A SOFTWARE AGENT SYSTEM FOR PRIVATE EMAIL

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Hari K. Mukka

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

March 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

A SOFTWARE AGENT SYSTEM

FOR PRIVATE EMAIL

By

HARI K. MUKKA

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Hari K. Mukka, M.S., Department of Computer Science, College of Science and Mathematics, North Dakota State University, March 2010. A Software Agent System for Private Email. Major Professor: Dr. Kendall E. Nygard.

The primary concerns in existing Email and instant messaging systems are authentication processes, privacy and security issues. These messaging systems transfer the message content through servers using protocols like the Simple Message Transfer Protocol (SMTP), Post Office Protocol (POP), and Internet Message Access Protocol (IMAP) to allow users to communicate. During these processes, messages are being stored on the servers which can be easily accessed by the network administrator. Invasion of privacy, integrity of Email content, lack of authentication, and unprotected back-up of messages stored on servers could be common underlying problems in the existing architecture.

The goal of this paper is to build an Email messaging system which would offer better privacy and security of Email content to users. The application can be viewed as a source of communication for people in a private club. In this paper we propose three different designs for building an email messaging system using the Java Agent Development Environment (platform to provide communication between agents) framework. Our application mainly uses intelligent software agents. The three Designs primarily implement two kinds of software agents, a Facilitator Agent and P-Mail (Private-Mail) agent. Based on the Design and architecture these agents act accordingly to meet the user requirements and provide better security and privacy. Finally we conclude by evaluating the three designs by assessing their performance.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Kendall E. Nygard, for his continuous support, guidance, and expertise. My sincere thanks to Dr. Juan Li, Dr. Gursimran Walia, and Dr. John Cook for serving on the committee. In addition, I would like to take this opportunity to thank the faculty and staff of Computer Science Engineering for being part of the success of my master's study.

Finally, I would like to thank my family and friends for their continuous encouragement and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Electronic mail (Email) and instant messaging system have become the most widely used internet applications to communicate. These applications have become a frequent form of communication for people in the modern era of technology and communication. The Internet and their rich internet application have changed the way we communicate and turned global business into reality. With great ease, we are able to communicate from one part of the globe to another in very little time through Emails. Email and instant messaging systems have shown significant impact in terms of the amount of data transmitted throughout the world. Email and instant messaging systems have significant drawbacks while providing privacy and security features to their clients. In this paper we make an attempt to develop an application for messaging which could be used by a private group of members (private club) to communicate, providing them with more privacy in a more secured network.

1.1. Impact of Email and Instant Messaging

Email and instant messaging system usage has shown considerable impact on the current nature of business. Many corporate companies are encouraging their employees to use instant messaging to communicate with their co-workers, which makes the communication process more efficient. It cuts down the usage of phone calls when people prefer something in written form. Email shows great significance in the following ways:

- Providing a convenient communication paradigm
- Replacing regular mail
- Supporting business communication

- Supporting users in rural areas
- Providing convenient access through handheld devices

1.2. Instant Messaging Systems

Instant messaging (IM) networks make use of servers and rich internet protocols as Email messaging system in-order to transfer the message content from one client messenger to another client on the network [11]. The user installs a messaging client on the system which connects to a server which is being processed by IM network vendors (such as Yahoo messenger, AOl, Gtalk). Each IM service makes use of different protocols to operate on servers, limiting the interoperability of users on one IM service to communicate with users on other IM services.

When users want to communicate they start sending the messages to each other via servers. During this process of transferring the messages the process is being logged on the server causing the threat to security and privacy of the user content. In most of the IM any user that has successfully logged onto the system can communicate with other user in plain text [11].

1.3. The Way Email Works

Millions of Email messages are being sent across the globe every day. When there is an Email sent across a server, the domain name is stripped off, and the respective Email server is contacted. The process of Email communication involves multiple protocols and various types of servers [2].

In Email communication, a protocol can be defined in terms of a set of predefined rules to be followed in order to successfully communicate or transfer data between the client and the server [2]. Some of the commonly used protocols for communication are

Internet Protocol (IP), Transmission Control Protocol (TCP), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Simple Mail Transfer Protocol (SMTP). These servers are being used as per the requirement to serve the purpose of communication in general. In the process of Email communication, we deal with a client and a server that exchange information with each other using a wide variety of protocols.

1.3.1. Protocols in Action

Email clients typically use a variety of protocols to allow users working on different systems to communicate with each other across the networks. The standard protocol used for client-server communication is Simple Mail Transfer Protocol (SMTP).

The SMTP protocol can be used to send Emails to the recipient mail server through the Mail Transfer Agent across Internet Protocol (IP) networks. This protocol is usually used with Post Office Protocol (POP) or Internet Message Access Protocol (IMAP), allowing users to access messages residing on the servers.

Internet Message Access Protocol (IMAP)

The IMAP protocol is a standard protocol used to retrieve Email messages from the local servers [2]. IMAP is the most prevalent internet protocol that allows the Email clients to access data on a remote mail server. IMAP also allows interoperability with other servers and clients on the networks.

Post Office Protocol (POP)

POP is one of the standardized Internet protocols to access mailboxes and allow messages to be downloaded to the systems on the application network layer. There are several versions of POP that have been developed; the current version in use is POP3.

1.3.2. Traditional Architecture of Email Messaging System

The process of sending an Email message is very similar to our traditional mail communication procedure. To send a mail message from one place to another, the mail has to pass through a group of local post offices and through the regional post offices. An Email message can be sent either through a web-based interface (i.e., Yahoo! or Gmail) or through Email client programs, such as Simple Mail Transfer Protocol (SMTP), which take care of most of the message-sending processes to the recipient on a network layer. When an Email message has to be sent to a recipient, the user needs to explicitly mention the server name to which the message is to be sent. The web server comes into play, contacts the respective SMTP server, and processes the Email message.

Once the SMTP server picks up the Email message from the web-server, it can send the message to the recipient SMTP server in multiple ways (Figure 1).

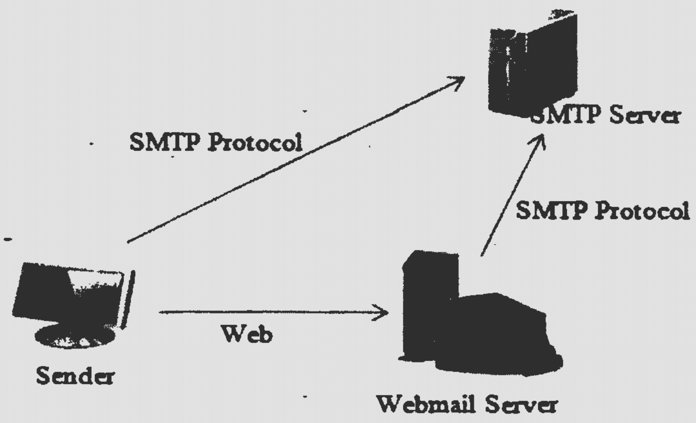


Figure 1. Sending an Email message [2].

- The Email message can be directly transmitted to the receiver's SMTP server.
- The sender's SMTP server calls the back-up server to back up the message and then, in turn, calls the receiver's server.

- The SMTP server can even stack the message and try sending it later.

Any of the steps could come into play, and the message will be dropped onto the receiver's server to be picked up by the user.

When transmitting the Email message from sender's SMTP server to the recipient's SMTP server, the message may be processed through several other SMTP servers (Figure 2). Each server adds a "Received" stamp to the message. These message stamps give users information about all the servers that have come into action in order to transmit the message to the destination SMTP server. The received Email messages will be stored in a file system on the recipient's server. In order to access the message stored in the file, the SMTP server has to process these messages either through Internet Message Access Protocol (IMAP) or Post Office Protocol (POP).

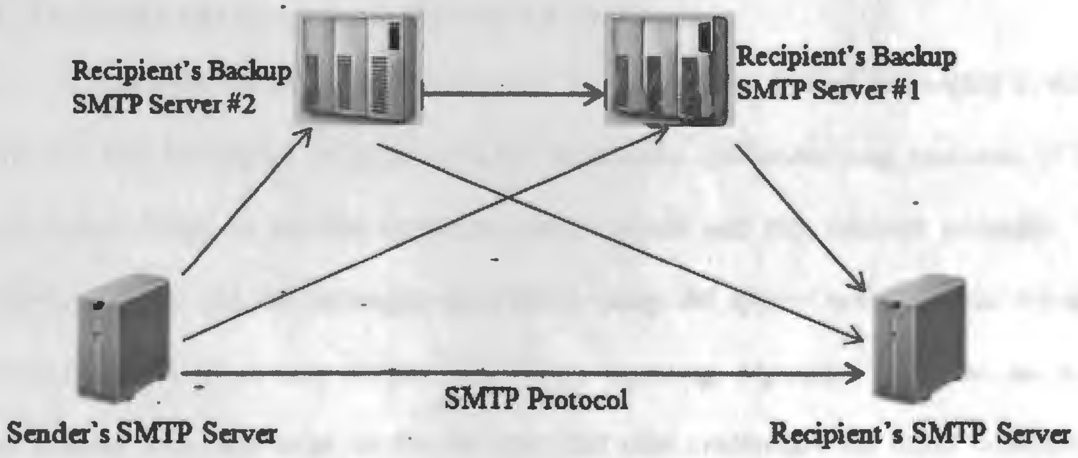

Recipient's Backup SMTP Server #2
Recipient's Backup SMTP Server #1
SMTP Protocol
Sender's SMTP Server
Recipient's SMTP Server

Figure 2. Delivery of Email from senders SMTP server to recipients SMTP server[2].

The Email client programs contact the servers through the IMAP or POP protocol to retrieve the Email message for the user. As a final step (Figure 3), the recipient can either use a web-based interface or a computer's Email client program to retrieve the Email message from the server.

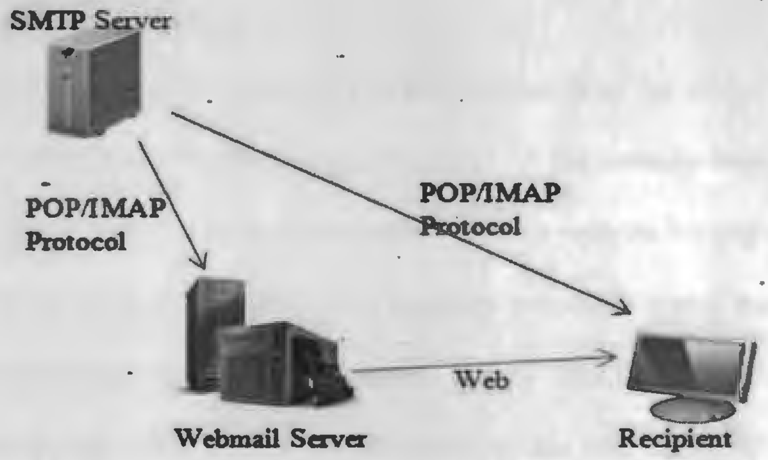


Figure 3. Retrieving the Email message from the SMTP server [2].

1.4. Problems Involved with Messaging Systems

A thorough understanding about how Email and the instant messaging systems work will also be-helpful to understand the drawbacks and underlying problems of the architectural design to process messages using servers and rich internet protocols. To elaborate, Emails and the messages transmitted using IM system are processed through servers are prone to security attacks and privacy concerns. Messages which we assumed were deleted may- still exist on the servers. The user credentials are never completely secure once we log-in to servers which can be easily hacked by professionals. If one gains access to a server, the messages can be easily accessed by a network administrator, and the message content can be changed before the message is sent to the recipient. These concerns about information security and data integrity of the Email messages exist on the application network layer. The standard Simple Message Transfer Protocol (SMTP) fails to support message encryption during the transmission process through the servers. The messages are sent in plain text format which, in turn, raises a question about integrity of the message content.

1.4.1. Security and Privacy Issues

During the process of delivering the Email message from the sender's SMTP server to the recipient's SMTP server, there is a good chance of the message being backed up on the server and stored there for a significant period of time without leaving any indications to the sender or the receiver. The process of message transfer is pretty much the same in case of instant messaging making use of servers. At any time, the administrator can gain access to the backed-up messages that are on the server and read the Email messages. The user's private information, like user name and password, which are needed by the POP and IMAP protocols in order to retrieve the Email messages residing on the server's file system can be read by any eavesdropper who understands the flow of information between the system and the servers on the application layer. The common underlying security threats are as follows:

- Eavesdropping (can easily gain access on the Email messages)
- Identity theft (insecure identity)
- Fraudulent Email messages (false Emails)
- Lack of authentication
- Invasion of privacy (concerns about Email privacy)
- Integrity of Email content (lack of data integrity)
- Unprotected backup (back-up of messages stored on servers)

Email messages sent through the Email clients may even include the internet protocol address which could reveal information such as the sender's location and the city from which the message was sent through the Email servers.

1.5. Pretty Good Privacy (PGP)

The privacy of Email messages over the internet has been compromised quite a bit because they are backed up on servers which can be easily accessed by the network administrator at any time [7]. Pretty good privacy (PGP) is a way of providing more security and privacy for email messages by encrypting and decrypting the Email messages throughout the system. PGP provides cryptographic privacy for Email messages.

1.5.1. How Does PGP Work?

PGP is public key encryption technique which uses the asymmetric key algorithms to encrypt and decrypt the Email message. PGP has a special command to process the creation of your own personalized pair of public and private keys. Public-key encryption provides the user with a pair of cryptographic keys known as public key and private key. The private key is always kept as a secret, and the public key can be distributed throughout the network [7].

When a user sends an Email to a recipient, the message is encrypted with the recipient's public key which can only be decrypted with its corresponding public key. Encryption algorithms never provide 100% security for the users. PGP assures users to protect the content of Email messages by encryption and decryption, but fails to protect the identity of either the sender or the recipient.

1.6. Proposed Idea

All the above-mentioned problems, which are related to the security and privacy issues of sending an Email message or an instant message through IM are being processed through servers and making use of standardized protocols. This threatens the users with regard to sending private content, leaving signs of insecurity in the minds of users [1]. The privacy and security concerns need to be considered, and by valuing the user concerns, we need to build an application which makes use of protocols to less effect and which avoids the unnecessary backup of user Email messages or the instant messages being logged on the servers.

This paper proposes an idea to build an application based on a software agent's collaboration. The Private mail (P-Mail) agent messaging system transfers the Email message from the sender to the receiver with the help of intelligent software agents. The P-Mail software agents are programmed as per the user requirements, making the content more secure and private by communicating the information to other agents. Software agents transfer the Email message to the recipient with less concern for privacy and security. This paper proposes an application which would emphasize user authentication more by providing more security to Email message content.

1.6.1. Comparing the Proposed Idea with the Existing Architecture

The proposed Private Mail (P-Mail) messaging system doesn't implement any kind of server's in-order to transfer the message from one user to another on the network. Instead they will be using a Facilitator to provide the access control to clients to transfer the message to the recipient. The proposed architecture is more suitable to the pool of people who want to communicate in private we call a private club. Every time the user needs to authenticate with the recipient client in order to send a message and also needs to get the permissions to transfer the message successfully. The proposed design is a different from the way communication occurs between the users when compared with the IM systems or Email messaging system aiming to have the process in more private and secure.

1.6.2. Goals and Objectives

The objectives of the P-Mail messaging system can be defined as

- To build an application which uses protocols to less extent
- To avoid unnecessary backup of Email messages
- To build a secure Email messaging system

2. SOFTWARE AGENT BACKGROUND

The use of software agents is rapidly growing in the field of computer science and artificial intelligence. Software agents are programs that can mimic most human behaviors or activities. The P-Mail software agent messaging system explains how software agents collaborate with other agents to carry out the task of successfully transferring Email messages from sender to recipient along with maintaining the privacy of the messages sent.

2.1. Software Agents

A simple definition that can be given for "software agent" would be any software program that is designed to perform or carry out an assigned task automatically in order to retrieve or transfer the information with intelligence. Programming agents can perform the tasks as per the context's requirements. They perform the task continuously in a flexible manner, responding to changes in the environment without requiring any human intervention. Nwana proposed an agent typology [3] for agent classification according to which agents can be classified as illustrated in Figure 4.

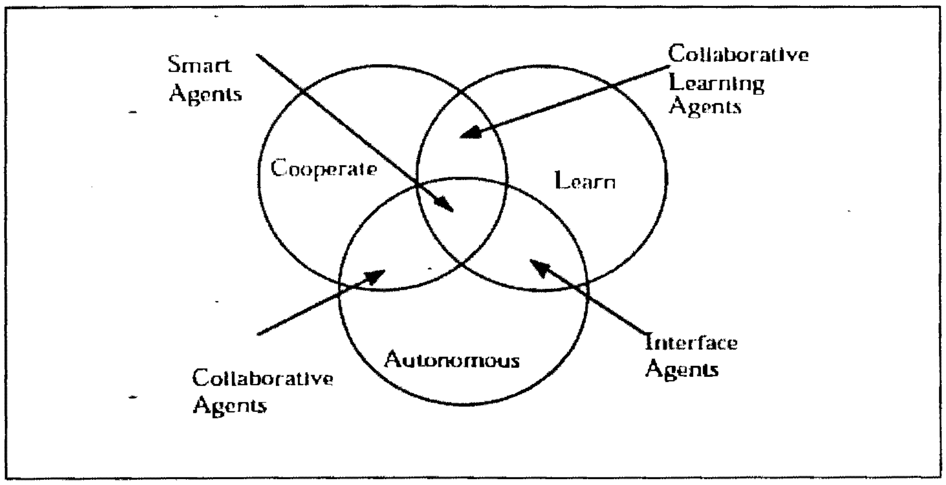


Figure 4. Nwana's Proposed Agent Typology [3].

Figure 4 illustrates that agents can be autonomous (ability to control their own actions to be performed), co-operative (ability to communicate with other agents effectively), reactive (responding in a timely manner), goal oriented (knowing the job to be performed), mobile (ability to move in an environment) and sufficiently flexible in their actions. Based on the above-inherited features, the agents can be classified into the following categories [8]

- Collaborative or cooperative agents
- Interface agents
- Learning agents
- Smart agents or hybrid agents

Multi-agent systems (MAS) can be defined as a group of intelligent agents that collectively interact with other agents to carry out a problem to be solved [8]. A MAS performs the operation by collaborating with other agents which might be difficult to carry out with single agent.

2.2. Why Software Agents?

In the proposed idea, software agents could be used to address the problems with privacy and security in the process of transmitting the Email message from one user to the other. In a traditional Email messaging system, there is always a problem with invasion or integrity of the message content. We choose software agents which can collaborate with other agents to process the entire Email messaging system.

2.2.1. Basic Structure of P-Mail Instant Messaging System

The P-Mail Instant Messaging System rules out the option of using servers and makes use of P-Mail agents. This is an attempt to ensure true privacy in an Email system.

In the proposed architecture, each P-Mail client is an intelligent software agent with considerable flexibility to collaborate with other P-Mail clients (figure 5).

