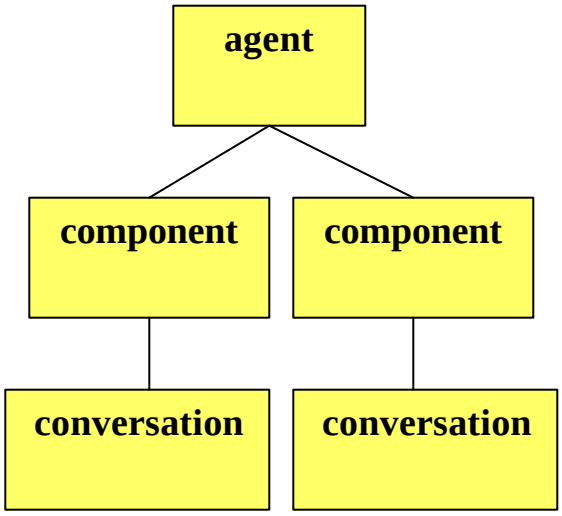
The ksu.cis.mom package, which makes up the basics of agentMom, is shown below. It consists of nine abstract classes, MomObject, Agent, Component, AgentConversation, Conversation, SecureUnicastConversation, MulticastConversation, SecureMulticastConversation and BroadcastConversation, and seven concrete classes MessageHandler, SecureUnicastHandler, MulticastHandler, SecureMulticastHandler, BroadcastHandler, Message and Sorry. The actual source codes associated with these classes are in Appendix A.



Note that the name Conversation and MessageHandler are not consistent with the other conversations because we want to 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

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. agentMom Package

3.1 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.

3.2 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, its 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.

3.3 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.

3.4 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.

3.5 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.

3.6 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]:  chairoj 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=chairoj 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=chairoj 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
```

3.7 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.

3.8 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.

3.9 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 to 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.

3.10 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 Appendix B. 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 Appendix B) 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();
```

3.11 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 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. 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 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 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.

3.12 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.

3.13 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:

```
SSLSocketFactory sslFact =
  (SSLSocketFactory) SSLSocketFactory.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();
```

3.14 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("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 **");
  }
}
```

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

3.15 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 ontoloty 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.

3.16 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.

4. 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 as shown in Appendix A.

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.

Appendix A
agentMom Source Code
please refer to
(www.cis.ksu.edu/~cme6556/src)
the source code will include in the final document.

Appendix B
Register-Deregister example

Appendix C
Key Distribution example