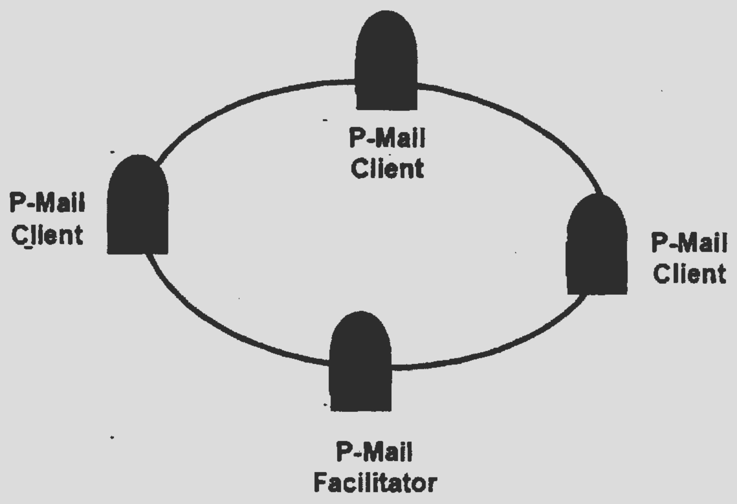


Figure 5. Private Instant Messaging System (P-Mail) [5].

Each P-Mail agent can communicate with every other P-Mail agent that it recognizes and can decide whether to accept the messages or not. During this process, messages are neither stored on any server nor back-up for any particular reasons. The above architecture is an idea derived from peer-to-peer communication on a network layer. The system is built using one of the most popular programming languages, Java on JADE (Java Agent Development Environment).

2.3. Java (Object-Oriented Programming Language)

This object-oriented programming language was developed by James Gosling at Sun Microsystems. Java is based on C++ syntax and differs in many ways in its style of programming. Once correctly compiled, Java programs are portable and can be run securely across different platforms. Java is a high-level, object-oriented programming language that provides many additional features and benefits. It is simple, object oriented, multithreaded, robust, portable, high performance, secure, etc. Java is the core programming language used in this paper to develop the P-Mail software agent application in JADE.

2.4. Java Agent Development Environment (JADE)

JADE is a framework that simplifies the implementation of multi-agent systems in compliance with the FIPA (Foundation for Intelligent Physical Agents) specifications [9]. JADE was fully developed using the Java programming language. It provides a middleware for developing and successfully executing the agent applications. It provides mobility, security, and many other features so that agents can collaborate with several other agents on the platform. JADE contains one main container for the agents and also allows the creation of several other containers which can reside on the same system or on different systems. A series of containers is collectively called a JADE platform.

The main container on JADE always holds two special agents:

- Agent Management System (AMS): provides the naming services, ensuring agents are unique and also destroying the agents.
- Directory Facilitator (DF): provides a service which is very similar to yellow pages and which is helpful for other agents to find and communicate successfully.

2.4.1. JADE Packages

The JADE [9] environment includes a library of classes that programmers need to use to build their software agents and to perform actions to collaborate with other agents. Some of the core packages used by JADE for multi-agent systems are

- Jade.core.acl: implements the Agent Communication Language (ACL) messaging service

- Jade.core.event: implements the event notification service
- Jade.core.management: implements the agent life-cycle management service
- Jade.core.messaging: implements the messaging service
- Jade.core.mobility: implements the mobility and cloning service

2.5. NetBeans IDE

NetBeans is a free, open-source integrated development environment (IDE) for software developers that are built on Java. NetBeans is used to develop rich client applications. Each NetBeans module provides a well-defined function. It provides reliable application architecture that is developed in no time. On NetBeans, it is easy to develop or build applications that are robust and extensible across the framework. NetBeans IDE is easy to use; it is quick to learn and to develop applications which are robust and rich in functionality. The P-Mail software agent messaging system is built using JADE on the NetBeans IDE.

2.6. Working with JADE on NetBeans

The only software requirement needed by JADE to execute the systems is the Java Run Time Environment (Version 1.4). After the class path is set to the root directory, the following command is used to launch the main container of the platform [9]:

Java jade.Boot [options] [AgentSpecifier list]

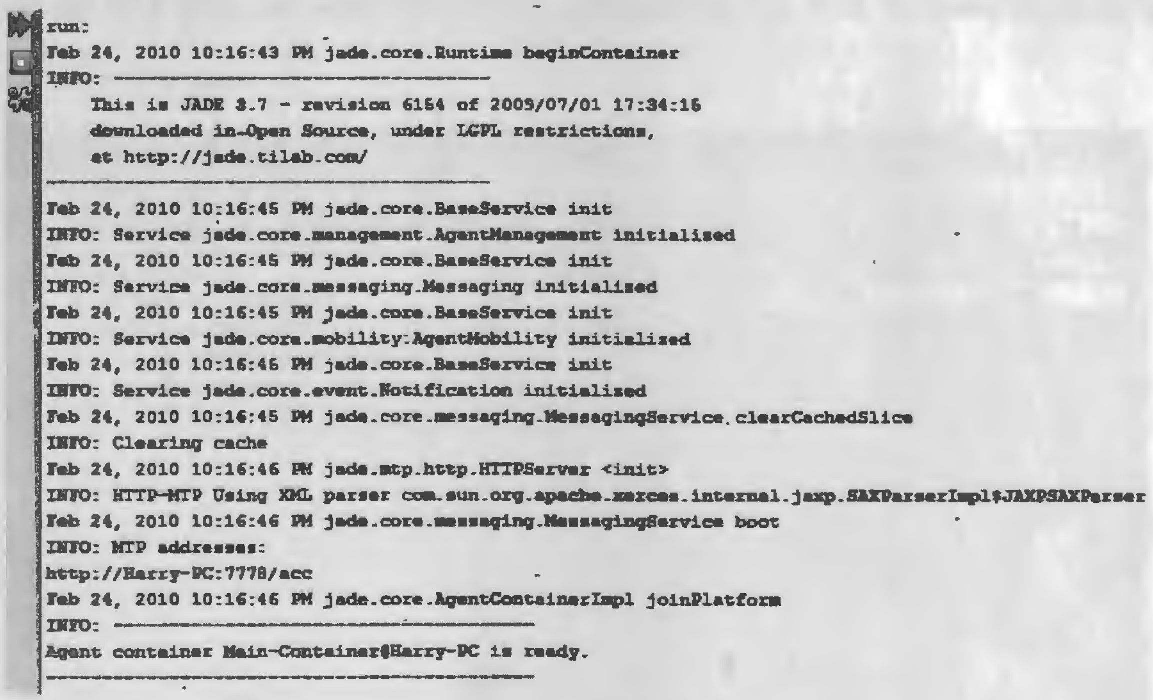
In order to launch a new agent container, the following command can be used:

Java jade.Boot –container [options] [AgentSpecifier list]

The *Agent.doDelete( )* method can be called to stop the agent execution, and the *Agent.takeDown()* method can be called to destroy or suspend the agent.

2.7. Loading Agents

JADE can be run in several different ways; it can be run on one computer or many other computers. The simple way is to run JADE on one single system and one single main container. The following command can be used to boot JADE: *run jade –gui* [9]. Once the boot process is complete, the message shown in Figure 6 is displayed on the console of NetBeans IDE.

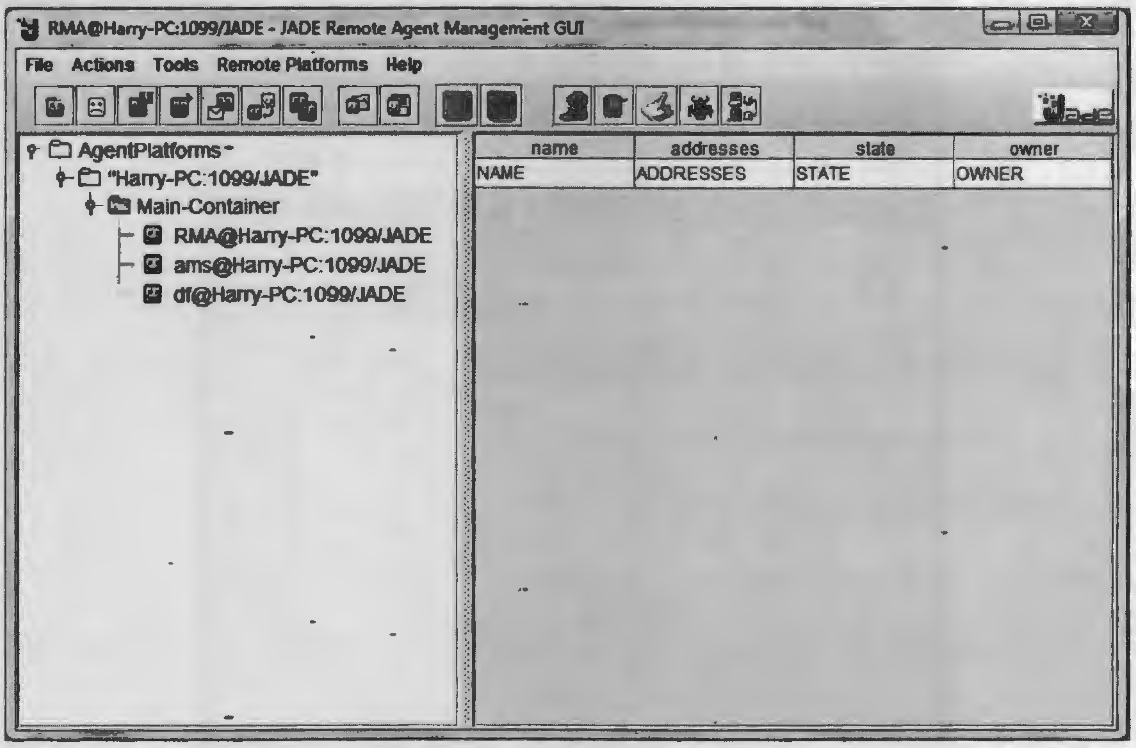


```
run:
Feb 24, 2010 10:16:43 PM jade.core.Runtime beginContainer
INFO: ----------------------------------------
    This is JADE 3.7 - revision 6154 of 2009/07/01 17:34:15
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
Feb 24, 2010 10:16:45 PM jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
Feb 24, 2010 10:16:45 PM jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
Feb 24, 2010 10:16:45 PM jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
Feb 24, 2010 10:16:45 PM jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
Feb 24, 2010 10:16:45 PM jade.core.messaging.MessagingService.clearCachedSlice
INFO: Clearing cache
Feb 24, 2010 10:16:46 PM jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
Feb 24, 2010 10:16:46 PM jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://Harry-PC:7778/acc
Feb 24, 2010 10:16:46 PM jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------------
Agent container Main-Container@Harry-PC is ready.
----------------------------------------
```

Figure 6. Console message on JADE boot.

The Remote Monitoring Agent (RMA) controls the life cycle of the agent [4]. The following commands can be executed from the RMA Graphical User Interface (GUI) tool bar. "*File Menu*" has the "*Close*" option to close an existing RMA agent. We can terminate the agent by calling the *doDelete( )* method. The "*Shut Down Agent Platform*" option terminates all the existing containers and the living agents. The "*Start New Agent*" option creates a completely new agent, and the "*Kill Selected Items*" option takes down all selected agents at once. "*Suspend Selected Agents*" calls the *doSuspend( )* method intact to suspend the agent actions [9].

Figure 7 shows a snapshot of the RMA graphical user interface window that pops up once the boot command is run successfully. The RMA GUI provides the user with a wide variety of options to create and manage agents. The user can initiate the agent with just one click on the GUI icon. All operations can be performed in the main container or when a container is created.

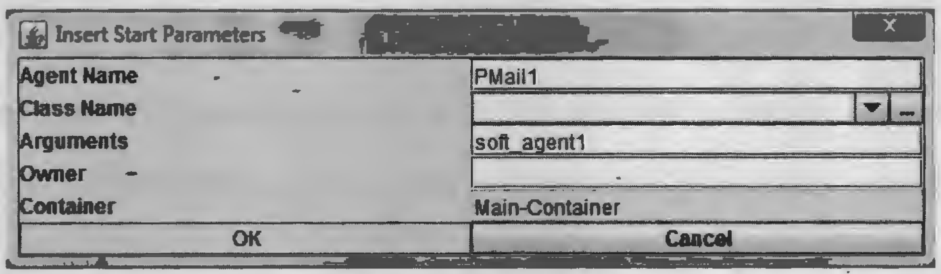


name	addresses	state	owner
NAME	ADDRESSES	STATE	OWNER

Figure 7. RMA GUI.

"Start new agent" pops up a new window (figure 8) that provides the user with the option to enter the agent name, argument, and class name from where the agent has to be initiated.

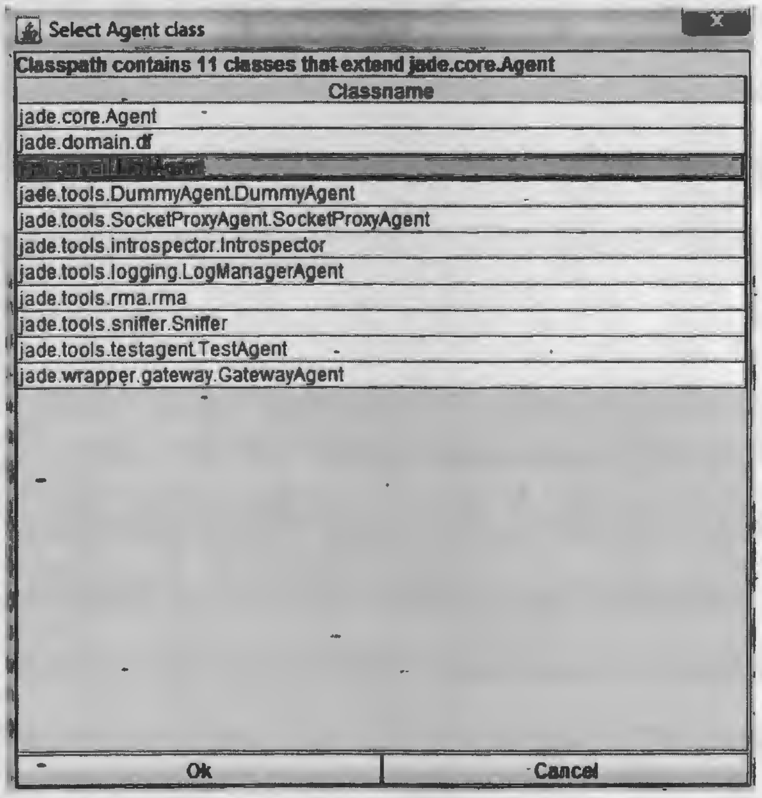
A tab besides the class name provides the user with all available agent classes to create an agent using it's instance.


Insert Start Parameters [redacted]

Agent Name	PMail1
Class Name	
Arguments	soft_agent1
Owner	
Container	Main-Container

Figure 8. Load agents with RMA.

Figure 9. shows the pop-up window with all the available class names to select and create an agent. By selecting a class name and then saying OK, the respective class agent will be created immediately.


Select Agent class

Classpath contains 11 classes that extend jade.core.Agent

Classname
jade.core.Agent
jade.domain.df
[illegible]
jade.tools.DummyAgent.DummyAgent
jade.tools.SocketProxyAgent.SocketProxyAgent
jade.tools.introspector.Introspector
jade.tools.logging.LogManagerAgent
jade.tools.rma.rma
jade.tools.sniffer.Sniffer
jade.tools.testagent.TestAgent
jade.wrapper.gateway.GatewayAgent

Figure 9. Select Agent Class.

In Chapter 3, we will discuss more about the design and requirements of the P-Mail Messaging System.

3. DESIGN AND REQUIREMENTS

This chapter gives a brief introduction about the design and requirements for building the P-Mail software agent messaging system. The idea behind implementing the P-Mail software agent messaging system is to keep the message secure and private while transmitting the Email message from sender to the receiver. To serve this purpose, we use agent-to-agent collaboration on the Java Agent Development Environment (JADE).

3.1. P-Mail Software Agent Requirements

P-Mail software agent messaging can be implemented by software agent collaboration to communicate in many different ways, from which we consider three potential scenarios that can be implemented. In this chapter, we will go into detail about how these three scenarios differ from each other in terms of design and requirements.

The P-Mail messaging system requires two different kinds of software agents. According to the purpose requirements, these agents have been named as

- P-Mail agent
- Facilitator agent

These agents behave according to the user requirements and the way they have been programmed to serve the purpose of messaging from one P-Mail client to another. Both P-Mail and facilitator agents are intelligent software agents. A facilitator agent plays an important role in the design of the entire P-Mail messaging system. It facilitates communication between the P-Mail agents and P-Mail clients. The messages are never stored on any of the servers or saved at any other locations. The P-Mails will be sent from one client agent to another client agent without any intermediate support to conduct the task. More about the agent's design will be discussed in the next section.

3.2. P-Mail Messaging System Design

The P-Mail messaging system has been designed to provide privacy and security to the Email messages. Potentially, this also eliminates the possibility of a client receiving SPAM messages. The peer-to-peer architecture of the P-Mail messaging system allows a P-Mail client to send an Email message to any client on the network.

Figure 10 presents the high-level architectural design of the P-Mail messaging system. An individual P-Mail client can be set-up on the user's machine, which can be a desktop, PDA, etc. Each client machine has a user-interface to send, receive, compose, and read the Email messages. In order to send messages to a client, all clients should communicate through the facilitator agent to get the address of a specific client to transmit the message which is shown in red. Messages can only be sent to other P-Mail clients, not to the facilitator, the communication paths of which are shown in black on the network [4].

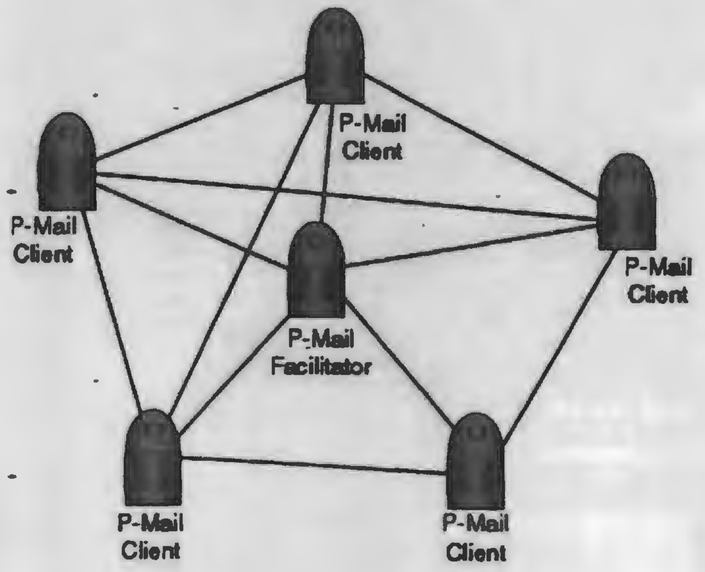
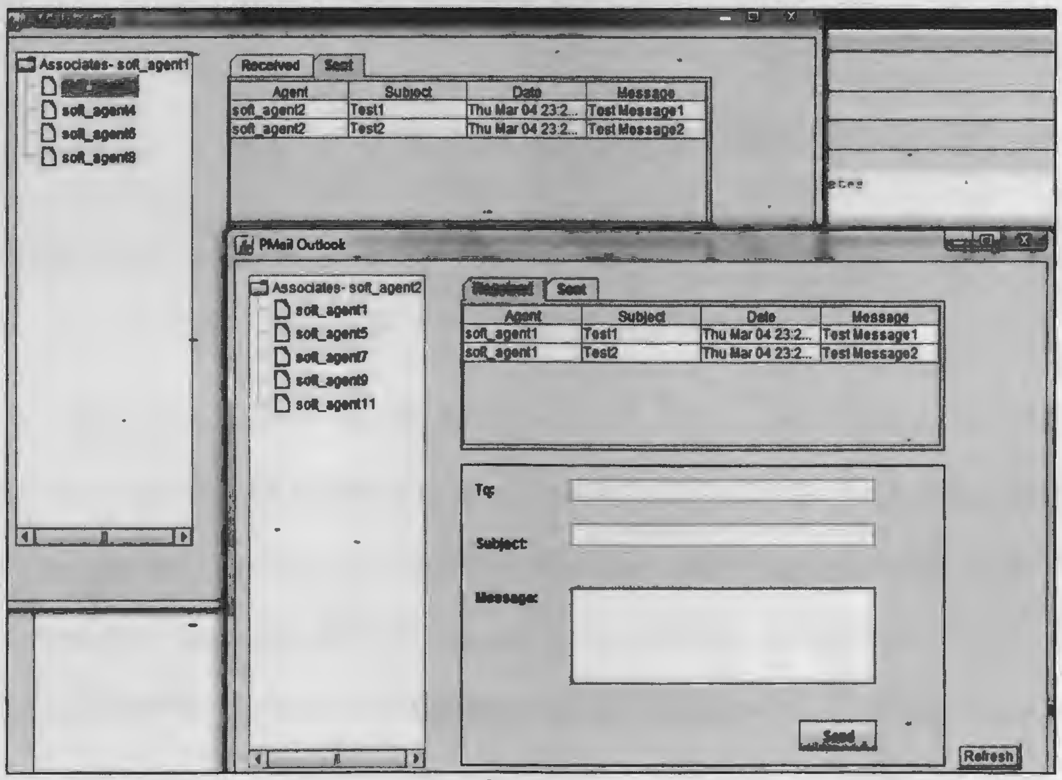


Figure 10. Simple Design of P-Mail Messaging System [4].

3.2.1. P-Mail Client

The P-Mail client has a user interface to send/receive Email messages to/from other P-mail clients on the network. Communication between the P-Mail clients is restricted on a network layer. A P-Mail client would communicate with other P-Mail clients only if it is present in the associate list, which means they are programmed in such a way that an agent can communicate with any other agent that it knows and can accept its messages. Each client agent has a list of agents with whom it can communicate. The client interface can be divided into three different components (figure 11):

- Associate list tab
- Sent and received messages tab
- Compose tab



Agent	Subject	Date	Message
soft_agent2	Test1	Thu Mar 04 23:2...	Test Message1
soft_agent2	Test2	Thu Mar 04 23:2...	Test Message2

Agent	Subject	Date	Message
soft_agent1	Test1	Thu Mar 04 23:2...	Test Message1
soft_agent1	Test2	Thu Mar 04 23:2...	Test Message2

Figure 11. P-Mail Client Interface for Messaging.

3.2.2. Associate List

The Associate list provides a list of all P-Mail clients(Figure 12) present on the network with which a user can communicate effectively. The Associate list displays list of clients with two different statuses, displaying all the network clients that are currently online/offline.

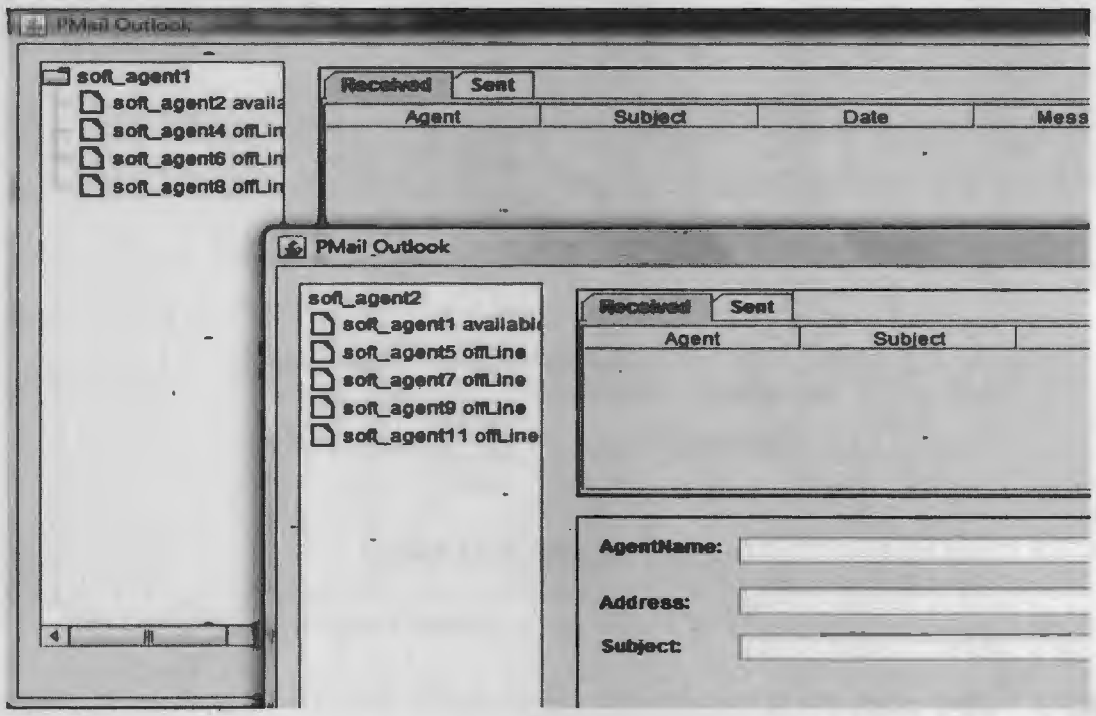


Figure 12. P-Mail Client Associate List.

The associate list will be refreshed periodically, updating the client status. The user cannot communicate with a client that is not present in his or her associate list; in order to do so, the user either needs to get permission from the client, or it needs to add the client to the current list of associates that are online. If the user sends a message to an offline client, the message will be displayed for the client when it goes online.

3.2.3. Facilitator Agent

The facilitator agent plays a significant role in the client communication process. The facilitator never receives the Email message; it only manages the communication between different agents (figure 13). The facilitator agent provides the agent name service and access control between the agents. It stores the information for each agent and its public address.

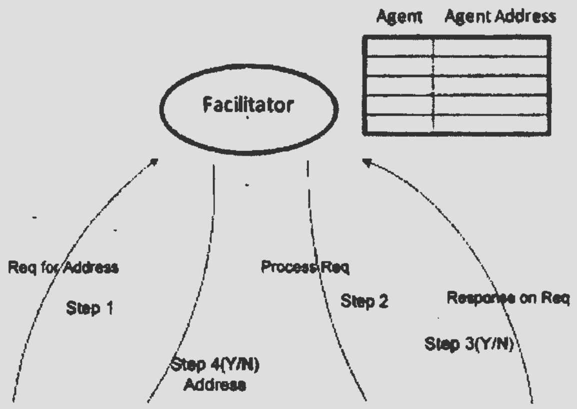


Figure 13. Facilitator Agent.

Whenever a client comes online, it registers with the facilitator saying that it is ready to communicate with other clients on the network, and at the same time, if a client goes offline, it un-checks with the facilitator. The P-Mail client always inquires with the facilitator to get the recipient's address to send the message successfully. The facilitator, in turn, will check with the receiver client if it is willing to communicate with the sender client and replies with the appropriate response. Depending on the way the facilitator works and controls access between the agents, we created three different designs for P-Mail client communication.

3.3. Design Scenarios

On the basis of the functions and the strategies employed by the facilitator to process the request from the P-Mail clients. We have designed three scenarios that can be implemented for the P-Mail messaging system.

3.3.1. Design-1 P-Mail Messaging System

In design-1, the facilitator communicates with the sender's P-Mail client and does not interact with the recipient's P-Mail client. Only on request, the facilitator directly provides the recipient client's address to communicate without trying to see if the recipient P-Mail is willing to communicate.

Process flow Design-1 [Figure 14]:

Step 1: The ultimate goal of the process is to transmit the message from agent-A to agent-B. Agent-B is part of agent-A's associate list. Agent-A, hence, inquires with the facilitator to get agent-B's address during step 1.

Step 2: A facilitator serving the purpose of agent name service has all the agent information and its public addresses for communication. After receiving the request from agent-A, the facilitator provides it with agent-B's address to communicate.

Step 3: Having received the agent name and agent address, agent-A pings agent-B, requesting to start the communication process.

Step 4: Receiving the ping request from agent-A, agent-B needs to decide whether it is willing to communicate with agent-A. If agent-B is willing to communicate with the sender, it would accept the ping request; otherwise, it might deny the request to communicate with agent-A.

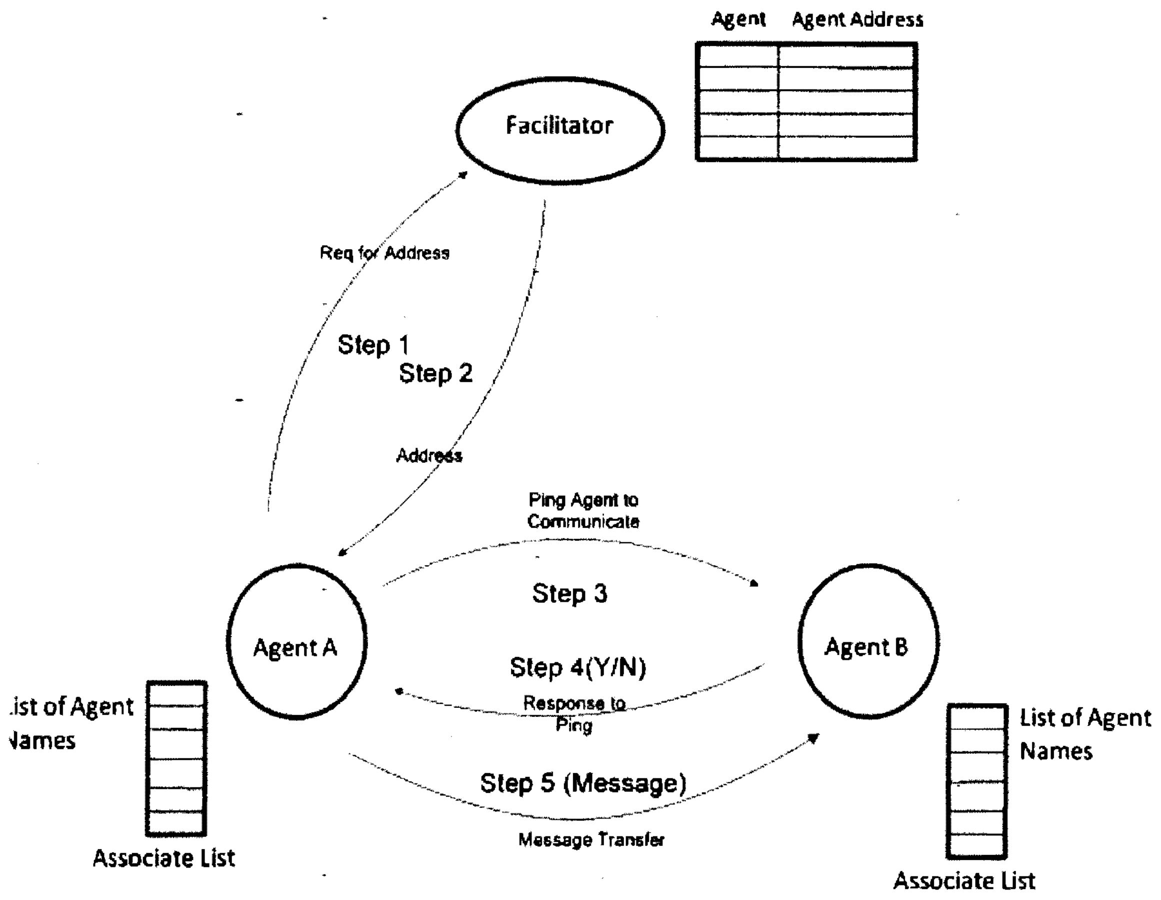


Figure 14. Design-1 P-Mail Messaging System.

Step 5: Agent-A is allowed to send the Email message after receiving the acceptance from agent-B in response to the ping, or it is simply a request denial.

3.3.2. Design-2 P-Mail Messaging System

In design-2, the facilitator is designed to handle the address request from the P-Mail client to the recipient agent to communicate with, thereby sending the same request to the destination agent asking whether to start the process of communication.

Process flow Design-2 [Figure 15]:

Step 1: With an intention to transmit the message from agent-A to agent-B, agent-A needs to get the address by inquiring with the facilitator. In this case, ta request is sent to facilitator agent to get the address.

Step 2: The facilitator makes use of the agent name service and contacts the destination agent to see if it is willing to communicate with the sender (agent-A).

Step 3: After receiving the request from the facilitator, agent-B has to make a decision about whether to allow or deny agent-A to send the message.

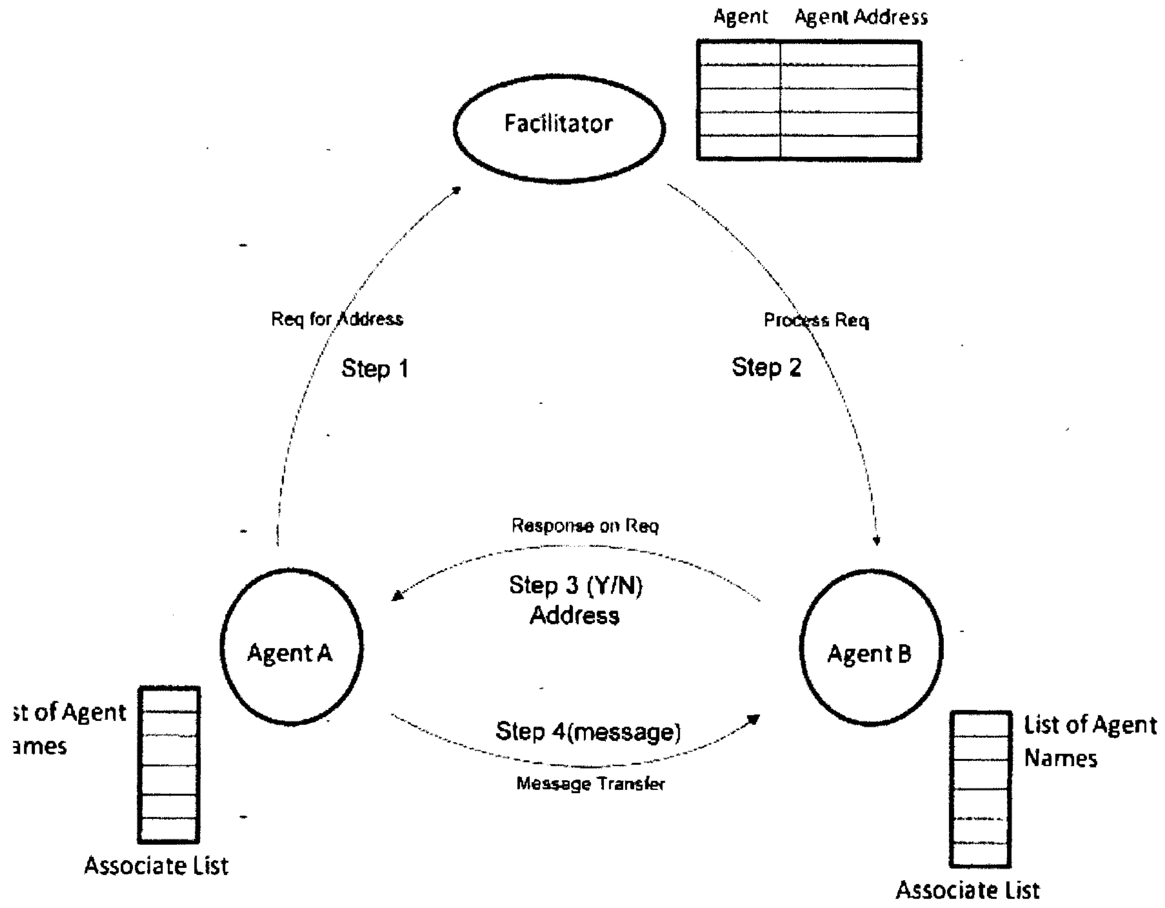


Figure 15. Design-2 P-Mail Messaging System.

Step 4: In step 3, if the destination agent accepts the request, then it will reply with its address to send the Email message. If agent-B accepts the request, then agent-A will have the address to communicate with agent-B; otherwise, agent-A has been denied to communicate with the agent-B.

3.3.3. Design-3 P-Mail Messaging System

Design-3 facilitates the sender P-Mail agent with the recipient's address through the facilitator only when destination agent is willing to communicate; otherwise, the facilitator will simply deny the address request.

Process flow Design-3 [Figure 16]:

Step 1: Agent-A would like to communicate with agent-B and since agent-B is in the associate list. Agent-A requests the facilitator for agent-B's address during the step 1.

Step 2: Having received the request from agent-A, the facilitator processes the request by transferring the request to agent-B. During this process, the facilitator uses the agent name service to contact agent-B.

Step 3: After step 2 and having received the request to communicate, agent-B either has to accept the request by sending the address in response, or it can simple deny the request.

Step 4: If agent-B is willing to communicate, the sender agent will get the address from step 3 to start sending the Email message. If the recipient agent has denied the request, then agent-A is unable to send a message to agent-B.

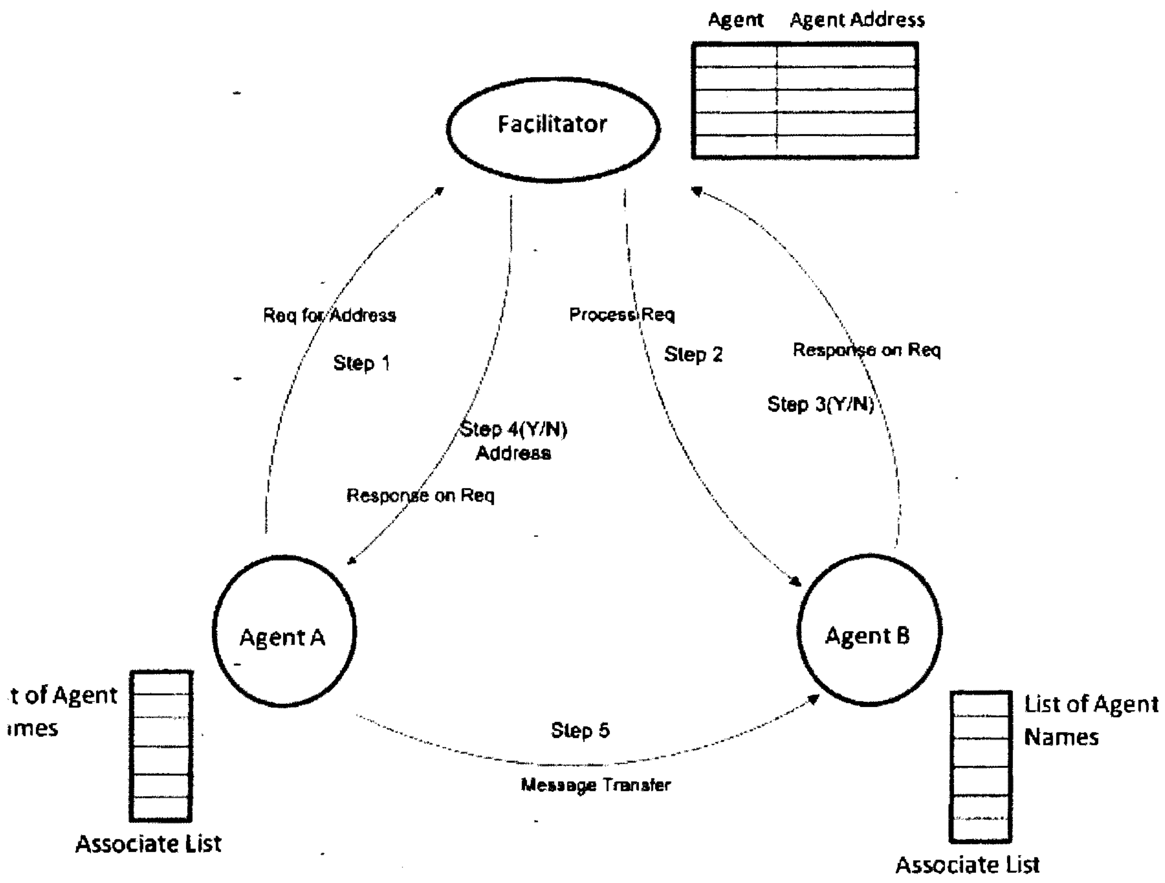


Figure 16. Design-3 P-Mail Messaging System.

Step 5: Once the sender agent has the recipient's address, it is always allowed to send an Email message to the destination agent.

The details of all three P-Mail messaging system designs and their implementations will be discussed in Chapter 4.

4. IMPLEMENTATION AND SCENARIO VISUALIZATION

In this chapter, we get into the details of all three scenarios and their implementation, and we visualize how each design has been implemented. The main goal of each implementation is to transfer Email messages from one P-Mail client agent to another. In order to send the Email message, the sender's P-Mail client needs to get the recipient's address from the facilitator agent. The three implementations differ in

- The way the P-Mail client checks with the facilitator
- The way the facilitator handles the access control.

4.1. Methodology and Classes

In this section, we discuss the classes and functions implemented to handle each design. The project mainly depends on the way the facilitator has been implemented. Each client will be able to compose an Email message with the following fields: client name, address, subject and message. Design has been implemented in such a way that a client can send messages only if the recipient is willing to communicate.

4.1.1. P-Mail Agent Class

The Mail Agent class is the primary class of the entire project, independent of the scenarios to be implemented. This class handles the creation of a user interface on each client machine. The Mail Agent class takes the client name as the argument, creates the user interface (UI), and registers itself with the facilitator agent by giving its name and address information. In table 1 we define the member functions used to build this class. This class creates the instances of two other classes

- *PMailFrame*: Creates the client UI by accepting the arguments and
- *Facilitator*: Client registration

Table 1. Mail Agent Class: Member Functions

[illegible]	[illegible]	[illegible]
void	**setUp()**	Creates agent
void	**takeDown()**	Terminates agent once the process is complete

4.1.2. Load Agents Class

The Load Agents class shown in table 2 is mainly used by the *PMailFrame* and *PmailAgentBehavior* classes to load the Associate list for each client. This class mainly has two kinds of functions; one would store all the client names and their respective associate agent lists, and the other contains the associate list of the client.

```
Class LoadAgents
{
        HashMap loadAgent() {…}
        List agentList1 () {…}
        List agentList2 () {…}
        List agentList3 () {…}
        List agentList4 () {…}
}
```

Table 2. Load Agent Class: Member Functions

Return Type	Function Definition	Description
HashMap<String, List<String>>	**loadAgent()**	Stores client name and its respective associate list of agents
List<String>	**agentList()**	Associate agent list

4.1.3. P-Mail Messages Class

P-Mail messages is the domain object through which we get and set the values of the sender's client name, receiver's client name, message, date, and subject. This domain object is mainly used in other classes to access the values set through this object. Table 3 shows the variables used in this class.

Table 3. P-Mail Messages Class: Member Variables

[illegible]	[illegible]	[illegible]
String	**Message**	Message which needs to be sent
String	**toAgent**	Recipient's client name
String	**from Agent**	Sender's client name
String	**Date**	Current date
String	**Subject**	Subject line of the message

4.1.4. P-Mail Agent Behavior Class

The P-Mail Agent Behavior Class handles most of the actions to be performed by any client to interact with either the facilitator or any other P-Mail client on the network. A few of those behaviors are to ping another agent, populate the received and sent lists on the client's user interface, client authentication process, etc. Table 4 and table 5 gives an brief idea of all the functions used in this class.

Table 4. P-Mail Agent Behavior Class: Member Functions

[illegible]	[illegible]	[illegible]
String	**sendMessage()**	Transfers the Email message with the content to another client.
HashMap<String, String>	**pingAgent()**	Sends a request to another client asking for permission
boolean	**authenticateAgent()**	Takes care of the client authentication process
List<PMailMessages>	**getSendList()**	Gives the send list
List<PMailMessages>	**getRecievedList ()**	Gives the received list
Void	**updateRecievedMessages()**	Updates the received content

Table 5. P-Mail Agent Behavior Class: Member Variables

[illegible]	[illegible]	[illegible]
String	**Current_agent**	Current agent name
List<PMailMessages>	**SendList**	List of sent messages
List<PMailMessages>	**RecievedList**	List of received messages
List< String >	**associateList**	List of associate agents
HashMap<String, List<PMailMessages>>	**recievedMessagesMap**	List of all agents and their respective received messages.

4.2. Facilitator Class and P-Mail Frame Class for Design-1

4.2.1. Scenario-1 Facilitator Class

In Scenario-1, the design facilitator class is designed in such a way that it takes the client name to the sender agent as shown in table 6, which then transmits the message and returns the required address back to the client. The facilitator class also handles the process of registering the client when it comes online.

```
Class Facilitator
{
        getAgentAddress(agent) {…}

        loadAgent( ) {…}

        agentAddressInfo(agent, agentAddress) {…}
}
```

Table 6. Scenario-1 Facilitator Class: Member Functions

[illegible]	[illegible]	[illegible]
String	**getAgentAddress( )**	Gets agent address
HashMap<String, String>	**loadAgent( )**	Registers online agents
HashMap<String, String>	**getAgentAddressInfo()**	Gives agent name and address

4.2.2. Scenario-1 PMailFrame Class

Each instance of the PMailFrame class creates a PMailFrame which is a client-user interface through which it can start the process of communicating with other P-Mail clients. In order to do so, the sender P-Mail client should ask the facilitator for the address. The facilitator responds with the recipient's address. Later sender client pings the other client asking permission to send the message.

This action can be demonstrated by having the "Get Address" button on the UI. Clicking on this button requests an address from the facilitator. Once the client has the address, it can ping the recipient P-Mail to allow transmission of the message. The "Send" button on the UI would only be rendered usable if the recipient is willing to communicate. P-Mail class functions are shown in table 7 and table 8.

```
Class PMailFrame
{
        getAgentAddress(agent) {…}
        pingAgentToCommunicate(agent, agent) {…}

        respondToCommunicate(agent, agent) {…}

        pingRequestYes(true) {…}

        refresh( ) {…}

        sendActionPerformed() {…}
}
```

Table 7. Scenario-1 PMail Frame Class: Member Variables

[illegible]	[illegible]	[illegible]
String	<u>**agent_name**</u>	Current agent name
String	<u>**agentAddress**</u>	Current agent address
Int	<u>**enableSendFlag**</u>	Enables the send button
HashMap<String, String>	<u>**respondAgentMap**</u>	Responds to the agent
HashMap<String, String>	<u>**pingAgentsMap**</u>	Pings the agent.

4.3. Scenario-1. Visualization

Scenario-1 has been implemented according to the design and requirements discussed in Chapter 3. To create new agents, we must initiate the RMA GUI; open the container; and create the agent by giving inputs like agent name, arguments, and the base agent class for which the agent has to be created (Figure 17).

Table 8. Scenario-1 PMail Frame Class: Member Functions

[illegible]	[illegible]	[illegible]
Void	<u>**refreshMail()**</u>	Refreshes email messages
Boolean	<u>**pingAgentToCommunicate()**</u>	Pings the agent to communicate
Boolean	<u>**respondAgentToCommunicate()**</u>	Response about whether to communicate
TreeModal	<u>**getAssociateTreeModal()**</u>	Populates the associates
Void	<u>**sendBtnActionperformed()**</u>	Attaches the content to an object when clicking "send" and performing the desired action
Void	<u>**btnGetAddressActionPerformed()**</u>	Query facilitator for address
Void	<u>**btnPingActionPerformed()**</u>	Having the recipient's address send a ping
Void	<u>**pingReqYesActionPerformed()**</u>	Recipient willing to communicate
Void	<u>**pingReqNoActionPerformed ()**</u>	Recipient not willing to communicate
DefaultTableModal	<u>**getSendMsgs()**</u>	Populates all the sent items
DefaultTableModal	<u>**getRecievedMsgs()**</u>	Populates all received messages
Void	<u>**clearComposeFields()**</u>	Clears all the data from the compose window

Once P-Mail client1 has been created (Figure 18), we can see the user interface of client1 and client2 with available functionalities, like compose the Email message, to view the sent and received messages on the panel. The interface also has a refresh button to get a quick refresh of the content with the all the available list of agent that it can communicate with under the associate list. To send a message to client2, client1 needs to get its address. By clicking on "Get Address," it asks the facilitator, and the address field will be populated instantly.

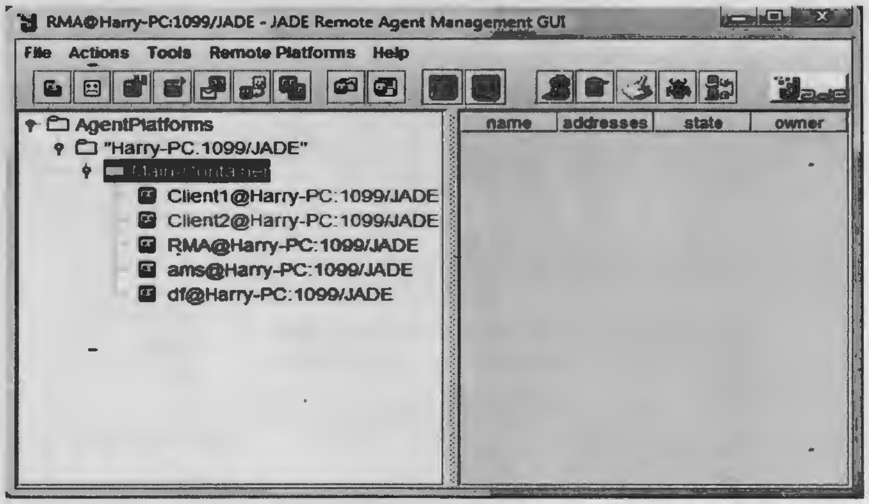

RMA@Harry-PC:1099/JADE - JADE Remote Agent Management GUI
File Actions Tools Remote Platforms Help
AgentPlatforms
"Harry-PC:1099/JADE"
Main-Container
Client1@Harry-PC:1099/JADE
Client2@Harry-PC:1099/JADE
RMA@Harry-PC:1099/JADE
ams@Harry-PC:1099/JADE
df@Harry-PC:1099/JADE
name
addresses
state
owner

Figure 17. Launch RMA GUI to Load agents.

Having the address, client1 can ping client2 with the "Ping Agent" button on the UI. This action will send a request to client2 asking for permission to communicate.

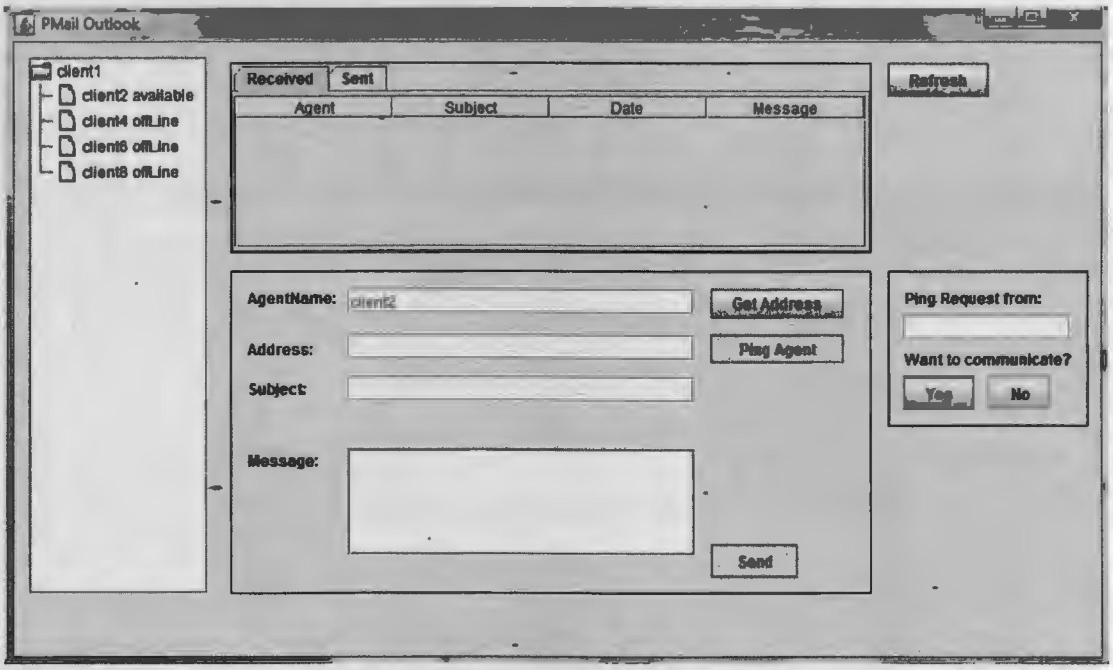


Figure 18. Scenario-1 P-Mail client1.

Once the request has been sent to client2 (Figure 19), client1's name is populated in the ping request panel to the right of the client2 interface. Client2 is then given the option to either allow or not to allow client1 to send the Email message by clicking "Yes" or "No" on the interface.

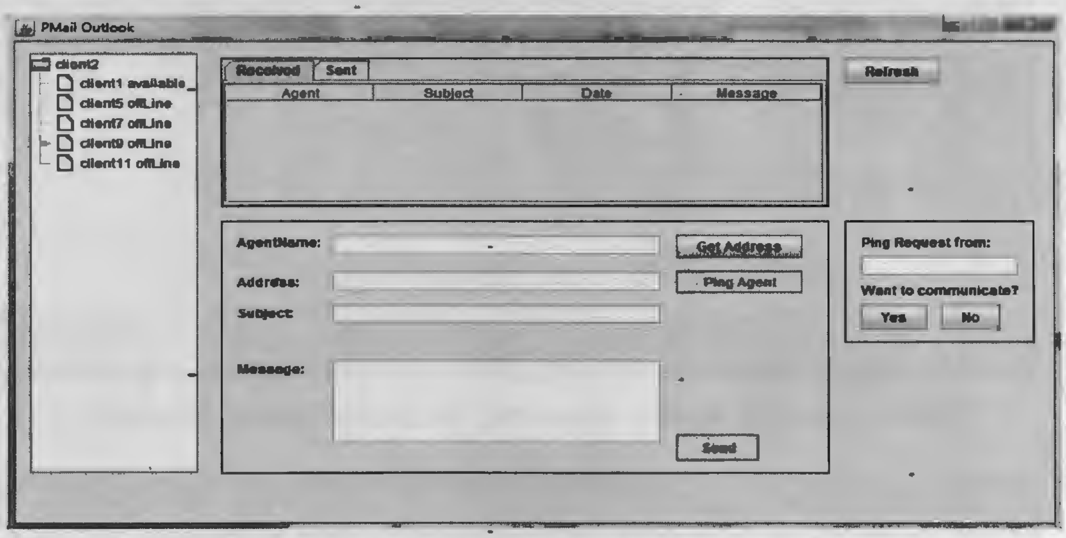


Figure 19. Scenario-1 P-Mail client2.

Client2 accepts or denies the ping request from client1 (Figure 20, Figure 21). If it accepts the request, then "send" button will be activated on the interface, and client1 can send Email to client2 (Figure 22).

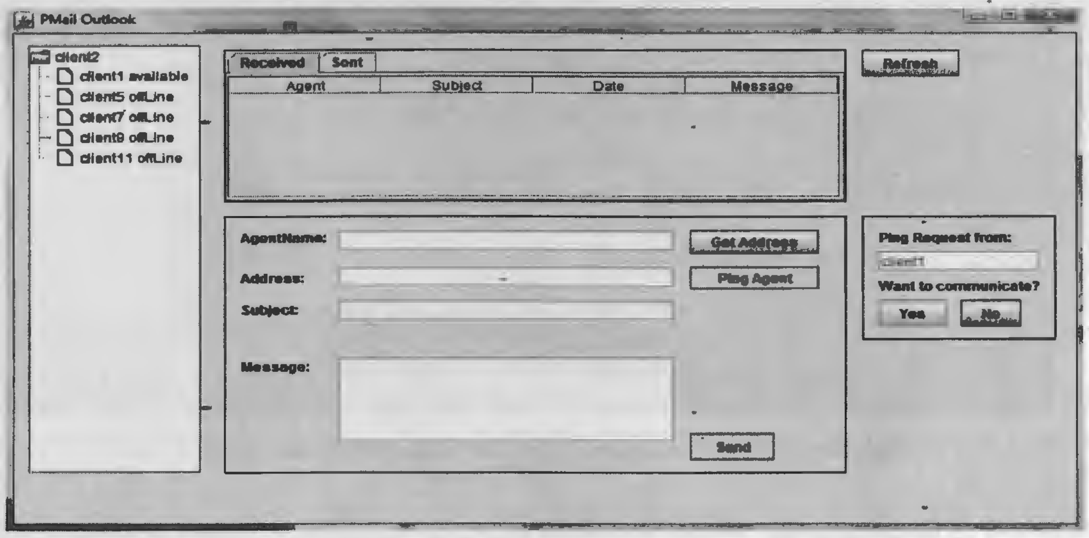
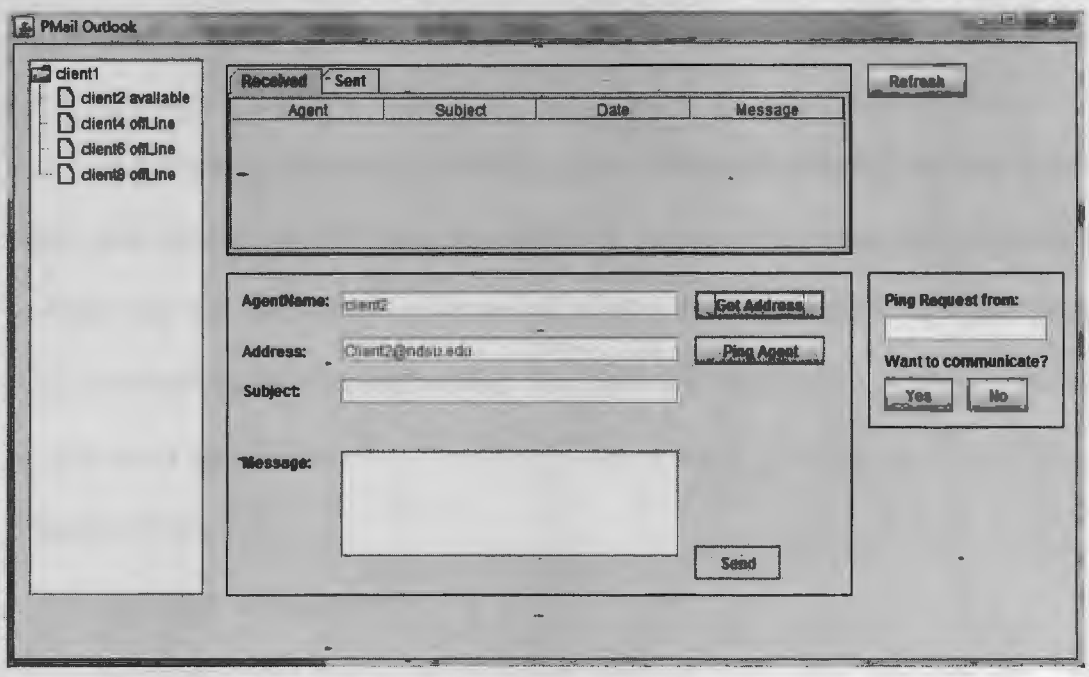


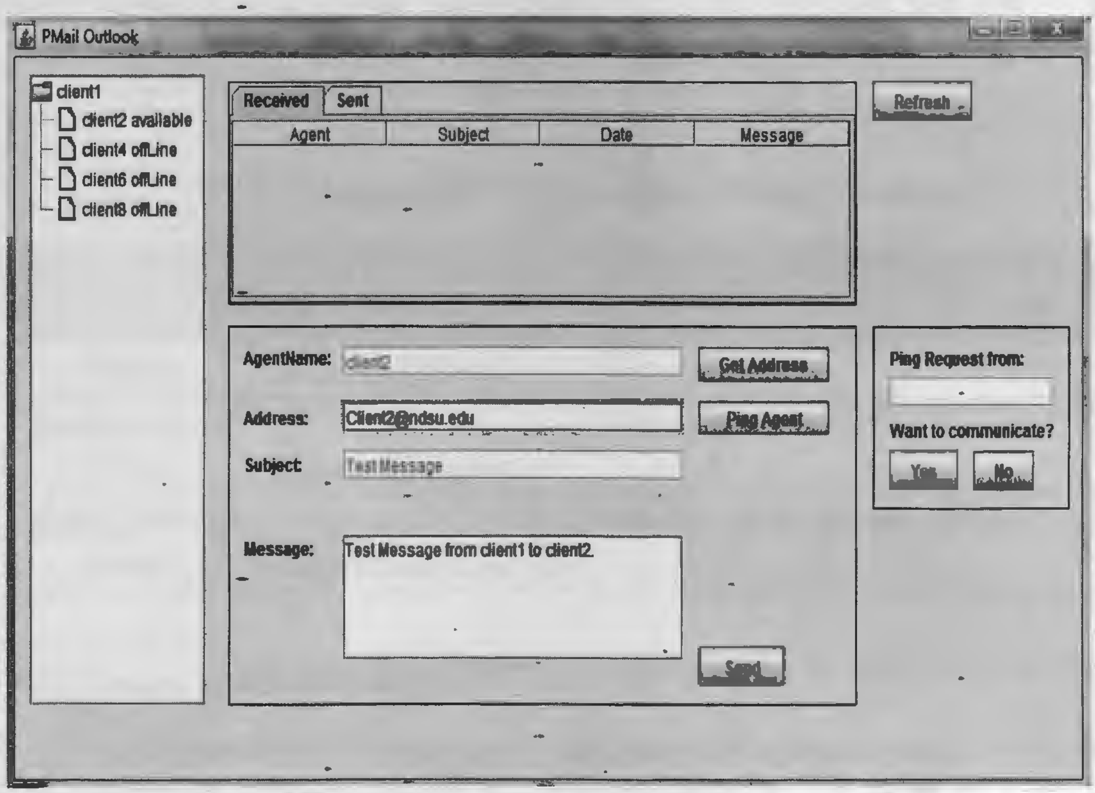
Figure 20. Scenario-1 Ping Request from client1 sent to client2.



client1
client2 available
client4 offLine
client6 offLine
client9 offLine

AgentName: client2
Address: Client2@ndsu.edu
Subject:
Message:

Figure 21. Scenario-1 client1 gets access to send message to client2.



client1
client2 available
client4 offLine
client6 offLine
client8 offLine

AgentName: client2
Address: Client2@ndsu.edu
Subject: Test Message
Message: Test Message from client1 to client2.

Figure 22. Scenario-1 client1 compose Email to client2.

4.4. Facilitator Class and P-Mail Frame Class for Designs-2

4.4.1. Scenario-2 Facilitator Class

In Scenario-2, the design facilitator class is designed in such a way that it takes the client name to the sender agent that wants to transmit the message. According to the received response, facilitator checks the recipient's address and, in turn, sends the request to the recipient on behalf of the sender. The facilitator class shown in table 9 also handles the process of registering clients when they come online.

```
Class Facilitator
{
        getAgentAddress(agent) {…}

        loadAgent( ) {…}

        agentAddressInfo(agent, agentAddress) {…}

        communicateRecipientAgent(agent, agent) {…}
}
```

Table 9: Scenario-2 Facilitator Class: Member Functions

[illegible]	[illegible]	[illegible]
String	<u>**getAgentAddress( )**</u>	Gets agent address
HashMap<String, String>	<u>**communicateAgent( )**</u>	Pings the recipient agent
HashMap<String, String>	<u>**getAgentAddressInfo()**</u>	Gives the agent name and address

4.4.2. Scenario-2 PMailFrame Class

Each instance of the PMailFrame class creates a PMailFrame which is a client-user interface through which client can start the process of communicating with other P-Mail clients. The P-Mail client asks the facilitator for the recipient's address. The facilitator receives the request from the sender. The facilitator then sends a request stating a P-Mail's willingness to communicate. All the class member variables and methods are mentioned in table 10 and table 11.

The same can be demonstrated by having the "GoFac & PingAgent" button on the UI. Clicking this button asks the facilitator for the address. Once the facilitator has the request, it can ping the recipient's P-Mail to allow message transmission. The "Send" button on the UI would only be rendered usable if the recipient is willing to communicate.

```
Class PMailFrame
{
        goFacilitatorToPingAgent(agent) {…}

        pingAgentToCommunicate(agent, agent) {…}

        respondToCommunicate(agent, agent) {…}

        pingRequestYes(true) {…}

        sendActionPerformed() {…}

}
```

Table 10. Scenario-2 PMail Frame Class: Member Variables

[illegible]	[illegible]	[illegible]
HashMap<String, String>	**respondAgentMap**	Responds to the agent
HashMap<String, String>	**pingAgentsMap**	Pings the agent.
String	**agent_name**	Current agent name
Int	**enableSendFlag**	Enable the send button

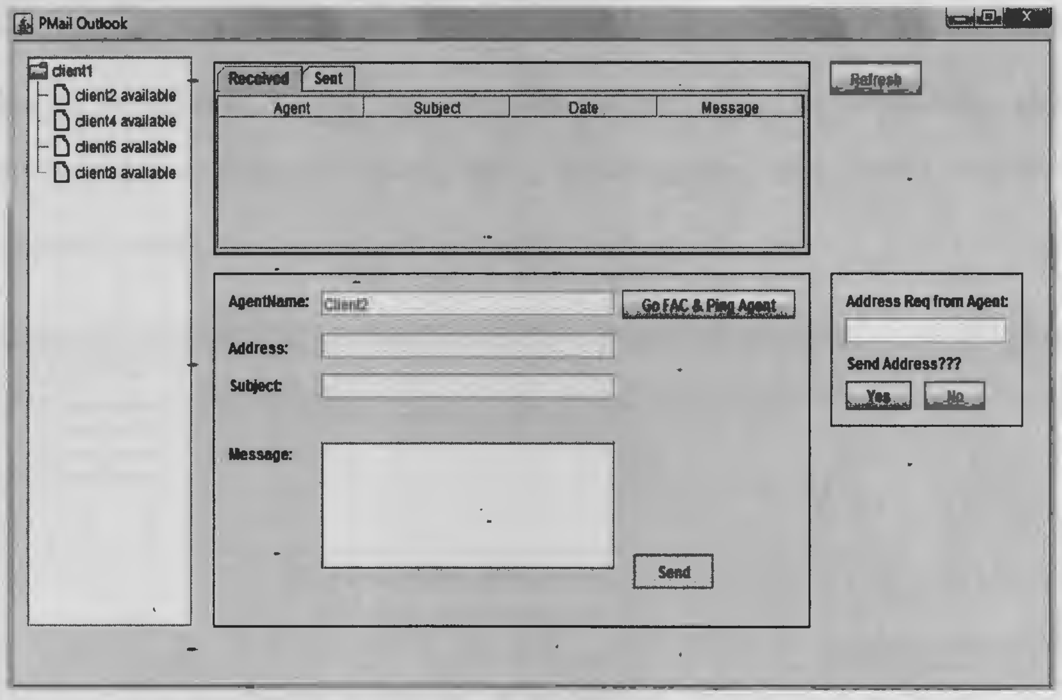
Table 11. Scenario-2 PMail Frame Class: Member Functions

Void	**refreshMail()**	Refreshes email messages
Boolean	**pingAgentToCommunicate()**	Pings the agent to communicate
Boolean	**respondAgentToCommunicate ()**	Response about whether to communicate
TreeModal	**getAssociateTreeModal()**	Populates the associates
Void	**sendBtnActionperformed()**	Attaches content to the object when "send" is clicked and performs the desired action
Void	**btnGoFacToPingAgentActionPerformed()**	Queries the facilitator for an address
Void	**btnPingActionPerformed()**	Having the recipient's address send a ping
Void	**pingReqYesActionPerformed()**	Recipient willing to communicate
Void	**pingReqNoActionPerformed ()**	Recipient not willing to communicate
DefaultTableModal	**getSendMsgs()**	Populates all the sent items
DefaultTableModal	**getRecievedMsgs()**	Populates all received messages
Void	**clearComposeFields()**	Clears all data from the compose window

4.5. Scenario-2. Visualization

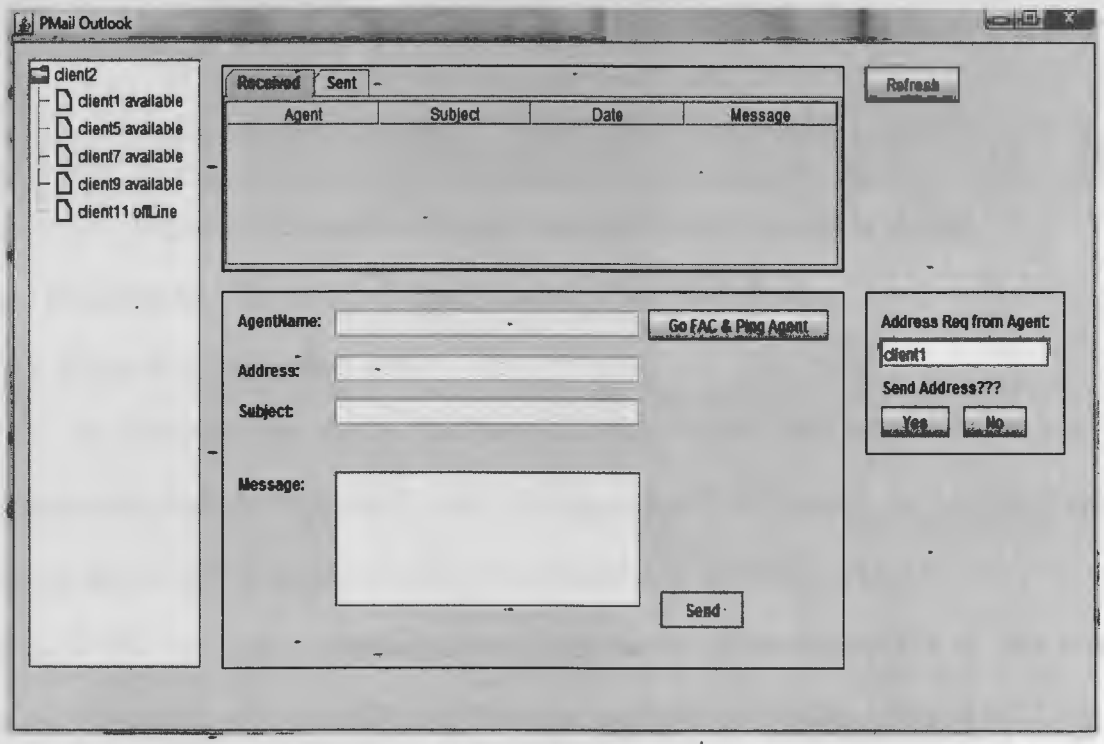
Scenario-2 has been successfully implemented according to the design and requirements. Client1 and client2 are launched accordingly by the RMA user interface. Both clients have the option of using "Go FAC & Ping Agent" on the interface.

By client1 clicking "Go FAC & Ping Agent" (Figure 23), the request will be sent to the facilitator querying for client2's address. The facilitator then sends a request for communication to client2. When client2 has the request for an address from a facilitator (Figure 24), the address field will be populated with the client1 name.



AgentName: Client2

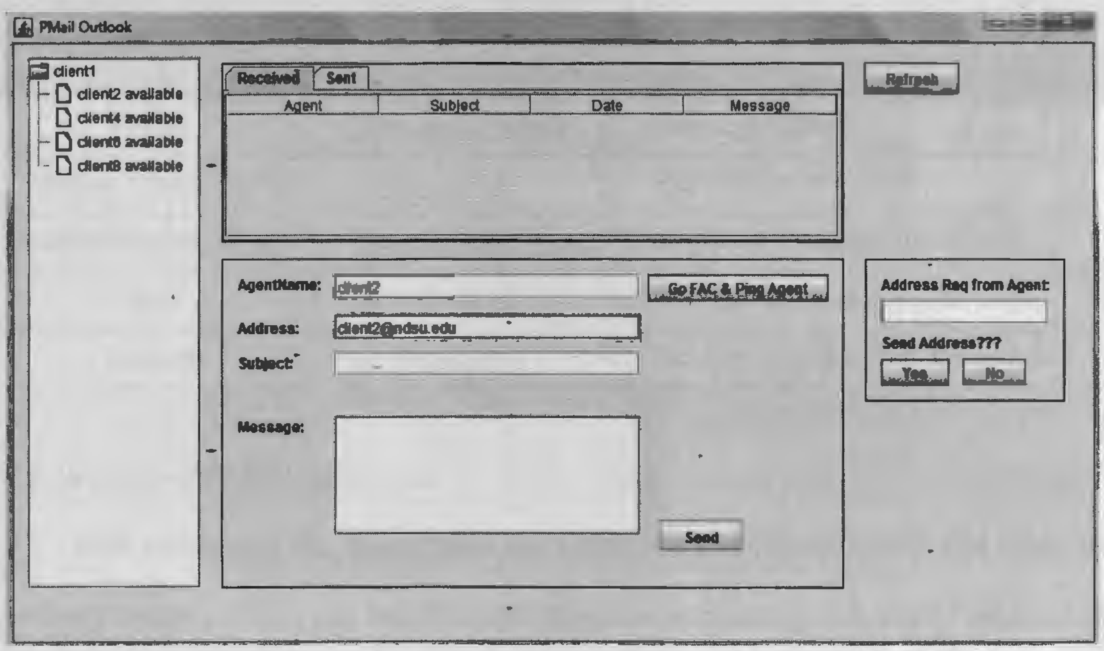
Figure 23. Scenario-2 client1.



Address Req from Agent: client1

Figure 24. Scenario-2 client2 has the request from client1.

Upon receiving a request from the facilitator, client2 can accept or reject the request by just clicking the Yes/No buttons provided. When client2 opts to communicate with client1, client2 will send its address directly to client1 in response (Figure 25), and the send button on the interface will be activated, which allows client1 to successfully send an Email message to client2. If client2 denies communication, then client1 cannot send messages to client2.



AgentName: client2

Address: client2@ndsu.edu

Subject:

Message:

Figure 25. Scenario-2 client1 composes Email to send to client2.

4.6. Facilitator Class and P-Mail Frame Class for Design-3

4.6.1. Scenario-3 Facilitator Class

In Scenario-3, the design facilitator gets the request from the sender's P-Mail to communicate with the recipient P-Mail. Having received the request, the facilitator sends another request to the recipient asking for permission to allow the sender to send an Email message. The recipient's acceptance/denial response to communicate will be sent to the sender's P-mail through the facilitator. Member functions can be seen in the table 12.

```
Class Facilitator
{
        getAgentAddress(agent) {...}
        loadAgent( ) {...}
        communicateAgent(agent, agent) {...}
        setResponse(agent, 1) {...}
}
```

Table 12. Scenario-3 Facilitator Class: Member Functions

String	<u>**getAgentAddress( )**</u>	Gets agent address
HashMap<String, String>	<u>**loadAgent( )**</u>	Registers online agents
HashMap<String, String>	<u>**communicateAgent()**</u>	Sends the request to the recipient
Void	<u>**setResponse()**</u>	Sets the response from the recipient
Boolean	<u>**getResponse()**</u>	Gets the response from the recipient

4.6.2. Scenario-3 PMailFrame Class

Each instance of the PMailFrame class creates a PMailFrame which is a client-user interface through which it can start the process of communicating with other P-Mail clients. In order to do so, the sender P-Mail client should ask the facilitator for the address. The facilitator responds to the sender's P-Mail with the recipient's decision about whether to start the communication process.

This action can be demonstrated by having the "FAC – Process Request" button on the UI. Clicking this button asks the facilitator for the address. Using the agent name service, the facilitator processes the request through the recipient and gets a decision about whether to communicate or deny the response. The same decision is sent to the sender's P-

Mail to enable/disable the send button on the interface. Member variables and member functions in table 13 and table 14.

```
Class PMailFrame
{
        facilitatorCommunicateAgent(agent, agent) {…}

        respondToCommunicate(agent, agent) {…}

        pingRequestYes(true) {…}

        sendActionPerformed() {…}

}
```

Table 13. Scenario-3 PMail Agent Behavior Class: Member Variables

[illegible]	[illegible]	[illegible]
String	**agent_name**	Current agent name
Int	**enableSendFlag**	Enables the send button
HashMap<String, String>	**pingAgentsMap**	Pings the agent.

According to the design and requirements discussed earlier, scenario-3 has been implemented. The created agent, client1 (Figure 26), and the client2 interface now have the "FAC – Process Req" button. If client1 decides to send an Email to client2, client has to click the button, and immediately, a query would be sent to the facilitator to get the address for client2.

Table 14. Scenario-3 PMail Frame Class: Member Functions

[illegible]	[illegible]	[illegible]
Void	**refreshMail()**	Refreshes email messages
Boolean	**pingAgentToCommunicate()**	Pings the agent to communicate

Boolean	respondAgentToCommunicate()	Response about whether to communicate
TreeModal	getAssociateTreeModal()	Populates the associates
Void	sendBtnActionperformed()	Attaches content to the object when the send button is clicked and performs the desired action
Void	facProcessReqActionPerformed()	Queries the facilitator for an address
Void	btnPingActionPerformed()	Has the recipient's address send a ping
Void	pingReqYesActionPerformed()	Recipient willing to communicate
Void	pingReqNoActionPerformed ()	Recipient not willing to communicate
DefaultTableModal	getSendMsgs()	Populates all sent items
DefaultTableModal	getRecievedMsgs()	Populates all received messages
Void	clearComposeFields()	Clears all data from the compose window

4.7. Scenario-3. Visualization

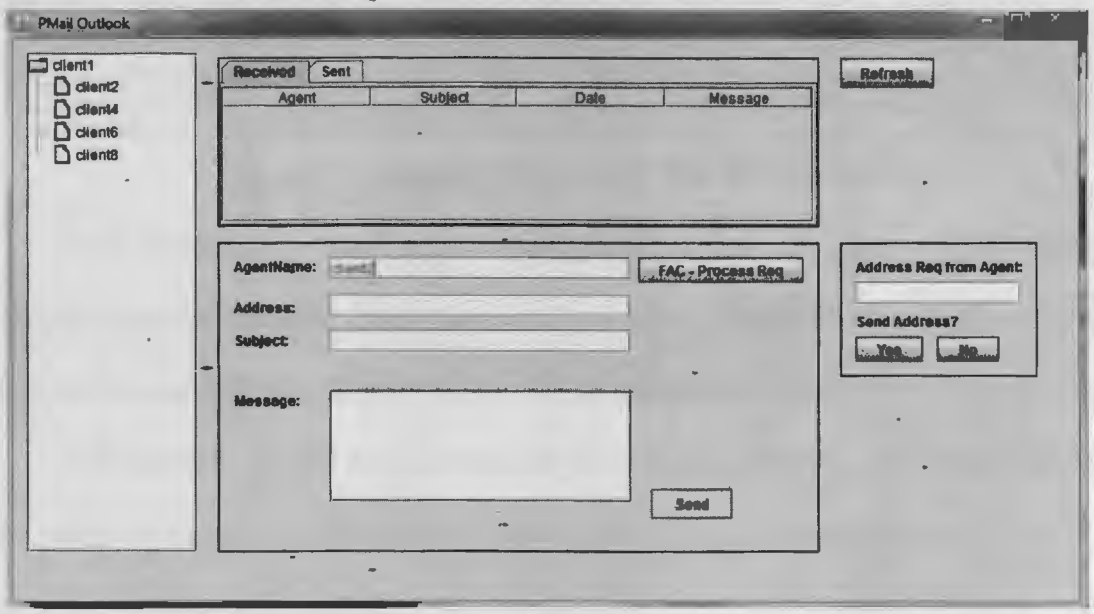
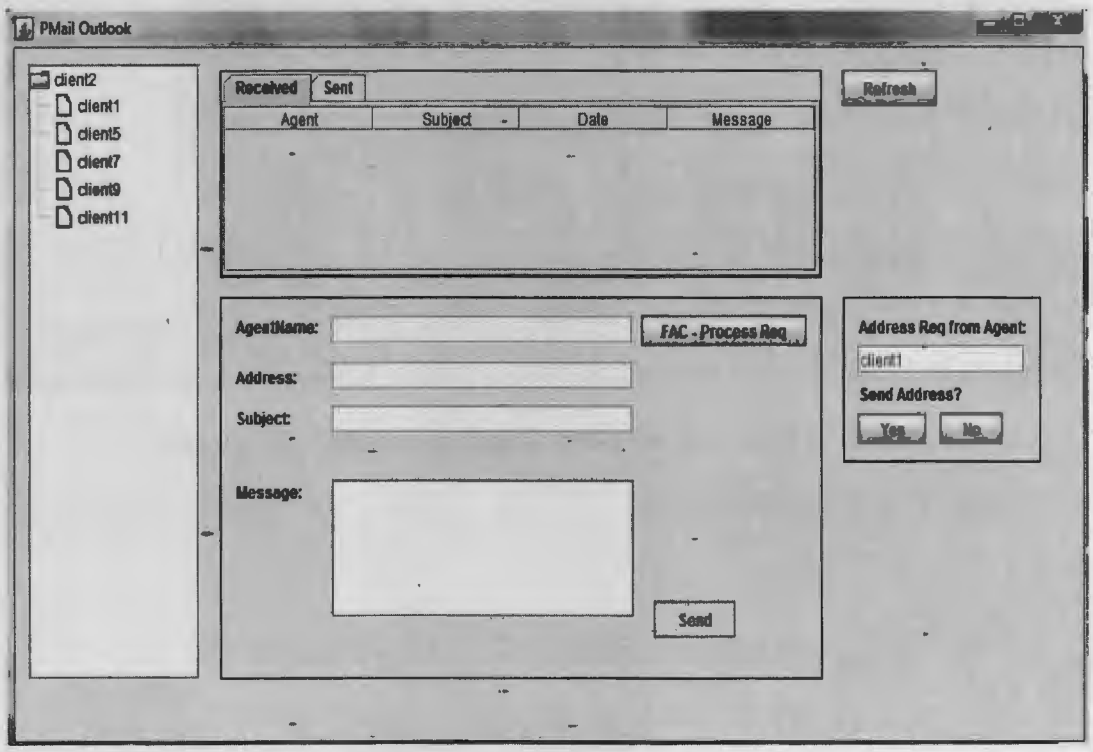


Figure 26. Scenario-3 for client1.

Once the facilitator receives the request for an address from client1, it processes the request by handing it to client2 (Figure 27). Then, client2 has to make a decision about accepting or rejecting the facilitator's request. Whatever client2 decides, it passes the response on to facilitator. Once the request has been accepted/denied, the facilitator transfers the same response to client1. Client2 can either click the "Yes" button to accept or the "No" button to reject the proposal to communicate with client1.

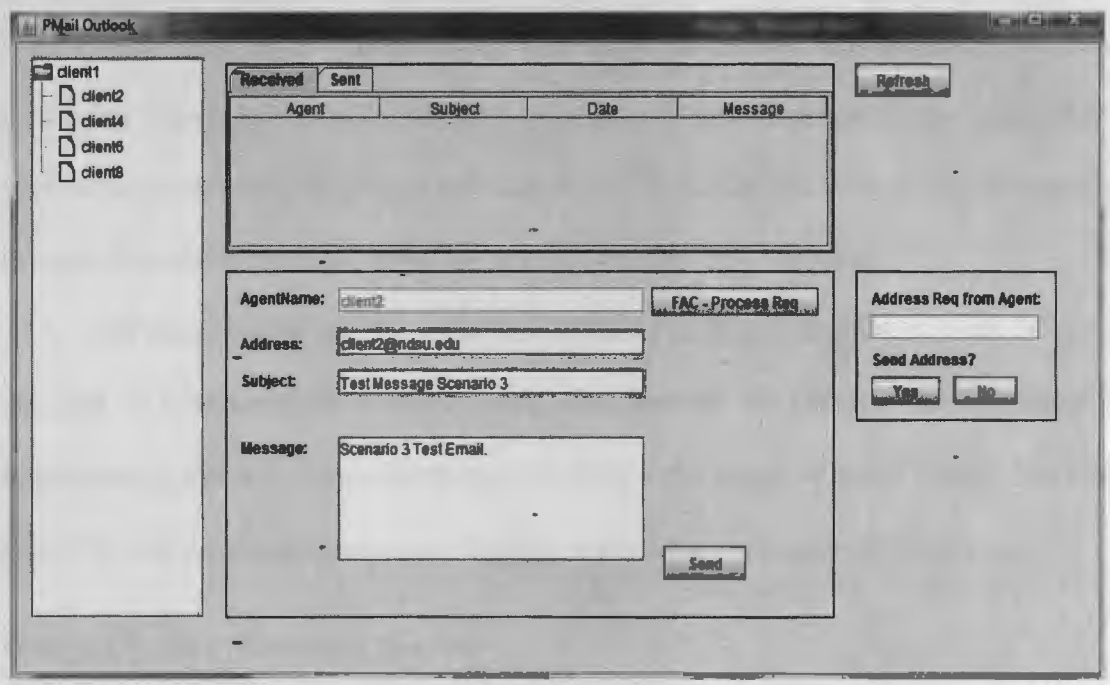


Address Req from Agent: client1

Figure 27. Scenario-3 for client2 with the request.

With client2's acceptance of the request from the facilitator, client1 is now allowed to start successfully sending Email messages to client2 (Figure 28). If client2 rejects the facilitator's request, client1 is never able to send a message to client2.

In Chapter 5, we will be analyzing the performance, security, and privacy issues of each design and will understand the way these systems react to security attacks. This is an attempt to verify if these designs meet the goals and objectives behind developing the P-Mail messaging system.



AgentName: client2

Address: client2@ndsu.edu

Subject: Test Message Scenario 3

Message: Scenario 3 Test Email.

Figure 28. Scenario-3 client1 compose an Email to client2.

5. ANALYSIS

In this chapter, we will analyze the three design models of the P-Mail messaging system that have been implemented. The purpose behind implementing the P-Mail messaging system is to provide security, privacy, and authentication for the users. This has been achieved by using intelligent software agents to collaborate with other software agents in order to transfer messages from one client to another on a network.

All three designs use the intelligent P-Mail Frame and facilitator agents (Figure 29, 30, and 31) accordingly without using any servers to transmit the messages. By implementing these designs effectively, we reduce the scope of using servers like SMTP and POP, and we avoid unnecessary backup of the Email messages on the servers.

Design-1 P-Mail Messaging System:

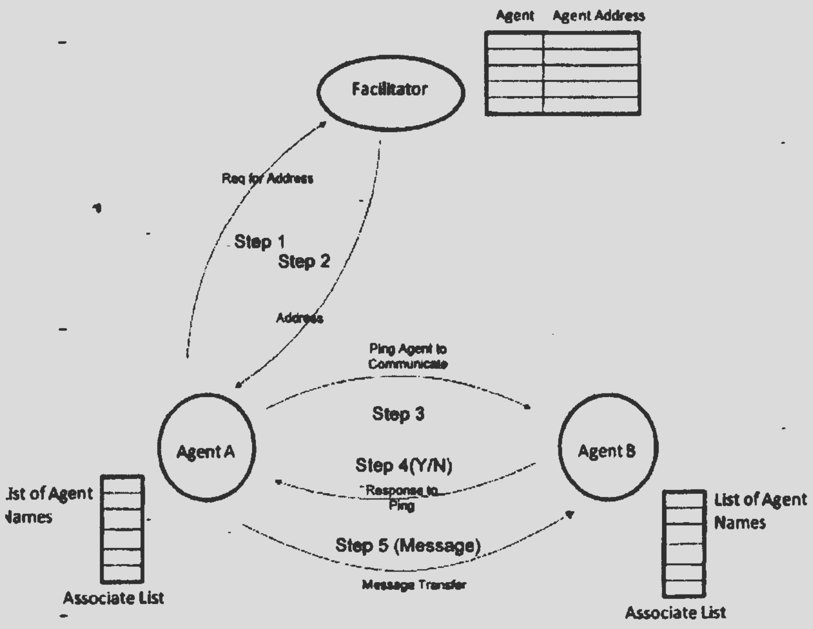


Figure 29. P-Mail Messaging System Design-1.

Design-2 P-Mail Messaging System:

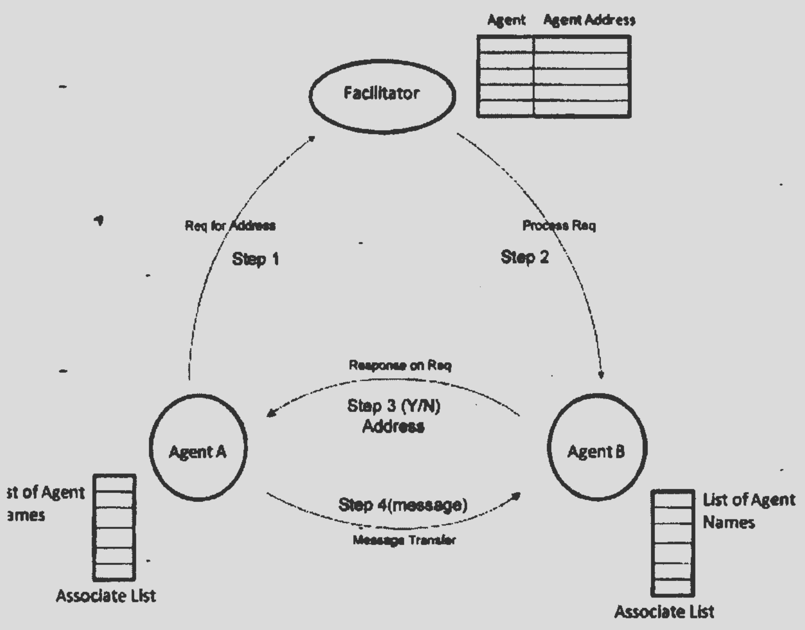


Figure 30. P-Mail Messaging System Design-2.

Design-3 P-Mail Messaging System:

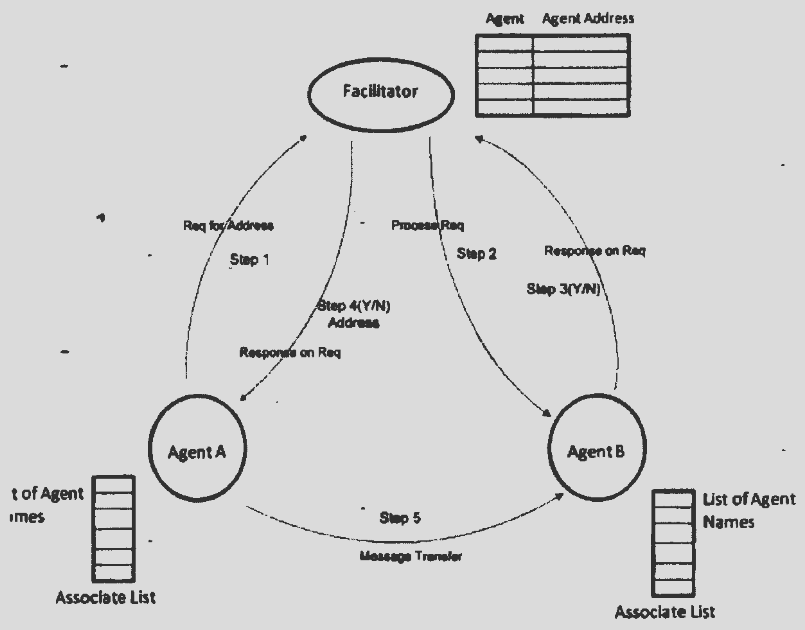


Figure 31. P-Mail Messaging System Design-3.

5.1. Advantages of the P-Mail Messaging System

The primary advantages that all three designs provide are as follows:

- The Email messages are never stored on any machine, and they will not use any servers, such as SMTP or a POP server, to transmit messages.
- The facilitator agent provides the agent name service to P-Mail agents.
- Messages will be transferred only between P-Mail client to another P-Mail client.

5.2. Message Traffic through a Facilitator

Each of the P-Mail messaging system design makes use of the facilitator agent. In all three designs, the sender P-Mail client would query for the recipient's address. Facilitator does its job by responding in a desired way, making good use of the agent name service. Message traffic through a facilitator would definitely affect the entire system's performance. Lesser the message traffic better the system response; this is because, when the facilitator starts registering a huge number of agents with itself, the message traffic would definitely affect the system while querying. Designs 1 and 2 have less message traffic compared to Design-3 of the P-Mail messaging system (Figure 32).

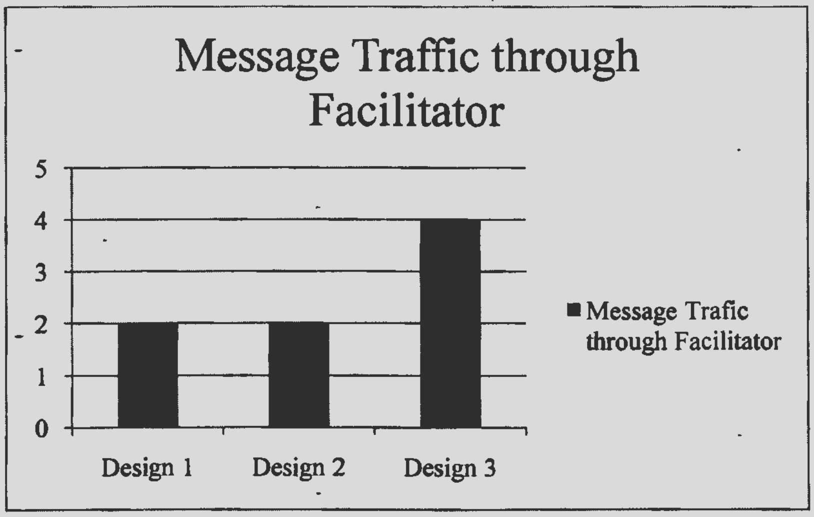


Figure 32. Message Traffic through the Facilitator

5.3. Number of Process Steps

Process steps in a design are all about the number of steps that the design needs to be processed or how long the process of sending a message from one P-Mail client to another P-Mail client is. The better design can be justified on the number of process steps it has to go through to send a message on the network.

To successfully send the message, the number of process steps in Design 2 is less when compared to Designs 1 and 3.

At the same time, the number of process steps in Design 2 is less when compared to Designs 1 and 3 even if the recipient's P-Mail is not willing to communicate. These graphs clearly suggest that, if the message has been transferred successfully or if it is unsuccessful, Design 2 does a good job.

Figures 33 and 34 illustrate the number of successful and unsuccessful steps to process the message transfer from the sender to the recipient.

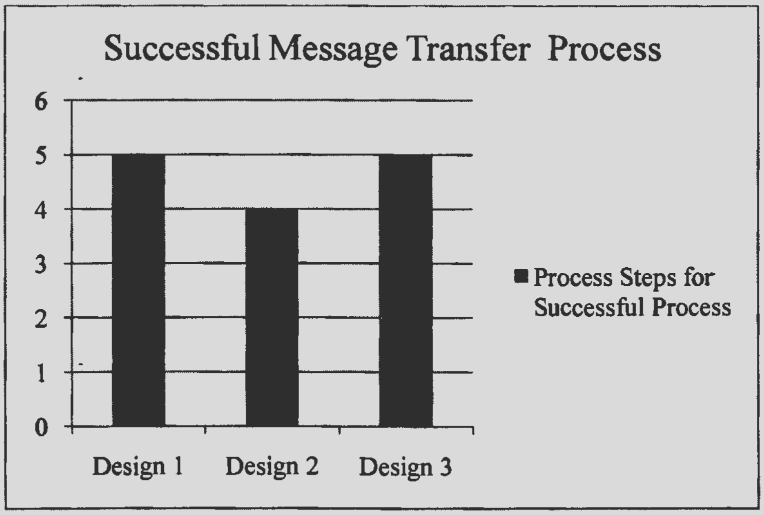


Figure 33. Process Steps in a successful process.

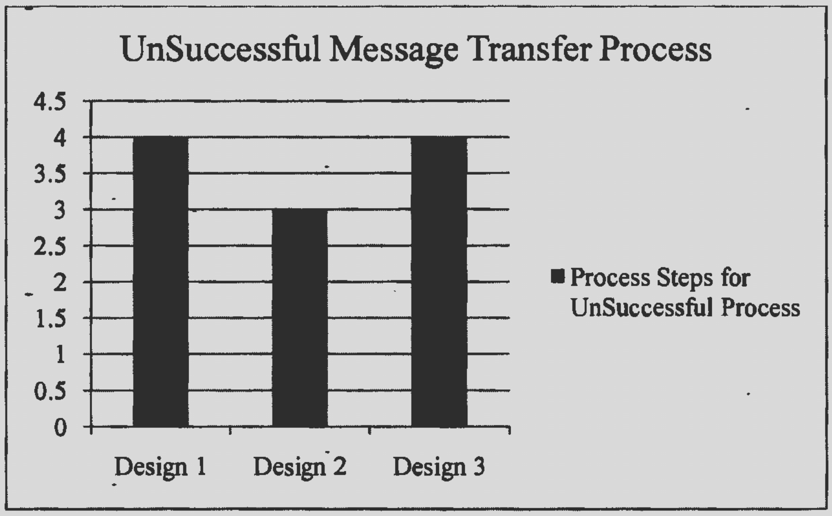


Figure 34. Process Steps in an Unsuccessful process.

5.4. Recipient Address to Communicate

In either of the designs, if any P-Mail client has to send a message or an Email, it has to have the recipient's address, so it queries the facilitator. It does matter at what point in the process the sender P-Mail client gets the recipient's address. Two important factors need to be considered:

- At what step the sender has the recipient's address
- Status of recipient: whether the recipient is willing to communicate

In Designs 2 and 3, the facilitator responds with the recipient's address to sender P-Mail client only shown in the table 15 with the permission of recipient P-Mail whether or not it is willing to communicate. With Design 1, the facilitator responds to the sender's P-Mail without any concern to recipient's status whether it is willing to communicate or not. From these analyses, it is always better to go with Designs 2 or 3 rather than Design 1.

Table 15. Recipient Address to communicate

Design 1	Before recipient decides to communicate
Design 2	Up on recipient's decision to communicate
Design 3	Up on recipient's decision to communicate

5.5. Security Attacks and Encryption

In other words, the P-Mail messaging system is a network with a flow of messages between P-Mail clients and the facilitator agent. In a physical network, there is always a chance of intruders trying to break through the network, causing issues about privacy and security. The chances of security attacks are always directly proportional to the number of process steps involved in sending the message or Email from one P-Mail client to another. There comes a need for encrypting the messages or the Email content on a network. There are several kinds of attacks that can potentially happen on the P-Mail messaging system. PGP, or Public key encryption, is always better to use when there is a request/response process happening between the P-Mail clients and the facilitator. There is always a chance of intrusion or security attacks on the P-Mail messaging system.

If we plan on encrypting the entire P-Mail messaging system to make it more secure from intruders, we need to encrypt almost all process steps that are being processed on the network to transmit the message. Therefore, encryption would be directly dependent on the number of process steps involved in the design. With a closer look at the design, we can decide that Design 2 requires fewer process steps to be encrypted when compared to other designs (Figure 35).

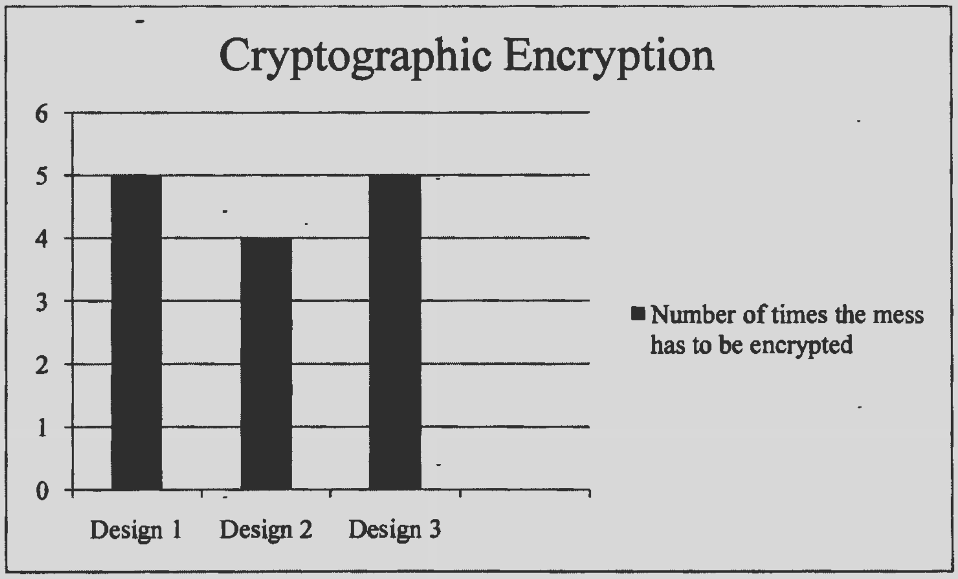


Figure 35. Cryptographic Encryption on a P-Mail Messaging System.

5.5.1. Security Attacks

Security attacks can be classified based on the effect they cause on the network. By assessing the effects of these attacks on the system, we will decide which design is most and least vulnerable. Most of these attacks are directly proportional to the process steps in the design [12]. Table 16 shows the significance of each attack on the designs.

Jamming: Jamming in P-Mail scenarios can be defined as the effect on the network caused by blocking the request/response from processing between agents because of the adversary. Among all three designs, when an adversary interrupts the process flow, there are good chances of jamming. The more process steps involved to process on the network, the greater the chances of jamming. Design 2 is much safer to employ in such cases because of its simple and straightforward design approach.

Tampering: This can be defined as an intruder making an attempt to break through the workflow process/process steps by compromising Email content on the network. For these kinds of attack, it is better to employ a design with fewer processing step carrying the private/more secured information. In Design-1, we have the recipient's address, which has to be kept private, at step 2, and in design 3, this happens at steps 3 and 4. Design 2 is much better in this scenario.

Collision: In P-Mail messaging, collision can be defined as the process of interception caused between request to request or request to response from the agents causing network disruption.

Manipulate the Routing Information: This term can be defined as the process of manipulating the network workflow process to give misleading information to the agents.

Table 16. Security Attacks and Their Effects on Each Design

[illegible]	[illegible]	[illegible]	[illegible]
Jamming	Good chances of occurrence	Fewer chances of occurrence when compared to other designs	Can happen quite often
Tampering	Most vulnerable	Less vulnerable	Most vulnerable
Collision	Most vulnerable	Less vulnerable	Most vulnerable
Manipulating Routing	Fewer chances of occurrence	Fewer chances of occurrence	Less chances of occurrence
Clone attack	No significant effect on the design	No significant effect on the design	No significant effect on the design
Encryption	High number of steps to encrypt when compared to other designs.	Fewer steps to encrypt when compared to other designs	High number of steps to encrypt when compared to other designs.

Clone Attack: In our scenarios, these kinds of attacks can be described as introducing external agents which are clones of other existing agents to the network. If this situation occurs on a network, it does not show any significant effect on system performance because, in order to interrupt the process, it has to register with the facilitator, and the facilitator does not allow an agent clone to register with it.

6. FUTURE WORK AND CONCLUSION

6.1. Future Work

Private Mail messaging system makes use of Facilitator to provide the access control to the clients and also takes care of agent name service. Facilitator plays a crucial role in the implementation of P-Mail messaging system. We definitely need to consider the message traffic through the Facilitator during the process. There's scope to improve the Facilitator design to handle the message traffic.

Depending up on the message traffic and the number of clients trying to use the Facilitator to send the message we could classify the Facilitator design to be either load based or distributed architectural design.

The proposed idea is more suitable to the private club communication process. We can develop a design making use of Facilitator and P-Mail clients to support traditional way of communication process

6.2. Conclusion

In the present world with modern technology and communication growing at a rapid pace, Email messages have been questioned for their privacy and security issues while transmitting-messages from one user to another user residing on a network. These Email clients use a wide variety of protocols and servers to transmit the messages which are vulnerable to intruders, and the unnecessary backup of data may cause an issue at certain times.

This paper proposed a technique for designing an Email messaging system modeled on intelligent software agent collaboration. We have discussed and analyzed three potential designs to implement the messaging system according to the user's requirements. The objectives achieved by implementing the P-Mail messaging system are that the messages are no longer stored on the servers, making the minimal use of protocols which might be a serious concern for security and privacy issues. The P-Mail messaging system ensures the secure transfer of messages between the users who belong to the pool of members in a private club on the network. The P-Mail messaging systems ensure the secure transfer of messages on a network from the sender to the recipient. In the analysis chapter, we discussed the pros and cons of each design and learned that Design-2 is less vulnerable to security attacks and is more secure when compared to Designs-1 and 3. There is less message traffic through the facilitator in Design-2 compared to the other two designs, and the facilitator acts much more intellectually in handling requests from the sender and provides access control to the P-Mail client on the system.

REFERENCES

1. Ruth Aylett1, Frances Brazier2, Nick Jennings3, Michael Luck4, Hyacinth Nwana5, and Chris Preist, "Agent Systems and Applications", Volume 13 , Issue 3, Year of Publication: 1998, ISSN: 0269-8889
2. Erik Kangas, PhD, President of LuxSci "The Cases for Email Security", Posted Friday, March 13th, 2009

   http://luxsci.com/blog/the-case-for-email-security.html
3. Jeffrey M. Bradshaw, "An Introduction to Software Agents – Agents & the User Experience", Year of Publication: 1997, ISBN:0-262-52234-9

http://agents.umbc.edu/introduction/01-Bradshaw.pdf

4. Reticular Systems, Inc, "P-Mail – Private Email and Instant Messaging with Intelligent Software Agents", Version 1.0 Rev.0 August 27, 1999, San Diego

   http://www.agentbuilder.com/Documentation/PMail/pmail.pdf
5. Reticular Systems, Inc, "Agent Builder, P-Mail: A Private Mail and Instant Messaging System", Version 1.0 Rev.0 August 27, 1999, San Diego

   http://www.agentbuilder.com/Documentation/PMail/index.html
6. CW Preist, Economic Agents for Automated Trading, in 'Software Agents for Future Communications Systems', ed A. Hayzelden & J.Bigham, Springer 1999
7. Fred W. Atkinson, III , "PGP Encryption", Johns Hopkins University, Network Security, April-1996

   http://www.mishmash.com/fredspgp/pgp.html

8. Hyacinth S. Nwana, "Software Agents: An Overview", Advanced Applications & Technology Department, Cambridge University, Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept 1996.

http://www.sce.carleton.ca/netmanage/docs/AgentsOverview/ao.html

9. Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB, formerly CSELT), Giovanni Rimassa (University of Parma), "JADE programmers guide", last update:18-June-2007.

10. Martin L. Griss, "Software Agents as Next generation Software Components", Software Technology Laboratories, Palo Alto, CA

11. Mohammad Mannan, P.C. van Oorschot, "Secure Public Instnat Messaging: A Survey", School of Computer Science, Carleton University, Ottawa.

12. Kendall. Nygard, "Security Attacks on Sensor Networks and Embeded Systems", Computer Science Dept, North Dakota State Univeristy, ND.

APPENDIX

In this section we will look into details of how the P-Mail messaging systems have been developed in programming point of view, based on three designs discussed in Chapter-3. We will go through all the important classes which are part of the implementation and get into their details.

MailAgent.java

```
package jade.pmail;
import jade.core.Agent; //Importing Agent Class from JADE Libraries
import jade.pmail.gui.PMailFrame;
import jade.pmail.util.Facilitator;

public class MailAgent extends Agent {
   protected void setup() {//Create Agent
      Object[] args = getArguments(); //Retrive all arguments passed to Agent
      Facilitator fac = new Facilitator();
         if (args != null) {
         System.out.println("Passed Arguments are:");//Print all Arguments
         for (int i = 0; i < args.length; ++i)
         {
            PMailFrame frame = new PMailFrame(args[i].toString());
            fac.agentAddressInfo(args[i].toString(), getName());
            frame.setVisible(true);
         }
      }
      doDelete();
   }// Terminate Agent
      protected void takeDown() {//Do Clean Up Process
      }
}
```

PMailAgentBehaviour.java

```
/*

 * To extend PMail Action Behavior.

 */

package jade.pmail.util;


import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;


/**
 * @author Hari Mukka
 */
public class PMailAgentBehaviour {
   private String current_agent;
   private List<String>  associatesList;
   public PMailAgentBehaviour(String agent) {
      this.current_agent = agent;
      this.associatesList = new LoadAgents().loadAgent().get(current_agent);
   }

   private List<PMailMessages> sendList = new ArrayList<PMailMessages>();
   private static List<PMailMessages> receivedList = new ArrayList<PMailMessages>();
   private static HashMap<String, List<PMailMessages>> receivedMessagesMap = new
HashMap<String, List<PMailMessages>>();

   public String sendMessage(String msg, String toAgent, String fromAgent, String date,
String subject){
      String authentication = "";
      if(authenticateAgent(toAgent)){
         authentication = "The agent is authenticated!!";
```

```
            PMailMessages sendObj = new PMailMessages();
            sendObj.clearAll();
            sendObj.setToAgent(toAgent);
            sendObj.setFromAgent(fromAgent);
            sendObj.setMessage(msg);
            sendObj.setDate(date);
            sendObj.setSubject(subject);
            sendList.add(sendObj);
            updateReceiveMessages(toAgent, sendObj);
        }else{
            authentication = "Agent authentication failed!!";
        }
        return authentication;
    }

    public HashMap<String, String> pingAgent(String toAgent, String address, String
fromAgent) {
        HashMap<String, String> pingMap = new HashMap<String, String>();
        pingMap.put(toAgent, fromAgent);
        return pingMap;
    }
    public boolean authenticateAgent(String associateAgent) {
        //System.out.println("To agent2: "+associateAgent);
        //System.out.println(associatesList.size());
        if(associatesList != null && associatesList.contains(associateAgent))
            return true;
        else
            return false;
    }
    public List<PMailMessages> getSendList() {
        return sendList;
```

```
  }
  public List<PMailMessages> getReceivedList(String agent) {
    return receivedMessagesMap.get(agent);
  }
   public void updateReceiveMessages(String toAgent, PMailMessages message){
     receivedList.add(message);
     receivedMessagesMap.put(toAgent, receivedList);
   }
  public HashMap<String, List<PMailMessages>> getReceivedMap() {
    return receivedMessagesMap;
  }
}
}
```

PMailActions.java

```
package jade.pmail.util;
import java.util.Calendar;
import java.util.GregorianCalendar;

/***
 * @author Hari Mukka
 */
public class PMailActions {

  public String compose(String msg, String to, String from, String sub,
PMailAgentBehaviour mail) {
     Calendar cal = new GregorianCalendar();
     String date = cal.getTime().toString();
     return mail.sendMessage(msg, to, from, date, sub);
  }
}
```

PMailMessages.java

```
package jade.pmaiI.util;


import java.util.Date;


/***
 * @author Hari Mukka
 */
public class PMailMessages {
   private String message;
   private String toAgent;
   private String fromAgent;
   private String date;
   private String subject;


   public String getMessage(){
      return message;
   }
   public void setMessage(String message){
      this.message=message;
   }
   public String getToAgent(){
      return toAgent;
   }
   public void setToAgent(String toAgent){
      this.toAgent = toAgent;
   }
   public String getFromAgent(){
      return fromAgent;
   }
   public void setFromAgent(String fromAgent){
        this.fromAgent = fromAgent;
    }
    public String getDate(){
        return date;
    }
    public void setDate(String date){
        this.date = date;
    }
    public String getSubject(){
        return subject;
    }
    public void setSubject(String subject){
        this.subject = subject;
    }
    public void clearAll() {
        this.subject = null;
        this.date = null;
        this.toAgent = null;
        this.message = null;
    }
}
```

Scenario-1. Facilitator.java

```
package jade.pmail.util;


import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
/***
 * @author Hari Mukka
 */
```

```
public class Facilitator {
   public String getAgentAddress(String agent){


      HashMap<String, String> map=loadAgent();
       return agentInfoMap.get(agent);
   }
   public HashMap<String, String> loadAgent(){
      HashMap<String, String> agentMap=new HashMap<String, String>();
      agentMap.put("client1", "client1@ndsu.edu");
      agentMap.put("client2", "client2@ndsu.edu");
      agentMap.put("client3", "client3@ndsu.edu");
      agentMap.put("client4", "client4@ndsu.edu");
      agentMap.put("client5", "client5@ndsu.edu");
      agentMap.put("client6", "client6@ndsu.edu");
      agentMap.put("client7", "client7@ndsu.edu");
      agentMap.put("client8", "client8@ndsu.edu");
      agentMap.put("client9", "client9@ndsu.edu");
      return agentMap;
   }
   public HashMap<String, String> agentAddressInfo(String agent, String agentAddress){
      String address = agentAddress.substring(0, agentAddress.indexOf("@"));
      address += "@ndsu.edu";
      agentInfoMap.put(agent, address);
      return agentInfoMap;
   }
   private static HashMap<String, String> agentInfoMap = new HashMap<String,
String>();
}
```

LoadAgents.java

```
package jade.pmail.util;


import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;


/***
 * @author Hari Mukka
 */
public class LoadAgents {
   public HashMap<String, List<String>> loadAgent(){
      HashMap<String, List<String>> agentMap = new HashMap<String, List<String>>();
      agentMap.put("client1", agentList1());
      agentMap.put("client2", agentList2());
      agentMap.put("client3", agentList3());
      return agentMap;
   }
   public List<String> agentList1(){
      List<String> associatesList = new ArrayList<String>();
      associatesList.add("client2");
      associatesList.add("client4");
      associatesList.add("client6");
      associatesList.add("client8");
      return associatesList;
   }
    public List<String> agentList2(){
      List<String>  associatesList = new ArrayList<String>();
      associatesList.add("client1");
      associatesList.add("client5");
      associatesList.add("client7");
```

```
        associatesList.add("client9");
        associatesList.add("client11");
        return associatesList;
    }
    public List<String> agentList3(){
        List<String>  associatesList = new ArrayList<String>();
        associatesList.add("client1");
        associatesList.add("client3");
        associatesList.add("client7");
        associatesList.add("client9");
        return associatesList;
    }
}
```

Scenario-1. PMailFrame.java

```
/*
 * PMailFrame.java
 * Created on Jan 23, 2010, 8:38:13 PM
 */
package jade.pmail.gui;

import jade.lang.acl.ACLMessage;
import jade.pmail.util.Facilitator;
import jade.pmail.util.PMailActions;
import jade.pmail.util.LoadAgents;
import jade.pmail.util.PMailAgentBehaviour;
import jade.pmail.util.PMailMessages;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.HashMap;
import java.util.List;
import java.util.Vector;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeModel;

/***
 * @author Hari Mukka
 */
public class PMailFrame extends javax.swing.JFrame {

   /** Creates new form PMailFrame */

   private PMailAgentBehaviour pMailAgent;

   javax.swing.Timer timer = new javax.swing.Timer(5000, new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           refreshMail();
           pingAgentToCommunicate();
           associatedAgents.setModel(getAssociatesTreeModel());

           if(respondToCommunicate()){
                sendButton.setEnabled(true);
           }else{
              sendButton.setEnabled(false);
           }
        }
     });
    public PMailFrame(String agent) {
      agent_name = agent;
    pMailAgent = new PMailAgentBehaviour(agent_name);
    initComponents();
    timer.start();
  }

  public void refreshMail() {
     receivedMsgsTable.setModel(getRecievedMsgs());
     System.out.println("Rrefreshing the mail");
  }

  public boolean pingAgentToCommunicate()
  {
     if(pingAgentsMap!=null ){
        for (String agentName : pingAgentsMap.keySet()) {
           if(agentName.equalsIgnoreCase(agent_name)){
              String fromAgent = pingAgentsMap.get(agent_name);
              pingReqFrom.setText(fromAgent);
              return true;
           }
        }
     }
     return false;
  }

  public boolean respondToCommunicate() {
    System.out.println("agent:"+agent_name + "flag:"+enableSendFlag);
    if(respondAgentMap!=null){
       for (String respondToAgentName : respondAgentMap.keySet()) {
          if(agent_name.equalsIgnoreCase(respondToAgentName) && enableSendFlag
== 0){
             String toAgent = respondAgentMap.get(respondToAgentName);
```

```
            boolean isVisibleSend = true;
            System.out.println("Print agent name to respond:"+toAgent);
            System.out.println("Set this Value to Send button on
fromAgent:"+isVisibleSend);
            return true;
          }
        }
      }
      return false;
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * */
    private void initComponents() {

      pingReqFrom = new javax.swing.JTextField();

      setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
      setTitle("PMail Outlook");

      receivedMsgsTable.setModel(getRecievedMsgs());
      jScrollPane1.setViewportView(receivedMsgsTable);
      sentMsgsTable.setModel(getSendMsgs());
      jScrollPane2.setViewportView(sentMsgsTable);

      jTabbedPane1.addTab("Sent", jScrollPane2);

      associatedAgents.setModel(getAssociatesTreeModel());
      jScrollPane4.setViewportView(associatedAgents);
    public TreeModel getAssociatesTreeModel() {
    DefaultMutableTreeNode root = new DefaultMutableTreeNode(agent_name);
    LoadAgents agents = new LoadAgents();
    HashMap<String, List<String>> agentMap = agents.loadAgent();
    List<String> agentList = agentMap.get(agent_name);
    if(agentList != null ){
      for (String agent : agentList) {
        String ifAgentOnline = new Facilitator().getAgentAddress(agent);
        DefaultMutableTreeNode child;
        if(ifAgentOnline !=null){
          child = new DefaultMutableTreeNode(agent+" available");
        }else{
          child = new DefaultMutableTreeNode(agent+" offLine");
        }
        root.add(child);
      }
    }else {
      DefaultMutableTreeNode child = new DefaultMutableTreeNode("No associates");
      root.add(child);
    }
    TreeModel model = new DefaultTreeModel(root);
    return model;
  }

  private void sendButtonActionPerformed(java.awt.event.ActionEvent evt) {
    PMailActions action=new PMailActions();
    String sub=enteredSubject.getText();
    String to=enteredToAgent.getText();
    String msg=enteredMessage.getText();
    String response=action.compose(msg, to, agent_name, sub, pMailAgent);
    if(response.contains("failed"))
```

```
          JOptionPane.showMessageDialog(composePanel, response);
        sentMsgsTable.setModel(getSendMsgs());
        clearComposeFields();
        sendButton.setEnabled(false);
        btnPing.setEnabled(false);
}

    private void btnGetAddressActionPerformed(java.awt.event.ActionEvent evt) {
        String enteredAgent = enteredToAgent.getText();
        agentAddress = new Facilitator().getAgentAddress(enteredAgent);
        enteredAddress.setText(agentAddress);
        if(agentAddress == null || agentAddress.trim().equals("")){
          JOptionPane.showMessageDialog(null,
ACLMessage.AMS_FAILURE_UNAUTHORIZED);
        }else{
          btnPing.setEnabled(true);
        }
    }

    private void btnPingActionPerformed(java.awt.event.ActionEvent evt) {
        pingAgentsMap = pMailAgent.pingAgent(enteredToAgent.getText(),
enteredAddress.getText(), agent_name);
    }

    private void btnRefreshActionPerformed(java.awt.event.ActionEvent evt) {
          refreshMail();
          pingAgentToCommunicate();
          if(respondToCommunicate()){
               sendButton.setEnabled(true);
          }
          getAssociatesTreeModel();
    }

    private void pingReqYesActionPerformed(java.awt.event.ActionEvent evt) {
        String respondToAgent = pingReqFrom.getText();
        respondAgentMap = pMailAgent.pingAgent(respondToAgent,
enteredAddress.getText(),agent_name );
        //      respondToCommunicate();
        pingReqFrom.setText("");
        pingAgentsMap.remove(agent_name);
        enableSendFlag = 0;
    }

    private void pingReqNoActionPerformed(java.awt.event.ActionEvent evt) {
        JOptionPane.showMessageDialog(null,
ACLMessage.AMS_FAILURE_REQUEST_ERROR);
        pingReqFrom.setText("");
        pingAgentsMap.remove(agent_name);
    }
    public DefaultTableModel getSendMsgs() {


        List<PMailMessages> lst = pMailAgent.getSendList();
        Vector field = new Vector();
        field.add("Agent");
        field.add("Subject");
        field.add("Date");
        field.add("Message");

        Vector<Vector> rowData = new Vector<Vector>();
         for (PMailMessages pmailMessages : lst) {
            Vector data = new Vector();
            data.add(pmailMessages.getToAgent());
            data.add(pmailMessages.getSubject());
            data.add(pmailMessages.getDate());
            data.add(pmailMessages.getMessage());
            rowData.add(data);
        }

        DefaultTableModel model = new DefaultTableModel(rowData, field);
        return model;
    }

    public DefaultTableModel getRecievedMsgs() {
        Vector field = new Vector();
        field.add("Agent");
        field.add("Subject");
        field.add("Date");
        field.add("Message");

        Vector<Vector> rowData = new Vector<Vector>();

        HashMap<String, List<PMailMessages>> receivedMap =
pMailAgent.getReceivedMap();
        if(receivedMap.size() >0) {
            List<PMailMessages> receivedList = receivedMap.get(agent_name);
            System.out.println("agent name is:"+agent_name);
                if(receivedList != null && receivedList.size()>0 ) {
                    for (PMailMessages pmailMessages : receivedList) {
                        if(agent_name.equalsIgnoreCase(pmailMessages.getToAgent())) {
                            Vector data = new Vector();
                            data.add(pmailMessages.getFromAgent());
                            System.out.println("the from agent:"+pmailMessages.getToAgent());
                            data.add(pmailMessages.getSubject());
                    data.add(pmailMessages.getDate());
                    data.add(pmailMessages.getMessage());
                    rowData.add(data);
                }
            }
        }
    }
    DefaultTableModel model = new DefaultTableModel(rowData, field);
    return model;
  }
  public void clearComposeFields() {
    enteredToAgent.setText(null);
    enteredAddress.setText(null);
    enteredMessage.setText(null);
    enteredSubject.setText(null);
    enableSendFlag = 1;
  }
  private String agent_name;
  private static HashMap<String, String> pingAgentsMap;
  private static HashMap<String, String> respondAgentMap;
  private String agentAddress;
  private static int enableSendFlag = 0;
}
```

Scenario-2. PMailFrame.java

```
/*
 * PMailFrame.java*
 * Created on Jan 23, 2010, 8:38:13 PM
 */

package jade.pmail.gui;
```

```
import jade.lang.acl.ACLMessage;
import jade.pmail.util.Facilitator;
import jade.pmail.util.PMailActions;
import jade.pmail.util.LoadAgents;
import jade.pmail.util.PMailAgent;
import jade.pmail.util.PMailMessages;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.HashMap;
import java.util.List;
import java.util.Vector;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeModel;

/***
 * @author Hari Mukka
 */
public class PMailFrame extends javax.swing.JFrame {

   private PMailAgent pMailAgent;
   private DefaultMutableTreeNode root;
   javax.swing.Timer timer = new javax.swing.Timer(5000, new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           refreshMail();
           pingAgentToCommunicate();

           if(respondToCommunicate()){
```

```
          sendButton.setEnabled(true);
        }else{
          sendButton.setEnabled(false);
        }
      }
    });
  public PMailFrame(String agent) {
    agent_name = agent;
    pMailAgent = new PMailAgent(agent_name);
    root = new DefaultMutableTreeNode(agent_name);
    initComponents();
    timer.start();
  }

  public void refreshMail() {
      receivedMsgsTable.setModel(getRecievedMsgs());
      System.out.println("Rrefreshing the mail");
  }

  public void pingAgentToCommunicate()
  {
      if(pingAgentsMap!=null ){
         for (String agentName : pingAgentsMap.keySet()) {
            if(agentName.equalsIgnoreCase(agent_name)){
               String fromAgent = pingAgentsMap.get(agent_name);
               addressReqFrom.setText(fromAgent);
            }
         }
      }
  }
```

```
    public boolean respondToCommunicate() {
      if(respondAgentMap!=null){
        for (String respondToAgentName : respondAgentMap.keySet()) {
          if(agent_name.equalsIgnoreCase(respondToAgentName)&& enableSendFlag ==
0){
            enteredAddress.setText(new
Facilitator().getAgentAddress(enteredToAgent.getText()));
            return true;
          }
        }
      }
      return false;
    }

public TreeModel getAssociatesTreeModel() {
      LoadAgents agents = new LoadAgents();
      HashMap<String, List<String>> agentMap = agents.loadAgent();
      List<String> agentList = agentMap.get(agent_name);
      String available = "";
      if(availableAgents!=null && availableAgents.contains(agent_name))
        available = "Available";
      System.out.println("available - "+available);
        for (String agent : agentList) {
        DefaultMutableTreeNode    child    =    new    DefaultMutableTreeNode(agent+"
"+available);
          root.add(child);
        }
      TreeModel model = new DefaultTreeModel(root);
      return model;
    }
```

```
  private void sendButtonActionPerformed(java.awt.event.ActionEvent evt) {
     PMailActions action=new PMailActions();
     String sub=enteredSubject.getText();
     String to=enteredToAgent.getText();
     String msg=enteredMessage.getText();
     String response=action.compose(msg, to, agent_name, sub, pMailAgent);
     if(response.contains("failed"))
        JOptionPane.showMessageDialog(composePanel, response);
     sentMsgsTable.setModel(getSendMsgs());
     clearComposeFields();
     sendButton.setEnabled(false);


}

  private void btnGoFacToPingAgentActionPerformed(java.awt.event.ActionEvent evt) {
     String enteredAgent = enteredToAgent.getText();
     System.out.println("Entered Agent:"+enteredAgent);
     pingAgentsMap = new Facilitator().communicateAgent(enteredAgent, agent_name);
     if(enteredAgent == null || enteredAgent.trim().equals("")){
        JOptionPane.showMessageDialog(null,
ACLMessage.AMS_FAILURE_UNAUTHORIZED);
     }
     for (String string : pingAgentsMap.keySet()) {
        System.out.println(string);
     }
}
private void btnRefreshActionPerformed(java.awt.event.ActionEvent evt) {
     refreshMail();
     pingAgentToCommunicate();
     if(respondToCommunicate()){
          sendButton.setEnabled(true);
        }
    }

    private void pingReqYesActionPerformed(java.awt.event.ActionEvent evt) {
        String respondToAgent = addressReqFrom.getText();
        respondAgentMap = pMailAgent.pingAgent(respondToAgent,
enteredAddress.getText(),agent_name );
        respondToCommunicate();
        addressReqFrom.setText("");
        pingAgentsMap.remove(agent_name);
        enableSendFlag = 0;
    }

    private void pingReqNoActionPerformed(java.awt.event.ActionEvent evt) {
        JOptionPane.showMessageDialog(null, "Request Denied to communicate");
        addressReqFrom.setText("");
        pingAgentsMap.remove(agent_name);
    }

    private void sendButtonOnClick(java.awt.event.KeyEvent evt) {
        sendButton.setEnabled(false);
    }

    public DefaultTableModel getSendMsgs() {

        List<PMailMessages> lst = pMailAgent.getSendList();
        Vector field = new Vector();
        field.add("Agent");
        field.add("Subject");
        field.add("Date");
        field.add("Message");
        Vector<Vector> rowData = new Vector<Vector>();
         for (PMailMessages pmailMessages : lst) {
           Vector data = new Vector();
           data.add(pmailMessages.getToAgent());
           data.add(pmailMessages.getSubject());
           data.add(pmailMessages.getDate());
           data.add(pmailMessages.getMessage());
           rowData.add(data);
        }
        DefaultTableModel model = new DefaultTableModel(rowData, field);
        return model;
    }

    public DefaultTableModel getRecievedMsgs() {
        Vector field = new Vector();
        field.add("Agent");
        field.add("Subject");
        field.add("Date");
        field.add("Message");

        Vector<Vector> rowData = new Vector<Vector>();

        HashMap<String, List<PMailMessages>> receivedMap =
pMailAgent.getReceivedMap();
        if(receivedMap.size() >0) {
           List<PMailMessages> receivedList = receivedMap.get(agent_name);
           System.out.println("agent name is:"+agent_name);
              if(receivedList != null && receivedList.size()>0 ) {
                 for (PMailMessages pmailMessages : receivedList) {
                    if(agent_name.equalsIgnoreCase(pmailMessages.getToAgent())) {
```

```
                    Vector data = new Vector();
                    data.add(pmailMessages.getFromAgent());
                    System.out.println("the from agent:"+pmailMessages.getToAgent());
                    data.add(pmailMessages.getSubject());
                    data.add(pmailMessages.getDate());
                    data.add(pmailMessages.getMessage());
                    rowData.add(data);
                  }
                }
              }
            }
        DefaultTableModel model = new DefaultTableModel(rowData, field);
        return model;
      }
     public void clearComposeFields() {
        enteredToAgent.setText(null);
        enteredAddress.setText(null);
        enteredMessage.setText(null);
        enteredSubject.setText(null);
        enableSendFlag = 1;
      }
      void setAvailableAgents(List<String> agentsList) {
        this.availableAgents = agentsList;
      }
      private String agent_name;
      private List<String> availableAgents;
      private static HashMap<String, String> pingAgentsMap;
      private static HashMap<String, String> respondAgentMap;
      private static int enableSendFlag = 0;

    }
```

Scenario-3. PMailFrame.java

```
/*
 * PMailFrame.java *
 * Created on Jan 23, 2010, 8:38:13 PM
 */

package jade.pmail.gui;

import jade.lang.acl.ACLMessage;
import jade.pmail.util.Facilitator;
import jade.pmail.util.PMailActions;
import jade.pmail.util.LoadAgents;
import jade.pmail.util.PMailAgent;
import jade.pmail.util.PMailMessages;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.HashMap;
import java.util.List;
import java.util.Vector;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeModel;

/***
 * @author Hari Mukka
 */
public class PMailFrame extends javax.swing.JFrame {
   private PMailAgent pMailAgent;
   private DefaultMutableTreeNode root;
```

```
    javax.swing.Timer timer = new javax.swing.Timer(5000, new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            refreshMail();
            pingAgentToCommunicate();

            if(facilitator.getResponse() && enableSendFlag == 0){

enteredAddress.setText(facilitator.getAgentAddress(enteredToAgent.getText()));
                sendButton.setEnabled(true);
            }else{
                sendButton.setEnabled(false);
            }
        }
    });

    public PMailFrame(String agent) {
      agent_name = agent;
      pMailAgent = new PMailAgent(agent_name);
      root = new DefaultMutableTreeNode(agent_name);
      facilitator = new Facilitator(agent_name);
      initComponents();
      timer.start();
    }

    public void refreshMail() {
        receivedMsgsTable.setModel(getRecievedMsgs());
        System.out.println("Rrefreshing the mail");
    }

    public void pingAgentToCommunicate()
    {
        if(pingAgentsMap!=null ){
            for (String agentName : pingAgentsMap.keySet()) {
                if(agentName.equalsIgnoreCase(agent_name)){
                    String fromAgent = pingAgentsMap.get(agent_name);
                    addressReqFrom.setText(fromAgent);
                }
            }
        }
    }

public TreeModel getAssociatesTreeModel() {
        LoadAgents agents = new LoadAgents();
        HashMap<String, List<String>> agentMap = agents.loadAgent();
        List<String> agentList = agentMap.get(agent_name);
        String available = "";
        if(availableAgents!=null && availableAgents.contains(agent_name))
            available = "Available";
        System.out.println("available - "+available);
        for (String agent : agentList) {
            DefaultMutableTreeNode child = new DefaultMutableTreeNode(agent+"
"+available);
            root.add(child);
        }
        TreeModel model = new DefaultTreeModel(root);
        return model;
    }

    private void sendButtonActionPerformed(java.awt.event.ActionEvent evt) {
        PMailActions action=new PMailActions();
        String sub=enteredSubject.getText();
        String to=enteredToAgent.getText();
        String msg=enteredMessage.getText();
        String response=action.compose(msg, to, agent_name, sub, pMailAgent);
        if(response.contains("failed"))
          JOptionPane.showMessageDialog(composePanel, response);
        sentMsgsTable.setModel(getSendMsgs());
        clearComposeFields();
        sendButton.setEnabled(false);


}

    private void facProcessReqActionPerformed(java.awt.event.ActionEvent evt) {
        String enteredAgent = enteredToAgent.getText();
        System.out.println("Entered Agent:"+enteredAgent);
        pingAgentsMap = facilitator.communicateAgent(enteredAgent, agent_name);
        if(enteredAgent == null || enteredAgent.trim().equals("")){
          JOptionPane.showMessageDialog(null,
ACLMessage.AMS_FAILURE_UNAUTHORIZED);
        }
    }
    private void btnRefreshActionPerformed(java.awt.event.ActionEvent evt) {
        refreshMail();
        pingAgentToCommunicate();


    }

    private void pingReqYesActionPerformed(java.awt.event.ActionEvent evt) {
        String respondToAgent = addressReqFrom.getText();
        HashMap<String, Integer> reqFacAddressMap = new HashMap<String, Integer>();
        reqFacAddressMap.put(respondToAgent, 1);
        facilitator.setResponse(reqFacAddressMap);
        addressReqFrom.setText("");
    pingAgentsMap.remove(agent_name);
    enableSendFlag = 0;
}

private void pingReqNoActionPerformed(java.awt.event.ActionEvent evt) {
    JOptionPane.showMessageDialog(null, "Request Denied to communicate");
    addressReqFrom.setText("");
    pingAgentsMap.remove(agent_name);
}

private void sendButtonOnClick(java.awt.event.KeyEvent evt) {
    sendButton.setEnabled(false);
}

public DefaultTableModel getSendMsgs() {
    List<PMailMessages> lst = pMailAgent.getSendList();
    Vector field = new Vector();
    field.add("Agent");
    field.add("Subject");
    field.add("Date");
    field.add("Message");

    Vector<Vector> rowData = new Vector<Vector>();
     for (PMailMessages pmailMessages : lst) {
        Vector data = new Vector();
        data.add(pmailMessages.getToAgent());
        data.add(pmailMessages.getSubject());
        data.add(pmailMessages.getDate());
        data.add(pmailMessages.getMessage());
        rowData.add(data);
    }
    DefaultTableModel model = new DefaultTableModel(rowData, field);
    return model;
  }

  public DefaultTableModel getRecievedMsgs() {

    Vector field = new Vector();
    field.add("Agent");
    field.add("Subject");
    field.add("Date");
    field.add("Message");

    Vector<Vector> rowData = new Vector<Vector>();
    HashMap<String, List<PMailMessages>> receivedMap =
pMailAgent.getReceivedMap();
    if(receivedMap.size() >0) {
      List<PMailMessages> receivedList = receivedMap.get(agent_name);
      System.out.println("agent name is:"+agent_name);
        if(receivedList != null && receivedList.size()>0 ) {
          for (PMailMessages pmailMessages : receivedList) {
            if(agent_name.equalsIgnoreCase(pmailMessages.getToAgent())) {
              Vector data = new Vector();
              data.add(pmailMessages.getFromAgent());
              System.out.println("the from agent:"+pmailMessages.getToAgent());
              data.add(pmailMessages.getSubject());
              data.add(pmailMessages.getDate());
              data.add(pmailMessages.getMessage());
              rowData.add(data);
            }
          }
```

```
            }
        }

        DefaultTableModel model = new DefaultTableModel(rowData, field);
        return model;
    }
    public void clearComposeFields() {
        enteredToAgent.setText(null);
        enteredAddress.setText(null);
        enteredMessage.setText(null);
        enteredSubject.setText(null);
        enableSendFlag = 1;
    }
    void setAvailableAgents(List<String> agentsList) {
        this.availableAgents = agentsList;
    }
    private String agent_name;
    private List<String> availableAgents;
    private static HashMap<String, String> pingAgentsMap;
    private static int enableSendFlag = 0;
    private final Facilitator facilitator;
}
```